SESSION

WORLDCOMP + ERSA KEYNOTE

Chair(s)

TBA

How Engineering Mathematics can Improve Software

David Lorge Parnas[1],
[1]Middle Road Software, Ottawa, Ontario, Canada

Abstract - *For many decades computer science researchers have promised that the "Formal Methods" developed by computer scientists would bring about a drastic improvement in the quality and cost of software. That improvement has not materialized. We review the reasons for this failure. We then explain the difference between the notations that are used in formal methods and the mathematics that is essential in other areas of Engineering. Finally, we illustrate the ways that Engineering Mathematics can be useful in software projects*

Keywords: Software design, mathematics, formal methods

1 Introduction

There are two basic differences between the field known (euphemistically) as Software Engineering and the traditional Engineering fields such as Electrical, Mechanical and Civil Engineering.

- In Software Engineering, physical science is not as important as it is in other Engineering areas. Knowledge of physical science may be important for specific applications but it is not in the core body of knowledge for software development.
- Mathematics is not used by software developers in the way that it is used in the traditional Engineering disciplines.

The first of these differences is not surprising. Unlike radios, cars, and bridges, software is not a physical product. The second difference, is very surprising. In software, the laws of physics must be replaced by mathematical laws that determine the behaviour of programs [15]. When an Electrical Engineer constructs a device from resistors, inductors, and capacitors, it is routine to calculate the behaviour of the assembled circuit mathematically. Laws describing program composition could be used by software developers in the same way that Kirchoff's laws are used by Electrical Engineers but, they are not; most software developers rely on intuition and trials to determine the behaviour of the constructed program.

2 Roles of mathematics in Engineering

In Engineering, documentation is the design medium [9] and mathematics is used in that documentation. There are two distinct roles for documents, description and specification.

- Documents can be used to describe properties of a product that exists (or previously existed).
- Documents can be used as specifications to state properties that are required of a product. The product specified might not exist.

The difference between a specification and other descriptions is one of intent, not form. Every specification that a product satisfies is a description of that product. The only way to tell if a description is intended to be interpreted as a specification is what is said about the document, not its contents or notation. I consider the phrase "specification language" to be nonsense. Any notation that can be used to produce specifications can also be used to produce descriptions.

Because documents contain mathematical expressions, they can be used to verify the correctness of designs or compute properties of a product.

3 Early approaches to using mathematics in software development

Many computer scientists have proposed ways to use mathematics or mathematical notation to help in program development. It is more than 40 years since the late Robert Floyd [6] showed us how to "assign meaning to programs" and how we could verify that programs will do what they are intended to do. It is at least 35 years since I first heard Jean-Raymond Abrial present the ideas that were the basis of Z and its many dialects. The VDM community began its work about the same time. Dijkstra [4] showed how to use predicate transformation rules shortly after that. Mills (and others) showed us how the classical mathematical concepts of a relation could be used to do the same things [15].

None of these approaches has had the effect on software development practice that was sought. Since 1967, there have been numerous "revolutions" on the hardware side and amazing improvements in man-machine interfaces. The computer systems on my desk today were unimaginable when Bob Floyd wrote his 1967 article. Unfortunately, there hasn't been comparable progress in formal methods. There have been new languages and new logics, but the program design errors we saw in 1967 can still be found in today's software. Paradoxically, successful applications of formal methods to industrial practice remain such exceptions that people write papers about them, thereby confirming that the use of the method is not common.

4 Claims of progress and adoption

The CS research literature reveals that 'formal methods for software development' are a very popular research area. Variants of the most popular approaches are frequently discussed at conferences and in journals.

Research funding agencies often require larger projects to involve cooperation with industrial organizations and to demonstrate the practicality of an approach on "real" examples. When such efforts are reported in papers, they are almost invariably presented as successful. Paradoxically, these success stories reveal the failure of industry to adopt formal methods as standard procedures; if use of these methods were routine, papers describing successful use could not be published. Industry is so plagued by errors and high maintenance costs that they would be eager to use any method they thought would help; it says something that the "successes" have not been followed up by requiring that the methods be routinely used.

Often, reports of successful industrial adoption do not stand up to scrutiny. Sometimes, the authors are just playing with words. For example, the technique of placing debugging statements in code, which was taught to me in 1959, has recently been trumpeted as "industrial use of assertions".

Close scrutiny of an effort to demonstrate the utility of formal methods often reveals truly heroic efforts to develop and verify complex formal models but little evidence that the actual code is correct. These efforts rarely lead to repeat use or broader adoption of the method.

Some of the reported successful trials may be attributed to the simple method of having two people solving a problem and critiquing each other's code. Thirty years ago, in a paper that is still worth reading today, Elovitz [5] described an experiment in which a program in a programming language was used in the way that formal method advocates suggest that their notations be used. One programming language was dubbed the specification language; the other identified as an implementation language. A program was written by one programmer in the specification language and given to another as a "specification" for a program to be written in the implementation language. The translator often found, and corrected, errors. The "implementation was reviewed by the programmer who had written the specification. The error rate was measurably reduced; the technique (an early version of pair programming) was considered successful. This "dual scrutiny" effect, could explain many of the reported successful applications of formal methods.

Reports that formal methods are ready for industrial use must be taken with a huge grain of salt. The need for effective methods is so great that, if the well-known methods were ready, their use would be widespread.

5 Formal methods and engineering mathematics - contrasts

Noting the widespread use of mathematics in traditional Engineering, and the failure of software developers to use the "formal methods" that were developed to help them, we discuss the differences between formal methods and applied mathematics.

On many occasions, when I have remarked that I advocate the use of mathematics in software development, but oppose "formal methods", I have sensed reactions ranging from puzzlement to amusement or annoyance. The distinction is not obvious to many in the field.

This section presents some differences between mathematics and formal methods and explains their importance.

5.1 Mathematics

Standard dictionary definitions of mathematics such as, "*the abstract science of number, quantity, and space*" tell us more about the origin of mathematics than about mathematics as it is understood today. Mathematics is no longer restricted to "number, quantity, and space";it has been extended to include strings, physical structures, and many other types of objects in our world. Abstract structures defined long ago have proven to be useful in understanding things that were devised quite recently.

Mathematicians work by defining abstract structures and then studying their properties. The structures may be very basic and general such as sets, and relations or more complex, algebraic structures comprising several sets and functions mapping between them. A structure is defined by describing properties of the set members and the functions. Mathematicians may derive further properties from the properties given in the definitions.

Broad classes of structures may be divided into smaller classes by stating additional properties. For example, we may define the broad class *relations* and then define the additional properties of functions and various narrower classes such as functions that can be represented by polynomial expressions, boolean functions, or trigonometric functions.

Mathematical structures are:

- **Abstract**: Mathematical definitions do not refer to the real-world objects or structures that inspired them. The properties are all formally stated without any mention of observed behaviour in the real world.
- **Static:** Mathematical structures do not change with time. Some statements may appear to be describing change. For example, we may illustrate set union with examples like {2,3} ∪ {3,4} = {2,3,4}. This is sometimes be paraphrased as "if you add the set {3,4} to the set {2,3} the result is the set {2,3,4}." Such a sentence

sounds as if it is describing an event or action but the sets are all static; the equation states a permanent property of set union.

- **Precisely defined**: Classical mathematical definitions are mature (well structured, well understood, widely accepted). All conclusions about the structures have been derived from the definitions.

We shall refer to these three properties as the ASP properties. These three properties are very important.

- **Abstract** definition makes it possible to apply research results to many different types of systems. For example results from research motivated by the study of systems of springs and weights, may be applied to electrical circuits or control systems without any change to the mathematics. The same equations arise; the same solution techniques work.
- **Static** structures are easier to study. The idea of time in a dynamic system is often confusing in a way that static structures are not. The fact that the structures are static does not mean that we cannot use mathematics to analyze and design dynamic systems. Many of the most useful mathematical results are about dynamic systems. These results treat time in the same way that they treat other variables.
- **Precise Definitions** are essential for productive discussions of complex products. Without them discussions will be full of misunderstandings and the meaning of agreements will be disputed.

5.2 Formal Methods

The phrase "formal methods" was introduced by computer science researchers to describe a class of approaches to studying the properties of computer programs. The two best known are Z and VDM, but there are many variations of these as well as many other methods that come under this rubric. Among these are B (developed by the originator of Z), SOFL, LOTOS, Larch, SDL, TLA, CCS, SDL, …, . Some would include UML as a formal method but many note that it is not formally defined. A list of all the, notations, the variations and their associated methods is beyond the scope of this paper.

Formal methods comprise (1) a notation (often called a language) for describing or specifying computer systems and (2) procedures for using the language to study a computer system.

Another set of methods introduced for analyzing software introduce little or no new notation. Instead, they show how to convert hypotheses about computer system to theorems which can then be proven using standard mathematical techniques. Paradigmatic of this class of approaches was the pioneering work of Robert Floyd [6] and his students. Other such approaches will be discussed below.

In this paper, I use the term "formal methods" for methods that are associated with a language or special notation for software. The others will be called "mathematical methods".

5.3 Are "formal methods" and mathematics the same?

The languages used in formal methods resemble mathematics. Many of the symbols used are symbols used in classical mathematics and the formulae have similar interpretations.

The similarities between formal methods and classical mathematics are clear when one reads tutorials or books explaining those languages. The introductions almost invariably begin by presenting basic ideas about set theory and logic (often, as if they were new) and introduce notation that is a syntactic variation of the notations used in mathematics. Concepts like set union, conjunction, implication, etc. are explained using the notation of the method's language.

However, the well-known formal methods do not have the ASP properties.

- Formal Method notations were developed for the specific purpose of describing the behaviour of computer programs. In many of the published examples there is a frequent domain change between the physical situation (the program) and mathematical symbol manipulation. The notions of state, programming language variables[1], and actions (state changes) are basic concepts in the formal method languages. The notations are chosen to make the description of programs more direct. Many of the "specifications" are actually descriptions of abstract state machines whose behaviour mimics that of a program that would satisfy the specification.
- The notion of time as "something special" is built into these languages and methods as they talk about values before and after operations (which change states). In some cases, the languages incorporate specialized logics known as temporal logics. The structures described are not static.
- Many of the languages were introduced without a full formal definition; in some cases there have been efforts to correct the lack of a precise abstract definition long after the language was introduced. Variant definitions are frequently introduced in the literature.

The developers of these methods deviated from classical mathematics in this way because they

[1] These are different from mathematical variables - see below.

believed that this was necessary to make their ideas practical.

5.4 Modeling

Recent years have seen a renewed interest in the use of models and the coining of the phrase, “model-driven engineering.”

A *model* of a product is a simplified version of that product. There are two kinds of models: physical and abstract.

Although, the popular formal methods are often described as specification languages, it would be more accurate to call them modelling languages. Many of the published examples are neither specifications nor descriptions but models.

The properties of a model are never the same as the properties of the actual system. Consequently, models must be prepared and used with great care. Decisions based on models can be wrong if they are derived from properties of the model that are not properties of the actual product.

6 Abstractions and their representation

Mathematics is the study of abstract structures. The characteristics of these structures are described by axioms or equations. The structure cannot be seen; we can only see examples.

To be useful, a structure (e.g. a function) must be described. A mathematical expression (classically, a string) that that describes the structure is called a *representation* of the object. Thus, when we write

“$f(x)= x+1$”,

the string in quotes is not the function called f but a representation of that function. A given abstract object can have many possible representations. For example,

“$f(x) = 1+x$”, and

“$f(z) = 2\times z - z + 1$”

are different representations of the same function. Much of mathematics deals finding simpler representation of functions or determining whether two expressions represent the same function.

In natural language discussions of mathematics and its applications, there is an unfortunate tendency to confuse the abstraction with one of its representations. For example, we often see “predicate” referring to an expression that describes a predicate.

Some classes of abstract objects are defined by whether or not they can be represented in a certain form. For example, both the class of <u>expressions</u> of the form:

$$f(z) = a_0 + a_1 \times z + a_2 \times z^2 + \dots a_n \times z^n$$

and the class of <u>functions</u> that can be represented (exactly) by expressions in that form are sometimes called polynomials. The function represented by

$$f(z) = (z+1) \times z$$

is a polynomial function even though the expression is not a polynomial, because that function can also be represented by the expression

$$f(z) = z^2 + z,$$

which is a polynomial expression.

Finding the best representation for performing a mathematical task is an essential step in applying mathematical results. For example, finding the roots of the function described above is (slightly) easier using the first representation than using the polynomial expression.

The issue of representations becomes important to mathematics researchers whenever new applications require them to work with a new class of structures. The use of mathematics to describe the behaviour of software requires us to represent functions with many points of discontinuity. Classical engineering deals primarily with continuous functions or piecewise-continuous functions with a small number of discontinuities. Section 10 discusses the representation of software with many discontinuities.

7 How mathematics is applied to physical systems

To apply mathematical results we have to find a way to connect the abstraction to the physical system that we want to study. This is usually done by using variables. The word “variable” is used in both mathematics and physics, but the meaning is different.

7.1 Variables in physics

In physics, “variable” refers to a measurable characteristic of a system whose value changes with time. To define a variable one must describe a way that it can be measured and the units that will be used. This allows a scientist to make statements about the value of this variable. Usually a variable is identified by a short character string such as “h” or “height”. Often the name used to identify the variable is not distinguished from the variable itself.

7.2 Variables in mathematics

In pure mathematics, variables are place holders used in the definition of relations. When used for this purpose, they do not have values. The variables have no meaning outside of the definition. For example, we can write

“doublesum(x,y) = x+y+ x+y” .

If we write,

“doublesum (w,z) = w+z+ w+z” ,

we have defined exactly the same function. The variables are just a convenient way of saying, “the first argument”, “the second argument”, etc. As in physics, a variable is usually represented by a short string.

Mathematical variables are used again when the function is applied. If we want to apply the function to specific values, we may write

doublesum(x,z) for x=2 and z = 3, or
doublesum(2,3).

A list of specific values for the arguments such as (2,3) is often called an assignment of values for (x,z) but mathematical variables do not have values outside of the expression, i.e. the “assignment” has no lasting effect on the .variable.

In informal discussions, there is a natural tendency to confuse the variable with the set of values that we intend to assign to it. For example, taken literally, the statement, “x is an integer”, where x is a variable is actually nonsense. A constant such as “2” is an integer but x is a variable not a number. Usually such a statement is intended say that the variable will only be assigned integer values or as a part of an informal conditional statement.

7.3 Associating physical variables with mathematical variables

When mathematics is used to study or describe of physical systems, relations are applied by associating the mathematical variables with physical variables. Usually this is done by a kind of pun, i.e by giving the mathematical variable and the associated physical variable the same name. For example we may write

doublesum(length, width)

where length and width are the names of defined physical variables.

When this punning is used, it may convey the (false) impression that the mathematical variables are the same as the physical variables and some practitioners ignore the distinction. In mathematical reasoning any association between the physical variable and the mathematical variable is irrelevant and should not be used.

7.1 Time as a variable

Time receives no special treatment in classical applied mathematics. In Physics, and all branches of Engineering that deal with dynamic systems, time is treated in the same way as other physical variables.[2] This allows us to predict the behaviour of moving objects, electrical circuits, control systems, etc. In Engineering and Physics there is no need for temporal logic to describe and analyze the dynamic systems that are ubiquitous in these fields.

7.2 Transfer functions

In traditional Engineering, a device’s behaviour is characterized by a function with a domain containing the possible histories of the observable behaviour (usually, the input history suffices) and a range that contains the value of the device’s outputs. The history contains information assumed to be known or controllable and the range represents the information we want to know. Transfer functions can describe the behaviour of the device without providing any information about its construction.

Given a network of devices, each described by its transfer function, it is possible to derive the transfer function for the network. This supports a hierarchical documentation and analysis process in which networks can be encapsulated and their transfer function (which is usually simpler than the full network description) used in analysis of the networks of which it is a part. [14]

Analysis allows the detection of anomalous behaviour, such as resonant frequencies or other types of behaviour, that might not be revealed by testing.

8 Mathematical reasoning:

Three distinct forms of reasoning can be observed in the development and application of mathematics.

- function evaluation
- predicate satisfaction
- Inference or deduction

8.1 Function evaluation

Function evaluation begins with an expression that represents a function whose domain is the set of possible values of the variables appearing in the expression and range is the set of possible values of the function. Evaluation proceeds by substituting assigned values for the variables, and evaluating functions when the value of all of its arguments have been computed. The order of evaluation is not fully determined but, when there is a choice, the result is not affected by the order chosen.[3]

8.2 Predicate satisfaction

Predicate satisfaction begins with an expression that represents a function with range {***true***, ***false***}. However, no values are assigned to the variables. Instead, there is a search for an assignment that the function maps to ***true***. Theoretically, the process could be a random search but research has found faster methods. There may be many assignments that would

[2] One can consider time to be he fourth dimension of a 4 dimensional space with the first three coordinates taken from a conventional spacial coordinate system.

[3] Readers are reminded that we are discussing mathematical expressions, not programs. There are no side-effects.

map to ***true***; the order in which assignments are tried can affect the result. In traditional engineering the most common form of the predicates is a set of equations. Predicate satisfaction is then called solving equations.

8.3 Inference or deduction

Logic provides a sound basis for mathematics. A particular logic comprises a set of axioms (predicates assumed to be true) and rules of inference (rules that can be used to transform a predicate expression to another expression). Anything inferred in this way from the axioms is considered to be true. If an expression is hypothesized to be a true, proof procedures search for a sequence of inferences that will demonstrate its truth.

8.4 Mathematical reasoning in Engineering.

Of the three forms of mathematical reasoning outlined above, Engineers prefer the first. If they are given equations, they will try to "solve" the equations to produce what is often called a "*closed form solution*", which is a function that maps from a domain consisting of the possible values of the known (independent) variables, to a range comprising the values of the variables that are not known but needed.

The reason for preferring function evaluation is that inference and predicate satisfaction generally involve a search. In the case of inference it is a search for the right sequence of inference applications. In the case of predicate satisfaction it is a search for the right assignment. In function evaluation there is no need for a search. One is given the values of the independent variables and evaluates functions in an obvious order.

Many Engineering mathematics programmes include little or no discussion of inference. Much of the mathematics curriculum is devoted to solving equations to find expressions that can be evaluated. Previously found solutions are taught in the engineering courses.

Function evaluation is the primary way that mathematics is used in practical engineering applications. Mathematicians use inference or deduction to develop new mathematical results and proof techniques. Predicate satisfaction is used for problems where no "closed form" solution is known.

9 Applying mathematics to software systems

This section discusses applying the methods used in older areas of Engineering in software development.

9.1 Variables and identifiers in programs

Above, I have pointed out that the meaning of "variable" in mathematics is different from its meaning in physics. In discussing programs, the word has yet another meaning.

Variables in programs are discrete state machines that serve as memory. They are generally finite state machines; the upper bound to the number of states may be very large and, for some implementations, hard to characterize. Unlike variables in mathematics, program variables have a value, their state, which can persist over time. They are not placeholders used for definition and application of functions. They are closer to physics variables than the mathematical ones because they correspond to a physical device and the values are observable characteristics of that device [4].

Because of the nature of many programming languages, it is important to distinguish between a variable and a string that is used to identify it (an identifier). In modern programming languages, it is possible for a variable to have several identifiers (aliasing) or for one identifier to be used to identify different variables in different parts of a program.

Because the relation between identifiers and variables is not 1:1, the "punning" method of associating mathematical variables with the program variables can cause difficulties. It is easy to replace identifiers that have been used more than once with unique names for the variables, but aliasing is not as easily avoided. The methods available to deal with aliasing depend on the design of the language. Argument passing in procedures is the main source of the aliasing problem as one can use one variable as an actual parameter corresponding to several distinct formal parameters. Unless one consistently deals with an array as a single variable, array indexing is also a source of aliasing. Another form of aliasing is caused by the use of pointer variables.

The sequel assumes that this problem has been solved in some way.

9.2 Assertion based methods

The best known, and most widely taught, methods of relating software to mathematics is the approach introduced by Robert Floyd, [6]]. Predicates are associated with points in the program text; predicate transformation rules are given for each type of statement in the language. Starting with a predicate that is assumed to be true before the program is executed, and following each path, one assumes that the assertion before a statement holds and must prove that the assertion after that statement will hold after execution. Demonstrating the correctness of a program is, thereby, reduced to proving a set of mathematical theorems.

Looking at proofs using this basic method one can make the following observations:

[4] Algol 60 introduced the distinction between variables (which were declared) and formal parameters (which were specified). Formal parameters were similar to the variables used in mathematics.

- In the proof examples, it is common to move back and forth between the program domain (where one exams the paths in an intuitive way) and a mathematical domain (where theorems are proven). It is possible to separate the two types of work by generating all proof obligations from the program text before moving into the mathematics domain. This may require backtracking if one did not introduce assertions that are strong enough.
- Abstraction is not supported. As one moves through the program text, one carries all state variables in the expressions so that some are mentioned in expressions even where they are not relevant.
- If code is reused, proofs may be repeated.
- Although it is natural to go forward, there is no requirement to do so and one could also go backward.
- Although it is common to regard the precondition and postcondition as separate, it was often necessary to add variables that had no other purpose than to support the proof.
- The process neither assumes nor exploits any hierarchical structure in the text.

Floyd did not introduce a new notation for his method; none was needed. In a slightly later paper, Hoare introduced what has become known as the "Hoare triple", that comprised a precondition pattern, a statement pattern, and a postcondition pattern. The triple was a "schema" describing the way that a predicate expression describing the precondition would be transformed to a predicate expression describing the postcondition. The rules given are valid only if certain assumptions about the variables hold. In particular, the relation between variables and identifiers must be 1:1 and that program variable identifiers cannot appear as bound variables in the predicate expressions.

9.3 Predicate transformer methods

Dijkstra [4] replaced the Hoare triple with a function from predicate expressions to predicate expressions, which he called a *predicate transformer.* In addition, he decided to reverse the direction, i.e. his predicate transformers transformed a postcondition to a precondition. He also took the problems of non-determinism into account and identified two distinct predicate transformers, known as the *weakest precondition* and *weakest liberal precondition.* For a deterministic program, these would be the same but for a non-deterministic program they could differ. Predicate transformers, like the earlier methods, did not abstract from the representation of the state. They were always expressed in terms of a specific form of predicate expressions; Dijkstra also assumes a 1:1 mapping between program variables and mathematical variables.

9.4 Abstract relational/functional methods

Many mathematicians, among the best known were Mills [15] and de Bruijn [2], pointed out that new concepts were not needed because the classical mathematical concept of relation[5] could be used A deterministic program computes a function from starting state to stopping state. For non-deterministic programs, the mapping may be a relation that is not a function. In the non-deterministic case, it is either necessary to introduce a way to denote non-termination or use the approach like that in [16].

Expressing the mathematical properties of programs with functions on states has a number of advantages:

- There is no need for new notation. This is not new mathematics; it is simply a new application for old mathematics.
- The definitions given in papers like [15,16] are independent of the way that the states and functions are represented.
- The definitions are simpler and, as a consequence, easier to understand. The simplicity is the result of abstracting from representation details.
- There is no need for a special treatment of time. Time is part of the state information and the passage of time during the execution can be treated in exactly the same way as changes of other variables. If the execution time is difficult to predict, time consumption can be considered part of the non-determinism.
- The definitions given in papers like [15] are not tied to a specific set of statement types. They distinguish two components of a programming language, primitive (built in "statements" or programs) and constructors (such as "**;**" "**if then else**", and "**while**") which are used to construct bigger programs from primitive programs and previously constructed programs. This eases their application to new programming languages.
- The absence of assumptions about the primitive programs supports hierarchical application of the laws. Any constructed program can be treated as a primitive program when constructing larger ones.

Relational methods are the closest of the software methods to the methods used in classical Engineering. In the author's opinion, they have many advantages. For example, no changes are required to use the tabular notations defined in [13]. The mathematics is applicable to a wide variety of programming languages including those with unusual data types. Using this approach facilitates a smooth integration of Software Engineering and traditional Engineering.

[5] Mills work [15] was restricted to deterministic programs and relations that were functions.

9.5 Additional views of software

Early work on the application of mathematics to programs only considered individual sequential deterministic, terminating programs. This was adequate for the programs of concern in the 1960s but it is not adequate for the type of software we build today. For today's systems, we need to view software from many "orthogonal" viewpoints. Each view can be described in a separate document as summarized in Figure 1.

Document	Content
Software Requirements Document	Black Box specification, identifies all outputs and inputs and describes relation between output values and input history.
Module Guide	Informal description showing the hierarchical decomposition of the system into modules and the secret of each module.
Module Interface Specifications	Black Box specification, identifies all outputs and inputs and describes relation between output values and input history." Usually shows externally invokable programs.
Module Implementation Design Document	Documents complete data structure, effects of all externally visible programs on data, abstraction function or relation (data interpretation) - design can be verified before coding begins.
Program Uses Structure	Description of permitted usage of one access-program by another. Determines the usable subsets of product.
Display Method Program Documentation	Hierarchical decomposition of program into "small programs" with specification of subject program and programs used.

Figure 1: Software Design Views

9.6 Applying mathematics for the other views

None of the well-known formal methods was designed with views other than individual terminating programs in mind. Because each method was primarily a language there have been a few attempts to apply the notations to other views but the result is not convincing. In fact, the communities behind these methods do not seem to understand the need for separation of concerns and a multi-view approach.

The relational approach, can be used for each of the views. Each of the views can be documented using mathematical relations but the range and the domain of the relations varies with the view. The contents of each document other than the module guide, which is the only informal document, will be a representation of a relation. [17]. Figure 2 summarizes the relations to be represented in each of the documents.

Document	Domain	Range	Relations
Software Requirements Document	Histories of I/O	Output values	One per output variable: possible value after history
Module Interface Specifications	I/O traces (sequences)	Output values	One per output variable possible value after trace
Module Implementation Design Document Domain	Declaration of complete data structure		
Module Implementation Design Document part 2	Data Structure States	set of traces	Abstraction Relation: Trace could lead to state
Module Implementation Design Document 3	Access Program Function	Data Structure States	Data Structure States
Program Uses Structure	Access Programs	Access Programs	Allowed to Use
Display Method Program Documentation	program start states	program end states.	program functions (with text)

Figure 2: Relational Content of Primary Software Design Documents

With a multiple view approach to software, it becomes extremely important to have clear statements of the contents of each document. Many developers have witnessed destructive arguments about whether some detail should be included in a given document or belongs elsewhere. More have witnessed misunderstandings because the information was discussed in more than one document and the documents were not consistent. Mathematics gives us a way to say precisely what belongs in each document without specifying format or notation [16].

It is also possible to use mathematics to verify key design decisions before spending time and money on implementation of faulty interfaces. Methods used in other areas of Engineering can be used to check higher-level documents before investing in code. [14]

10 Representing Software Relations

The mathematical methods that have been used in traditional Engineering fields have not been easy to apply to software because the classical representations are not suitable for the functions that must be described. While the functions that arise when working with physical products are usually either continuous or have a small number of discontinuities, the functions that describe software have many discontinuities, both because we are dealing with digital systems and because the power of software lies partly in its ability to implement behaviour that has many special cases.

Mathematical expressions that describe computer systems can become very complex, hard to write and hard to read. When software functions are described by expressions in conventional format, the depth of nesting of subexpressions gets to high. As first demonstrated more than 30 years ago in [8, 7], the use of a tabular format for mathematical expressions can turn an unreadable string of symbols into an easy to access, complete and unambiguous document.

Figure 3 [6] is an expression that describes the behaviour of a keyboard checking program that was developed by Dell in Limerick, Ireland [1, 18]. Even those who are mathematically inclined find such expressions hard to read and write.

Keyboard Checker: Conventional Expression

```
(N(T)=2∧keyOK∧(¬(T=_)∧N(p(T))=1))∨(N(T)=1∧(T=_∨(¬(T=_)∧N(p(T))=1))∧
(¬keyOK∧¬prevkeyOK∧¬prevkeyesc))∨((¬(T=_)∧N(p(T))=1)∧
((¬keyOK∧keyesc∧¬prevkeyesc)∨(¬keyOK∧keyesc∧prevkeyesc∧
prevexpkeyesc))∨((N(T)=N(p(T))+1)∧(¬(T=_)∧(1<N(p(T))<L))∧(keyOK))∨
((N(T)=N(p(T))-1))∧(¬keyOK∧¬keyesc∧(¬prevkeyOK∧prevkeyesc∧
preprevkeyOK)∨prevkeyOK)∧((¬(T=_)∧(1<N(p(T))<L))∨(¬(T=_)∧N(p(T))=L)))∨
((N(T)=N(p(T)))∧(¬(T=_)∧(1<N(p(T))≤L))∧((¬keyOK∧¬keyesc∧(¬prevkeyOK∧
prevkeyesc∧¬preprevkeyOK))∨(¬keyOK ∧¬prevkeyOK∧ ¬prevkeyesc)∨
(¬keyOK∧keyesc∧¬prevkeyesc)∨(¬keyOK∧keyesc∧prevkeyesc∧
prevexpkeyesc))∨((N(P(T)=Fail)∧(¬keyOK∧keyesc∧prevkeyesc∧
¬prevexpkeyesc)∧(1≤N(p(T))≤L))∨((N(P(T)=Pass)∧(¬(T= _)∧N(p(T))=L)∧(keyOK))
```

Figure 3: Characteristic Predicate of Keyboard Checker Program

[6] Auxiliary predicates such as keyesc, keyOK, etc. are defined separately. Each is simply defined.

Figure 4 is a tabular mathematical expression describing the same function. It comprises 3 elements called grids. Each grid contains a number of cells. For this type of expression, the top grid and the left grid are called “headers”. The lower right grid is the “main grid”.

To use this type of tabular expression, one evaluates the cells the headers to find the ones that evaluate to ***true***. If the table has been properly constructed, exactly one of the column header cells and one of the row header cells will evaluate to ***true*** for any assignment of values to the variables. The indices of those cells identify a row and column. Together they identify one cell in the main grid. Evaluating the expression in that cell will yield the value of the function.

A tabular expression parses the conventional expression for the user. Instead of trying to parse, evaluate or understand the complex expression in figure 3, one looks at the simpler expressions that appear in the tabular expression. Generally, one will only need to evaluate a few of the expressions that appear in the cells to compute an answer. Note that although the tables use predicate expressions, the user is evaluation functions, not using predicate satisfaction or inference.

There are many forms of tabular expressions. The grids need not be rectangular. The format is not limited to two-dimensional grids or tables. Any expression in any cell can itself be a tabular expression. A variety of types of tabular expressions are defined in [13].

N(T) =

			T=_	¬ (T= _) ∧		
				N(p(T))=1	1<N(p(T))< L	N(p(T))= L
keyOK				2	N(p(T))+ 1	Pass
¬keyOK ∧	¬keyesc ∧	(¬prevkeyOK ∧ prevkeyesc ∧ preprevkeyOK) ∨ prevkeyOK			N(p(T)) - 1	N(p(T)) - 1
		¬prevkeyOK ∧ prevkeyesc ∧ ¬preprevkeyOK			N(p(T))	N(p(T))
		¬prevkeyOK ∧ ¬ prevkeyesc	1	1	N(p(T))	N(p(T))
	keyesc ∧	¬ prevkeyesc		1	N(p(T))	N(p(T))
		prevkeyesc ∧ ¬prevexpkeyesc		Fail	Fail	Fail
		prevkeyesc ∧ prevexpkeyesc		1	N(p(T))	N(p(T))

Figure 4: Tabular Expression for Figure 1

Although tabular expressions were successfully used without proper definition for many years, precise semantics are needed for tools and full analysis. There have been four basic approaches to defining the meaning of these expressions. Janicki and his co-authors developed an approach based on information flow graphs that can be used to define a number of expressions.[11]. Zucker based his definition on predicate logic.[19] . Khédri and his colleagues based their approach on relational algebra [3]. All three of these approaches were limited to basic forms of tables [12]. The most recent approach, [13], is less restricted; it defines the meaning of these expressions by means of translation schema that can be used convert any tabular expression of a known type to an equivalent conventional expression. This is the most general approach and provides a good basis for tools. The appropriate table form will depend on the characteristics of the function being described. Jin shows a broad variety of useful table types and which provides a general approach to defining the meaning of any new type of table [13].

This newer approach to tabular expressions makes it possible to chose a function representation that is intuitive and easy for its intended audience to use without losing the precision of mathematics. Both commercial and academic prototype tools have demonstrated the ability to use this type of representation to check information for completeness and consistency as well as to check correctness by automatic construction of a prototype that can be used in testing.

Perhaps the most important thing to note about tabular methods in software development is that tabular expressions are “closed form” expressions and can be used for function evaluation; this is easier than using the inference approach, which is the most common approach used in formal methods.

11 Conclusions

Software developers and educators must ask a simple question, “Why is mathematics, which has often been shown to be an essential tool in traditional engineering, so seldom used in software development? Computer Scientists have made three basic mistakes.

- They based their “formal methods” on the way that mathematicians and philosophers develop new mathematics, rather than the way that Engineers have applied mathematics to design products.
- They failed to recognize that mathematicians always have developed new forms for expressions whenever they study a new class of functions. The functions that describe software behaviour are quite different from those encountered when describing physical products; new notation was needed.
- They failed to think deeply about the roles that mathematics could play in software development, i.e in specifications, in descriptions, and in verification. For the most part, they thought only about verification and failed to distinguish between, modelling, description and specification.

We now have a clearer picture of the ways that mathematics can and should be used, and a much better notation for describing functions and relations. It is time for software developers to act more like traditional Engineers and make use of the powerful mathematical tools available to them.

12 References

[1] Baber, R., Parnas, D.L., Vilkomir, S., Harrison, P., O'Connor, T., "*Disciplined Methods of Software Specifications: A Case Study*", Proceedings of the International Conference on Information Technology Coding and Computing (ITCC 2005), April 4-6, 2005, Las Vegas, NV, USA, IEEE Computer Society.

[2] de Bruijn, N. G., "*The Mathematical Language AUTOMATH – Its Usage And Some Of Its Extensions*" In Symposium on Automatic Demonstration, volume 125 of Lecture Notes in Mathematics, pages 29–61. Springer-Verlag, 1970.

[3] Desharnais, Jules, Khédri, Ridha; Mili Ali, "*Towards a Uniform Relational Semantics for Tabular Expressions*" *Proceedings of RelMiCS* 1998, pp. 53-57

[4] Dijkstra, E.W., "*A Discipline of Programming*", Prentice Hall, Englewood Cliffs, NJ, 1976

[5] Elovitz, H. S., "An experiment in software engineering: *The Architecture Research Facility as a case study*", Proceedings of the 4th International Conference on Software engineering Munich, Germany, Pages: 145 - 152, 1979

[6] Floyd, R.W., "*Assigning Meanings to Programs*", Proceedings of the Symposium of Applied Mathematics, Vol.19, 1968. Also in: Schwartz, J.T. (editor), Mathematical Aspects of Computer Science, American Mathematical Society, pp. 19-32, 1967.

[7] Heninger, K., Kallander, J., Parnas, D.L., Shore, J., "*Software Requirements for the A-7E Aircraft*", Naval Research Laboratory Report 3876,Nov. 1978, 523 pgs.

[8] Heninger, K.L., "*Specifying Software Requirements for Complex Systems: New Techniques and their Application*", IEEE Transactions Software Engineering, Vol. SE-6, pp. 2-13, January 1980.

- Reprinted as chapter 6 in [10]

[9] Hester, S.D., Parnas, D.L., Utter, D.F., "*Using Documentation as a Software Design Medium*", Bell System Technical Journal,60,8, pp.1941-1977, October 1981

[10] Hoffman,D.M.,Weiss, D.M. (eds.), "*Software Fundamentals: Collected Papers by David L. Parnas*", Addison-Wesley, 664 pgs., ISBN 0-201-70369-6,

[11] Janicki, R. "T*owards a Formal Semantics of Parnas Tables*", Proc. of 17th International Conference on Software Engineering, (ICSE), pp. 231-240 1995,.

[12] Janicki, R., Parnas, D.L., Zucker, J., "*Tabular Representations in Relational Documents*", in "Relational Methods in Computer Science", Chapter 12, Ed. C. Brink and G. Schmidt. Springer Verlag, pp. 184 - 196, 1997, ISBN 3-211-82971-7.

- Reprinted as Chapter 4 in item [10].

[13] Jin, Ying, Parnas, D.L., "*Defining The Meaning Of Tabular Mathematical Expressions*", Science of Computer Programming (Elsevier), Vol. 75, Issue 11, 1, Pp. 980-1000, doi:10.1016/j.scico.2009.12.009, November 2010

[14] Liu, Zhiying , Parnas, D, L, Trancón y Widemann, B., "*Documenting and Verifying Systems Assembled from Components, Frontiers of Computer Science in China*", Higher Education Press, co-published with Springer-Verlag GmbH ISSN1673-7350 (Print) 1673-7466 (Online), 2010.

[15] Mills, Harlan D.: "*The New Math of Computer Programming*". *Comm. ACM 18,*1): 43-48 (1975)

[16] Parnas, D.L., "A *Generalized Control Structure and its Formal Definition*", Comm. ACM, 26, 8, pp. 572-581, Aug. 1983

[17] Parnas, D.L.,Madey, J., "*Functional Documentation for Computer Systems Engineering*", Science of Computer Programming (Elsevier) vol. 25, no. 1, pp 41-61, Oct. 1995

[18] Parnas, D.L., Vilkomir, S.A,. "*Precise Documentation of Critical Software*", Proc. of the Tenth IEEE Symposium on High Assurance Systems Engineering (HASE), 14-16 pp. 237-244, Nov. 2007

[19] Zucker, J.I. "*Transformations of Normal and Inverted Function Tables*", Formal Aspects of Computing 8, pp. 679-705,1996.

SESSION

ERSA/WORLDCOMP TUTORIAL

Chair(s)

TBA

A Run-Time Evolvable Hardware Tutorial

Jim Torresen
Department of Informatics, University of Oslo, Oslo, Norway
E-mail: jimtoer@ifi.uio.no

Abstract – *Below follows a short intro to a tutorial about evolutionary computation applied to hardware – called evolvable hardware. This allows for run-time adaptable hardware systems which would be outlined as a part of the presentation. A short CV of the presenter is also included below.*

Keywords: Evolvable hardware, FPGA, evolutionary computation

1 Introduction

Traditional hardware design aims at creating circuits which, once fabricated, remain static during run-time. This changed with the introduction of reconfigurable technology and devices (typically FPGAs) which opened up the possibility of dynamic hardware. However, the potential of dynamic hardware for the construction of self-adaptive, self-optimizing and self-healing systems can only be realized if automatic design schemes are available.

One such method for automatic design is evolvable hardware. Evolvable hardware was introduced more than ten years ago as a new way of designing electronic circuits. Only input/output relations of the desired function need to be specified, the design process is then left to an adaptive algorithm inspired from natural evolution. The design is based on incremental improvement of a population of initially randomly generated circuits. Circuits among the best ones have the highest probability of being combined to generate new and possibly better circuits. Combination is by crossover and mutation operation of the circuit description.

In this tutorial, an introduction to evolutionary computation and how it can be applied to hardware evolution would be given. That is, an overview of how evolvable hardware can be applied to provide run time adaptivity for systems within e.g. classification applied to real-world applications will be the main content of the tutorial.

2 A Short CV for Jim Torresen

Professor Jim Torresen received his M.Sc. and Dr.ing. (Ph.D) degrees in computer architecture and design from the Norwegian University of Science and Technology, University of Trondheim in 1991 and 1996, respectively. He has been employed as a senior hardware designer at NERA Telecommunications (1996-1998) and at Navia Aviation (1998-1999). Since 1999, he has been a professor at the Department of Informatics at the University of Oslo (associate professor 1999-2005). Jim Torresen has been a visiting researcher at Kyoto University, Japan for one year (1993-1994), four months at Electrotechnical laboratory, Tsukuba, Japan (1997 and 2000) and is now a visiting professor at Cornell University.

His research interests at the moment include bio-inspired computing, machine learning, reconfigurable hardware, robotics and applying this to complex real-world applications. Several novel methods have been proposed. He has published a number of scientific papers in international journals, books and conference proceedings. 10 tutorials and several invited talks have been given at international conferences. He is in the program committee of more than ten different international conferences as well as a regular reviewer of a number of international journals. He has also acted as an evaluator for proposals and projects in EU FP7.

A list and collection of publications can be found at the following web page:
http://www.ifi.uio.no/~jimtoer/papers.html

More information on the web: http://www.ifi.uio.no/~jimtoer

SESSION

INVITED SESSION - CREATING THE SCIENCE + ENGINEERING FOR CYBER-SECURITY

Chair(s)

PROF. SHIU-KAI CHIN
PROF. WILLIAM L. HARRISON

Logic Design for Access Control, Security, Trust, and Assurance

Shiu-Kai Chin
Department of Electrical Engineering and Computer Science
Syracuse University, Syracuse, New York
Senior Scientist, Serco-NA, Inc., Rome, New York
http://www.ecs.syr.edu/faculty/chin

***Abstract*— Designs created by hardware and software engineers are often part of larger systems where confidentiality, integrity, and availability of information and other resources is a primary concern. Whether the system is a military one where assurance of mission-critical capabilities is paramount, or the system delivers financial services where assurances of integrity of financial data and transactions are crucial, the concerns are still the same. What are the access control and security policies and mechanisms that permit or deny the use of a resource or capability? Trust—who or what is believed and under what circumstances? Assurance—how do we know that what is proposed or implemented makes sense and is justifiable? This paper describes a logic and tools for access control, security, and trust. Over two hundred students from over 40 US universities have learned and applied this logic to reason about access control, security, and trust. As credible assurance requires certification and verification by independent evaluators, the logic and its applications are implemented as conservative extensions of the Higher-Order Logic (HOL) theorem prover. The HOL implementation is a means for rigorous and reusable descriptions and verifications of the access-control logic and its application.**

***Index Terms*— Security, access control, trust, formal verification, assurance**

I. Introduction

Security experts have long said security must be designed into systems from the very beginning. The principles of building trusted systems remain the same [1][2]:

- *complete mediation:* all access to objects must be checked to ensure they are allowed,
- *least privilege:* grant only the capabilities necessary to compete the specified task, and
- *economy of mechanism:* security mechanisms should be as simple as possible.

However, the methods by which design and verification engineers assess whether any given access request is allowed, falls well short of the logic-based methods and computer-aided design tools routinely used by hardware engineers.

For understandable economic reasons, most systems are designed and implemented by junior engineering staff. Secure system design within this context means that system security will remain poor unless newly-graduated engineers and engineers actually doing design and verification are able to design and reason about security in ways that are similar to the ways engineers use logic to design and verify hardware.

This paper describes an access-control logic created to help engineers design with security in mind at abstraction levels spanning hardware to concepts of operations and policies. We have taught these methods to over to 226 junior and senior ROTC cadets from over forty US universities with the Air Force Research Laboratory Information Directorate. From 2003–2010, we have taught what amounts to logic design for access control in the Air Force Research Laboratory's Advanced Course in Engineering (ACE) Cyber Security Boot Camp [3], [4]. We have also taught these methods to over 25 active-duty lieutenants, captains, and civilian contractors. We routinely teach these methods at Syracuse University to our undergraduate and graduate students.

Computer-aided design tools are necessary for independently verifying and reusing hardware designs. Systems with security concerns are no different in their need for independent verification and reuse. To help meet these needs we have implemented the access-control logic as a conservative extension to the Higher Order Logic (HOL) theorem prover [5].

The rest of this paper is organized as follows. In Section II we review the role logic plays in hardware design and draw some analogies for security. The syntax, semantics, and inference rules of the access-control logic are described in Section III. Section IV provides some illustrative examples applying the access-control logic to hardware and policies. We give an overview of the implementation of the logic as a conservative extension to the HOL theorem prover in Section V. Section VI shows how the examples in Section IV are verified in HOL. We close with related work and conclusions in Sections VII and VIII.

II. Logic Design for Hardware and Logic Design for Security

It is nearly impossible to separate hardware design from logic design. Propositional logic describes the behavior of components; it informs the design process; it is used to verify implementations satisfy their specifications. Propositional logic makes hardware design and verification rigorous.

Hardware engineers derive or prove what their designs do using propositional logic. During preliminary and critical design reviews, if a designer is given: (1) all the primary input values, and (2) the values stored in memory, then he or she is expected to *derive* whether the value of any particular gate or

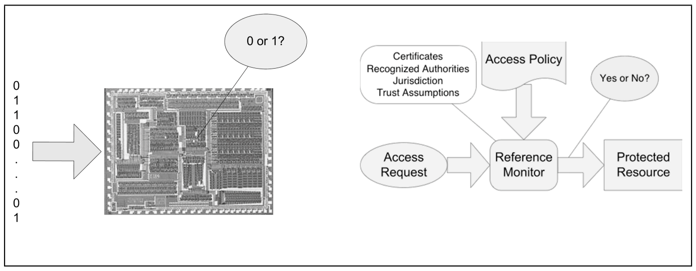


Fig. 1: Expectation of engineers: rigorous derivation of behavior

flip-flop output is a 1 or a 0. Inability to do so is regarded as incompetence.

Figure 1 illustrates our view that similar expectations should hold for engineers designing systems with security requirements. When given: (1) a request to access a protected resource, (2) statements of policies, recognized authorities, jurisdiction of authority, certified statements, and (3) trust assumptions, engineers should be able to *derive mathematically* if the request should be honored or not. Anything less should be regarded as incompetence.

Educating and equipping engineers with the logical concepts and tools to meet the expectations illustrated in Figure 1 is a critical requirement. To meet this requirement, there are two lessons the security community can learn from the hardware community.

A. Keep the logic simple so it is widely used

Logic design for hardware still relies on propositional logic and finite-state machines. First courses in digital design rely on both as a means of describing concepts, component behavior, specifications, implementations, and design procedures. Propositional logic has no notion of time. Yet, it is still the logical system of choice of engineers because of its simplicity. Every hardware engineer can explain design concepts and implementations using propositional logic.

While temporal logics, e.g., computation tree logic (CTL) [6] are used for model checking and design verification, these richer and more complex systems are harder to learn and require more time and effort to master. As a profession, hardware engineers have decided pragmatically that the simpler propositional logic gives superior descriptive and analytical value for the investment in time and effort when compared to more complicated temporal logics. For forty years, computer hardware design courses have taught introductory hardware design in much the same way—relying on propositional logic as the means for explaining concepts and designs.

Propositional logic is equivalent to the architect's ruler and the contractor's tape measure. While it is simple and does not solve every problem, it is essential, used constantly, and no one can conceive of doing their job without it.

Keeping this lesson on simplicity in mind, our goal to help engineers design for security requires us to keep the access-control logic as simple as possible so it can be mastered by undergraduates and widely used in many contexts.

B. Remember how VLSI design became mainstream

In the 1970s, VLSI (very large scale integrated) circuit design was thought to be too exotic and complex to be taught at the undergraduate level. Engineers of the time routinely spoke of the need for "tall thin men," i.e., engineers who could translate algorithms into working silicon by relating algorithmic specifications to register-transfer level designs implemented as custom VLSI circuits. People of the time lamented the lack of VLSI designers in professional practice. Comments such as, "there are fewer VLSI designers than there are NFL (National Football League) players" were common.

The situation for security and trustworthiness is not unlike the state of VLSI in the 1970s.

1) Few engineers can do secure system design.
2) US interests depend on system security.
3) The US government is attempting to address its needs by offering scholarships to students to study security.

What made VLSI circuit design feasible to teach at the undergraduate level in the 1980s?

1) Carver Mead's landmark textbook *Introduction to VLSI Systems* [7], which made VLSI accessible to electrical and computer engineering faculty and practitioners,
2) free computer-aided design (CAD) tools such as Magic—a circuit layout tool with a design-rule checker [8]—and simulators like SPICE (Simulation Program with Integrated Circuit Emphasis) [9], which made electrically correct integrated circuit design possible,
3) MOSIS (Metal Oxide Semiconductor Implementation Service) [10], an inexpensive (free) semiconductor fabrication service subsidized by the US government available to US universities that enabled students and faculty to make VLSI circuits, and
4) government-sponsored programs to produce a critical mass of faculty to teach VLSI to undergraduates.

Today, VLSI is part of the undergraduate curriculum. We can learn from the example of VLSI to move security and trustworthiness into mainstream engineering.

III. Syntax, Semantics, and Inference Rules

This section defines the access-control logic. It requires only a sophomore level understanding of discrete mathematics. However, this is enough for a full accounting of the logic and its use across a wide spectrum of applications.

A. Syntax and Semantics

Syntax of Principal Expressions: Principals are the actors in a system, such as people, processes, cryptographic keys, personal identification numbers (PINs), userid–password pairs, and so on. Principals are either *simple* or *compound*. **PName** is the collection of all simple principal names, which can be used to refer to any simple principal. For example, the following are all allowable principal names: *Alice*, *Bob*, the key K_{Alice}.

Compound principals are abstract entities that connote a combination of principals: for example, "the President in conjunction with Congress" connotes an abstract principal comprising both the President and Congress. Intuitively, such a principal makes exactly those statements that are made by *both* the President and Congress. Similarly, “the reporter quoting her source” connotes an abstract principal that comprises both the reporter and her source. Intuitively, a statement made by such a principal represents a statement that the reporter is (rightly or wrongly) attributing to his source.

The set **Princ** of *all* principal expressions is given by the following BNF specification:

$$\textbf{Princ} \quad ::= \quad \textbf{PName} \;/\; \textbf{Princ} \;\&\; \textbf{Princ} \;/\; \textbf{Princ} \mid \textbf{Princ}$$

That is, a principal expression is either a simple name, an expression of form $P \,\&\, Q$ (where P and Q are both principal expressions), or an expression of form $P \mid Q$ (where, again, P and Q are both principal expressions).

Syntax of Logical Formulas: The abstract syntax of logical formulas **Form** are constructed from the set of *principal names* and a countable set of *propositional variables* **PropVar**:

$$\begin{array}{lll}\textbf{Form} & ::= & \textbf{PropVar} \;/\; \neg\ \textbf{Form} \;/ \\ & & (\textbf{Form} \vee \textbf{Form}) \;/\; (\textbf{Form} \wedge \textbf{Form}) \;/ \\ & & (\textbf{Form} \supset \textbf{Form}) \;/\; (\textbf{Form} \equiv \textbf{Form}) \;/ \\ & & (\textbf{Princ} \Rightarrow \textbf{Princ}) \;/\; (\textbf{Princ}\ \textsf{says}\ \textbf{Form}) \;/ \\ & & (\textbf{Princ}\ \textsf{controls}\ \textbf{Form}) \;/\; \textbf{Princ}\ \textsf{reps}\ \textbf{Princ}\ \textsf{on}\ \textbf{Form}\end{array}$$

The first six cases deal with standard propositional logic: propositional variables, negation, conjunction, disjunction, implication, and equivalence. The remaining four cases are specific to access control.

1) P says φ asserts that principal P made the statement φ.
2) $P \Rightarrow Q$ (pronounced “P speaks for Q”) indicates that every statement made by P can also be viewed as a statement from Q.
3) P controls φ represents authority or trust. It is an abbreviation for the implication $(P \textsf{ says } \varphi) \supset \varphi$. P is a trusted authority with respect to the statement φ.
4) P reps Q on φ represents delegation. It is an abbreviation for $(P \mid Q \textsf{ says } \varphi) \supset (Q \textsf{ says } \varphi)$. P is a trusted authority on what Q says regarding φ.

Semantics: The semantics of formulas is given via Kripke structures, as follows.

Definition: A **Kripke structure** $\mathcal{M}$ is a three-tuple $\langle W, I, J\rangle$, where:

- W is a nonempty set, whose elements are called *worlds*.
- I : **PropVar** $\rightarrow \mathcal{P}(W)$ is an *interpretation* function that maps each propositional variable p to a set of worlds.
- J : **PName** $\rightarrow \mathcal{P}(W \times W)$ is a function that maps each principal name A into a relation on worlds (i.e., a subset of $W \times W$).

Given the above, we define the extended function $\hat{J}$: **Princ** $\rightarrow \mathcal{P}(W \times W)$ inductively on the structure of principal expressions, where $A \in$ **PName**.

$$\begin{array}{rcl}\hat{J}(A) & = & J(A) \\ \hat{J}(P \,\&\, Q) & = & \hat{J}(P) \cup \hat{J}(Q) \\ \hat{J}(P \mid Q) & = & \hat{J}(P) \circ \hat{J}(Q).\end{array}$$

Note: $R_1 \circ R_2 = \{(x,z) \mid \exists y.(x,y) \in R_1 \text{ and } (y,z) \in R_2\}$. $\diamond$

Definition: Each Kripke structure $\mathcal{M} = \langle W, I, J\rangle$ gives rise to a **semantic function**

$$\mathcal{E}_{\mathcal{M}}[\![-]\!] : \textbf{Form} \rightarrow \mathcal{P}(W),$$

where $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!]$ is the set of worlds in which φ is considered true. $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!]$ is defined inductively on the structure of φ, as follows:

$$\begin{array}{rcl}\mathcal{E}_{\mathcal{M}}[\![p]\!] & = & I(p) \\ \mathcal{E}_{\mathcal{M}}[\![\neg\varphi]\!] & = & W - \mathcal{E}_{\mathcal{M}}[\![\varphi]\!] \\ \mathcal{E}_{\mathcal{M}}[\![\varphi_1 \wedge \varphi_2]\!] & = & \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!] \\ \mathcal{E}_{\mathcal{M}}[\![\varphi_1 \vee \varphi_2]\!] & = & \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!] \cup \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!] \\ \mathcal{E}_{\mathcal{M}}[\![\varphi_1 \supset \varphi_2]\!] & = & (W - \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!]) \cup \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!] \\ \mathcal{E}_{\mathcal{M}}[\![\varphi_1 \equiv \varphi_2]\!] & = & \mathcal{E}_{\mathcal{M}}[\![\varphi_1 \supset \varphi_2]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\varphi_2 \supset \varphi_1]\!] \\ \mathcal{E}_{\mathcal{M}}[\![P \Rightarrow Q]\!] & = & \begin{cases} W, & \text{if } \hat{J}(Q) \subseteq \hat{J}(P) \\ \emptyset, & \text{otherwise} \end{cases} \\ \mathcal{E}_{\mathcal{M}}[\![P \textsf{ says } \varphi]\!] & = & \{w \mid \hat{J}(P)(w) \subseteq \mathcal{E}_{\mathcal{M}}[\![\varphi]\!]\} \\ \mathcal{E}_{\mathcal{M}}[\![P \textsf{ controls } \varphi]\!] & = & \mathcal{E}_{\mathcal{M}}[\![(P \textsf{ says } \varphi) \supset \varphi]\!] \\ \mathcal{E}_{\mathcal{M}}[\![P \textsf{ reps } Q \textsf{ on } \varphi]\!] & = & \mathcal{E}_{\mathcal{M}}[\![(P \mid Q \textsf{ says } \varphi) \supset Q \textsf{ says } \varphi]\!]\end{array}$$

Note that, in the definition of $\mathcal{E}_{\mathcal{M}}[\![P \textsf{ says } \varphi]\!]$, $\hat{J}(P)(w)$ is simply the image of world w under the relation $\hat{J}(P)$. $\diamond$

The semantic functions $\mathcal{E}_{\mathcal{M}}$ provide a fully defined and fully disclosed interpretation for the formulas of the logic. These functions precisely define the meaning of statements and what is described in the logic.

Definition: We say a Kripke structure $\mathcal{M}$ **satisfies** a formula φ when $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!] = W$, i.e., φ is true in all worlds W of $\mathcal{M}$. We denote $\mathcal{M}$ satisfies φ by $\mathcal{M} \models \varphi$. $\diamond$

In practice, reasoning at the level of Kripke structures is cumbersome. Instead, we use logical rules to reason about access control.

B. Logical Rules

Logical rules in our access-control logic have the form

$$\frac{H_1 \quad \cdots \quad H_k}{C,}$$

where $H_1 \cdots H_k$ and C are formulas in the logic. $H_1 \cdots H_k$ are the *hypotheses* or *premises* and C is the *consequence* or *conclusion*. Informally, we read logical rules as “if all the hypotheses above the line are true, then the conclusion below the line is also true.” If there are no hypotheses, then the logical rule is an *axiom*.

Logical rules are used to manipulate well-formed formulas of the logic. If all the hypotheses of a rule are written down (derived) then the conclusion of the rule also can be written down (derived). All logical rules must maintain logical consistency. If all logical rules are *sound*, as defined below, then logical consistency is assured.

Definition: A logical rule

$$\frac{H_1 \quad \cdots \quad H_k}{C,}$$

$$\textit{Taut}\quad \frac{}{\varphi}\ \text{if } \varphi \text{ is an instance of a prop-logic tautology} \qquad \textit{Modus Ponens}\quad \frac{\varphi \quad \varphi \supset \varphi'}{\varphi'} \qquad \textit{Says}\quad \frac{\varphi}{P \text{ says } \varphi}$$

$$\textit{MP Says}\quad \frac{}{(P \text{ says } (\varphi \supset \varphi')) \supset (P \text{ says } \varphi \supset P \text{ says } \varphi')} \qquad \textit{Speaks For}\quad \frac{}{P \Rightarrow Q \supset (P \text{ says } \varphi \supset Q \text{ says } \varphi)}$$

$$\textit{\& Says}\quad \frac{}{(P \,\&\, Q \text{ says } \varphi) \equiv ((P \text{ says } \varphi) \wedge (Q \text{ says } \varphi))} \qquad \textit{Quoting}\quad \frac{}{(P \mid Q \text{ says } \varphi) \equiv (P \text{ says } Q \text{ says } \varphi)}$$

$$\textit{Idempotency of} \Rightarrow \quad \frac{}{P \Rightarrow P} \qquad \textit{Transitivity of} \Rightarrow \quad \frac{P \Rightarrow Q \quad Q \Rightarrow R}{P \Rightarrow R} \qquad \textit{Monotonicity of} \Rightarrow \quad \frac{P \Rightarrow P' \quad Q \Rightarrow Q'}{P \mid Q \Rightarrow P' \mid Q'}$$

$$\textit{Equivalence}\quad \frac{\varphi_1 \equiv \varphi_2 \quad \psi[\varphi_1/q]}{\psi[\varphi_2/q]} \qquad P \text{ controls } \varphi \overset{\text{def}}{=} (P \text{ says } \varphi) \supset \varphi \qquad P \text{ reps } Q \text{ on } \varphi \overset{\text{def}}{=} (P \mid Q \text{ says } \varphi) \supset Q \text{ says } \varphi$$

Fig. 2: Core logical rules for the access-control logic

$$\textit{Controls}\quad \frac{P \text{ controls } \varphi \quad P \text{ says } \varphi}{\varphi}$$

$$\textit{Derived Speaks For}\quad \frac{P \Rightarrow Q \quad P \text{ says } \varphi}{Q \text{ says } \varphi} \qquad \textit{Derived Controls}\quad \frac{P \Rightarrow Q \quad Q \text{ controls } \varphi}{P \text{ controls } \varphi}$$

Fig. 3: Some Useful Derived rules

$$l_1 =_s l_2 \overset{\text{def}}{=} (l_1 \le_s l_2) \wedge (l_2 \le_s l_1)$$

$$\textit{Reflexivity of} \le_s \quad \frac{}{l \le_s l} \qquad \textit{Transitivity of} \le_s \quad \frac{l_1 \le_s l_2 \quad l_2 \le_s l_2}{l_1 \le_s l_3}$$

$$\le_s \textit{ Subst}\quad \frac{\mathsf{sl}(P) =_s l_1 \quad \mathsf{sl}(Q) =_s l_2 \quad l_1 \le_s l_2}{\mathsf{sl}(P) \le_s \mathsf{sl}(Q)}$$

Fig. 4: Rules on Security Levels

is **sound** if, for all Kripke structures $\mathcal{M}$, whenever $\mathcal{M}$ satisfies all the hypotheses $H_1 \cdots H_k$, then $\mathcal{M}$ also satisfies C, i.e., if for all $\mathcal{M}$: $\mathcal{M} \models H_i$ for $1 \le i \le k$, then it must be the case that $\mathcal{M} \models C$. $\diamond$

Figure 2 shows the core logical rules for the logic. All the rules are proved sound with respect to the Kripke semantics. Figure 3 shows three useful derived rules, which are proved using the core rules.

C. Security and Integrity Levels

Confidentiality and integrity policies such as Bell-LaPadula [11] and Biba's Strict Integrity policy [12], depend on classifying (i.e., assigning confidentiality or integrity levels to) information, subjects, and objects. It is straightforward to extend the access-control logic to include confidentiality, integrity, or availability levels as needed. In what follows, we show how the syntax and semantics of *confidentiality* levels are added to the core access-control logic. The same process is used for other classification schemes, e.g., integrity and availability.

a) Syntax: The first step is to introduce syntax for describing and comparing security levels. **SecLabel** is the collection of *simple security labels*, which are used as names for the confidentiality levels (e.g., HI and LO).

Often, we refer abstractly to a principal P's integrity level. We define the larger set **SecLevel** of *all* possible security-level expressions:

$$\textbf{SecLevel} \ ::= \ \textbf{SecLabel} \ / \ \mathsf{sl}(\textbf{PName}).$$

A security-level expression is either a simple security label or an expression of the form $\mathsf{sl}(A)$, where A is a simple principal name. Informally, $\mathsf{sl}(A)$ refers to the security level of principal A.

Finally, we extend our definition of well-formed formulas to support comparisons of security levels:

$$\textbf{Form} \ ::= \ \textbf{SecLevel} \le_s \textbf{SecLevel} \ / \ \textbf{SecLevel} =_s \textbf{SecLevel}$$

Informally, a formula such as LO $\le_s$ $\mathsf{sl}(\textit{Kate})$ states that Kate's security level is greater than or equal to, i.e., *dominates* the security level LO. Similarly, a formula such as $\mathsf{sl}(\textit{Barry}) =_s \mathsf{sl}(\textit{Joe})$ states that Barry and Joe have the same security level.

b) Semantics: Providing formal and precise meanings for the newly added syntax requires us to first extend our Kripke structures with additional components that describe security classification levels. Specifically, we introduce extended Kripke structures of the form

$$\mathcal{M} = \langle W, I, J, K, L, \preceq \rangle,$$

where:

- W, I, and J are as defined earlier.
- K is a non-empty set, which serves as the universe of *security levels*.
- $L : (\textbf{SecLabel} \cup \textbf{PName}) \to K$ is a function that maps each security label and each simple principal name to a security level. L is extended to work over arbitrary security-level expressions, as follows:

  $$L(\ \mathsf{sl}(A)) = L(A),$$

  for every simple principal name A.
- $\preceq \subseteq K \times K$ is a **partial order** on K: that is, $\preceq$ is *reflexive* (for all $k \in K$, $k \preceq k$), *transitive* (for all $k_1, k_2, k_3 \in$

K, if $k_1 \preceq k_2$ and $k_2 \preceq k_3$, then $k_1 \preceq k_3$), and *anti-symmetric* (for all $k_1, k_2 \in K$, if $k_1 \preceq k_2$ and $k_2 \preceq k_1$, then $k_1 = k_2$).

Using these extended Kripke structures, we extend the semantics for our new well-formed expressions as follows:

$$\begin{aligned}\mathcal{E}_{\mathcal{M}}[\![\ell_1 \leq_s \ell_2]\!] &= \begin{cases} W, & \text{if } L(\ell_1) \preceq L(\ell_2) \\ \emptyset, & \text{otherwise}\end{cases} \\ \mathcal{E}_{\mathcal{M}}[\![\ell_1 =_s \ell_2]\!] &= \mathcal{E}_{\mathcal{M}}[\![\ell_1 \leq_s \ell_2]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\ell_2 \leq_s \ell_1]\!].\end{aligned}$$

As these definitions suggest, the expression $\ell_1 =_s \ell_2$ is simply an abbreviation for $(\ell_1 \leq_s \ell_2) \wedge (\ell_2 \leq_s \ell_1)$.

Logical Rules: Based on the extended Kripke semantics we introduce logical rules that support the use of integrity levels to reason about access requests. Specifically, the definition, reflexivity, and transitivity rules in Figure 4 reflect that $\leq_s$ is a partial order. The fourth rule is derived and useful to have.

IV. Examples

A. Physical Memory Security and Virtualization

Figure 5 is a block diagram of a simple virtual machine (VM) with an instruction register IR and an accumulator ACC. Instructions from VM are monitored by the virtual machine monitor (VMM). VMM has a *memory access register* (MAR) pointing to a real address in physical memory and a *relocation register* (RR) specifying the *base* address of the memory segment being accessed in physical memory and the segment's *bound* (number of memory locations). The physical memory shown has q memory locations (0 through $q-1$). The memory has three segments, corresponding to Alice, Bob, and Carol. Alice's segment starts at physical address $base_{Alice}$ and ends at $base_{Alice} + bound_{Alice} - 1$.

Values in registers are described in the logic as statements made by registers. For example, suppose the value loaded in IR is the instruction LDA @A (i.e., load ACC with the value in virtual address A in physical memory). From an access-control perspective, the question is whether this instruction should be executed or *trapped* by VMM, which mediates all access requests to physical memory.

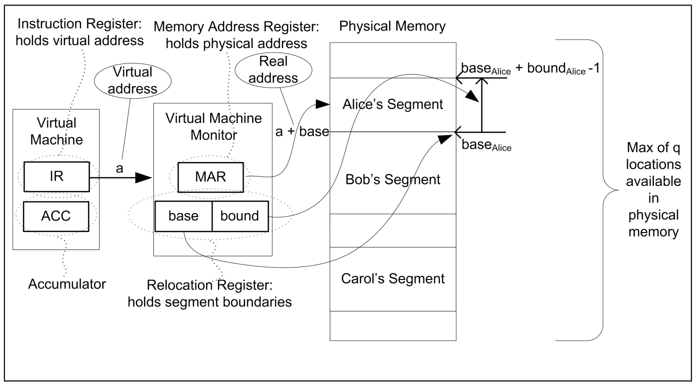
VMM grants access if (1) the virtual address A is less than or equal to *bound* and, (2) $base + \text{A} < q$ (i.e., the physical address corresponding to A is within the physical memory address limit). Otherwise, the instruction is *trapped* by VMM and control is turned over to the supervisor.

The decision to grant or deny (trap) request LDA @A depends on the *state* of VM and VMM and the *mandatory access control* (MAC) policy governing access to physical memory. The state of the machine (VM and VMM) is given by the register values. The registers used to decide access are IR and RR; ACC and MAR are not used. The states of IR and RR are given by:

$$IR \textsf{ says } \langle \text{LDA @A} \rangle \quad \text{and} \quad RR \textsf{ says } \langle (base_{Alice}, bound_{Alice}) \rangle,$$

where $\langle \text{LDA @A} \rangle$ and $\langle (base_{Alice}, bound_{Alice}) \rangle$ denote "it is a good idea to execute LDA @A" and "the value is in fact $(base_{Alice}, bound_{Alice})$."

The request to execute LDA @A is *trapped* when either the real address $base_{Alice} + \text{A}$ is greater or equal to q (i.e., exceeds



Instruction Register: holds virtual address
Memory Address Register: holds physical address
Physical Memory
Virtual address
Real address
$base_{Alice}$ + $bound_{Alice}$ -1
Virtual Machine
IR
ACC
a
Virtual Machine Monitor
MAR
base
bound
a + base
Alice's Segment
$base_{Alice}$
Bob's Segment
Carol's Segment
Max of q locations available in physical memory
Accumulator
Relocation Register: holds segment boundaries

Fig. 5: Virtual Machine and Virtual Machine Monitor

the size of physical memory), or the virtual address A exceeds the segment bound $bound_{Alice}$. This policy is expressed as follows:

$$IR \textsf{ says } \langle \text{LDA @A} \rangle \supset (RR \textsf{ says } \langle (base_{Alice}, bound_{Alice}) \rangle \supset \\ (((base_{Alice} + \text{A} \geq q) \vee (\text{A} \geq bound_{Alice})) \supset \langle trap \rangle)),$$

where $\langle trap \rangle$ denotes "it is a good idea to trap."

The request to execute LDA @A is granted, i.e., IR has discretionary access, if virtual address A falls within the bounds of real memory and the segment. This is expressed as follows:

$$(RR \textsf{ says } \langle (base_{Alice}, bound_{Alice}) \rangle \supset ((base_{Alice} + \text{A} < q) \supset \\ ((\text{A} < bound_{Alice}) \supset IR \textsf{ controls } \langle \text{LDA @A} \rangle))).$$

Figure 6 gives a simple proof justifying granting execution for the instruction LDA @A (loading the accumulator with the contents of address A in the active segment), where the base address of the segment is *base*, the segment bound is *bound*, and the size of physical memory is *q*. Lines 1–5 above the line are the starting assumptions: the instruction request, the state of the relocation register RR, and the mandatory access control policy for the specific memory segment, physical memory, the physical address is within memory, and the virtual address is within the segment. The remaining lines are *derived* by applying the *Modus Ponens* inference rule in Figure 2. The proof justifies the following *derived* inference rule:

$$\frac{\begin{array}{c} IR \textsf{ says } \langle \text{LDA @A} \rangle \qquad RR \textsf{ says } \langle (base, bound) \rangle \\ RR \textsf{ says } \langle (base, bound) \rangle \supset (base + A < q) \supset \\ (A < bound) \supset IR \textsf{ controls } \langle \text{LDA @A} \rangle \\ base + A < q \qquad A < bound \end{array}}{\langle \text{LDA @A} \rangle.}$$

B. Confidentiality

We now give an example of the Bell-LaPadula multi-level security policy [11] governing read and write access to files. The policy has two conditions:

1) **Simple security condition:** A principal P can read object O if and only if P's security level dominates (at least as high as) O's and P has discretionary read access to O. The policy implemented by reference monitors is

$$(\,\textsf{sl}(O) \leq_s \textsf{sl}(P)) \supset (P \textsf{ controls } \langle read, O \rangle).$$

1.	IR says $\langle \textsc{lda}\ @\text{A}\rangle$	Access request
2.	RR says $\langle (base, bound)\rangle$	Segment location
3.	RR says $\langle (base, bound)\rangle \supset (base + A < q) \supset (A < bound) \supset IR$ controls $\langle \textsc{lda}\ @\text{A}\rangle$	Access policy
4.	$base + A < q$	Physical address OK
5.	$A < bound$	Virtual address OK
6.	$(base + A < q) \supset (A < bound) \supset IR$ controls $\langle \textsc{lda}\ @\text{A}\rangle$	2, 3 Modus Ponens
7.	$(A < bound) \supset IR$ controls $\langle \textsc{lda}\ @\text{A}\rangle$	4, 6 Modus Ponens
8.	IR controls $\langle \textsc{lda}\ @\text{A}\rangle$	5, 7 Modus Ponens
9.	$\langle \textsc{lda}\ @\text{A}\rangle$	1, 8 Controls

Fig. 6: LDA @A Access Proof

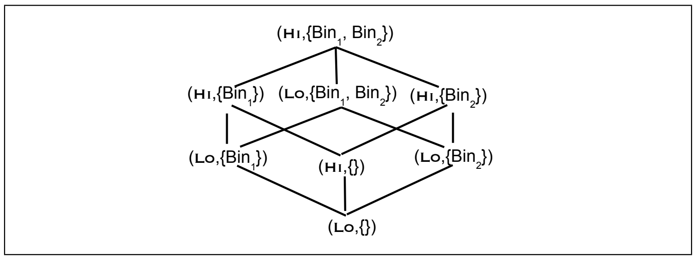


Fig. 7: Partially Ordered Security Levels

2) ***-property:** A principal P can write to object O if and only if O's security level dominates P's and P has discretionary write access to O. The policy implemented by reference monitors is

$$(\ \mathsf{sl}(P) \leq_s\ \mathsf{sl}(O)) \supset (P \text{ controls } \langle write, O\rangle).$$

The converse of the above, i.e., $(P \text{ controls } \langle read, O\rangle) \supset (\ \mathsf{sl}(O) \leq_s\ \mathsf{sl}(P))$ and $(P \text{ controls } \langle write, O\rangle) \supset (\ \mathsf{sl}(P) \leq_s\ \mathsf{sl}(O))$ correspond to *verification conditions* that must be proved about the implementation. Specifically, we must verify there is no way to bypass the reference monitor to access O.

As a concrete example, consider security levels that are *pairs* (L, C), where $L \in \{\textsc{hi}, \textsc{lo}\}$ and $C \subseteq \{Bin_1, Bin_2\}$. L is a level and C is a (possibly empty) set of compartments. The partial ordering $\leq$ on L is

$$\leq \stackrel{\text{def}}{=} \{(\textsc{hi},\textsc{hi}), (\textsc{lo},\textsc{hi}), (\textsc{lo},\textsc{lo})\}.$$

We define a partial ordering on (L, C) as follows:

$$(L_2, C_2)\ dom\ (L_1, C_1) \text{ if and only if } L_1 \leq L_2 \text{ and } C_1 \subseteq C_2.$$

Figure 7 is a Hasse diagram (arcs corresponding to reflexivity and transitivity are omitted) of the partial ordering.

Suppose Carol's security level is $(\textsc{hi}, \{Bin_1, Bin_2\})$ and she has discretionary read access on O_3, whose security level is $(\textsc{lo}, \{Bin_1\})$. The following derived inference rule states that given the above security level assignments, Carol's request to read O_3 would be granted under the simple security condition of Bell-LaPadula. Figure 8 is the proof.

1.	$Carol$ says $\langle read, O_3\rangle$	Access request
2.	$\mathsf{sl}(O_2) =_s (\textsc{lo}, \{Bin_1\})$	O_3's security level
3.	$\mathsf{sl}(Carol) =_s (\textsc{hi}, \{Bin_1, Bin_2\})$	Carol's security level
4.	$\mathsf{sl}(Carol)\ dom\ \mathsf{sl}(O_3) \supset (Carol$ controls $\langle read, O_3\rangle)$	Simple security condition
5.	$(\textsc{hi}, \{Bin_1, Bin_2\})\ dom\ (\textsc{lo}, \{Bin_1\})$	Ordering
6.	$\mathsf{sl}(Carol)\ dom\ \mathsf{sl}(O_3)$	2, 3 $\leq_s$ Subst
7.	$Carol$ controls $\langle read, O_3\rangle$	6, 4 Modus Ponens
8.	$\langle read, O_3\rangle$	7, 1 Controls

Fig. 8: Proof of Derived Read Access Rule

$$\frac{\begin{array}{c} Carol \text{ says } \langle read, O_3\rangle \quad \mathsf{sl}(O_2) =_s (\textsc{lo}, \{Bin_1\}) \\ \mathsf{sl}(Carol) =_s (\textsc{hi}, \{Bin_1, Bin_2\}) \\ \mathsf{sl}(Carol)\ dom\ \mathsf{sl}(O_3) \supset (Carol \text{ controls } \langle read, O_3\rangle) \\ (\textsc{hi}, \{Bin_1, Bin_2\})\ dom\ (\textsc{lo}, \{Bin_1\}) \end{array}}{\langle read, O_3\rangle}$$

V. Implementation in HOL

We now focus on the computer-assisted reasoning tools that provide the capability to reuse and verify designs and theories with assurance.

A. Introduction to the HOL Theorem Prover

The HOL (higher-order logic) system is a collection of functional programs written in the functional programming language ML (meta-language) and executed by ML interpreters such as Moscow ML and PolyML. While it is infeasible to give a complete description of how to use HOL and implement the access-control logic within HOL, we give enough detail to give a qualitative understanding of what HOL does, how the logic is implemented as a conservative extension of the logic, and some examples.

HOL is used in two modes: compiled or interactive. HOL is used in a batch mode to compile pre-existing and previously verified theories efficiently and quickly. Users do not interact with HOL in this mode. Typically, all that is done is to execute `Holmake` on a command line in the appropriate subdirectory.

HOL is used interactively to explore existing theories and to build new theories. In interactive mode users work with the HOL interpreter using ML. An example appears below.

```
- BOOL_CASES_AX;
> val it = |- !t. (t <=> T) \/ (t <=> F) : thm
```

The user prompt is "-". The user input is "`BOOL_CASES_-AX`" terminated by "`;`". HOL's response is after ">".

`BOOL_CASES_AX` is the name of a theorem in HOL. The value associated with `BOOL_CASES_AX` is a *theorem*, i.e., a value of type "`thm`"). The theorem itself, using Table I (taken from [5]) to translate HOL notation into standard notation, is

$$\vdash \forall t.(t \iff T) \lor (t \iff F).$$

We recognize this theorem as stating that for all Boolean terms t, t is either true or false.

Theorems in HOL are *sequents*, i.e., expressions of the form $\Gamma \vdash \Sigma$. A theorem consists of a (possibly empty) set of

Terms of the HOL Logic

Kind of term	*HOL notation*	*Standard notation*	*Description*
Truth	`T`	$\top$	*true*
Falsity	`F`	$\bot$	*false*
Negation	`~`t	$\neg t$	*not* t
Disjunction	t_1`\/`t_2	$t_1 \vee t_2$	t_1 *or* t_2
Conjunction	t_1`/\`t_2	$t_1 \wedge t_2$	t_1 *and* t_2
Implication	t_1`==>`t_2	$t_1 \supset t_2$	t_1 *implies* t_2
Equality	t_1`=`t_2	$t_1 = t_2$	t_1 *equals* t_2
$\forall$-quantification	`!`x`.`t	$\forall x.\ t$	*for all* $x : t$
$\exists$-quantification	`?`x`.`t	$\exists x.\ \ t$	*for some* $x : t$
ε-term	`@`x`.`t	$\varepsilon x.\ t$	*an* x *such that:* t
Conditional	`if` t `then` t_1 `else` t_2	$(t \to t_1, t_2)$	*if* t *then* t_1 *else* t_2

TABLE I
NOTATION IN HOL PROOF CHECKER

$$\textit{ASSUME } t \quad \frac{}{t \vdash t} \qquad \textit{MP} \quad \frac{A_1 \vdash t_1 \supset t_2 \quad A_2 \vdash t_1}{A_1 \cup A_2 \vdash t_2}$$

$$\textit{DISCH\_ALL} \quad \frac{\{H_1, \cdots, H_n\} \vdash t}{\vdash H_1 \supset \cdots \supset H_n \supset t}$$

Fig. 9: Example HOL Inference Rules

assumptions Γ separated by the turnstile symbol $\vdash$ from the conclusion Σ. A theorem asserts that whenever all the terms in set Γ are true, then the conclusion Σ is guaranteed to be true as well.

Inference rules in HOL are functions that return values of type `thm` when applied to their arguments. Figure 9 describes three HOL inference rules, *`ASSUME t`*, *`MP`*, and *`DISCH_ALL`*. The rule *`ASSUME t`* says that if you assume Boolean term t, then you may conclude it as well. *`MP`* corresponds to Modus Ponens: *if p then q, p, therefore q*. The rule *`DISCH_ALL`* moves all of the assumptions of a theorem into an implication ending in the conclusion.

The example shown below in HOL shows the application of `ASSUME` twice followed by `MP` to conclude q given $p \supset q$ and p. Note that terms in the HOL logic (as opposed to ML values and ML functions) are within backwards quotes ` ``-`` `. ML functions and values are meta-logical; terms within backwards quotes are HOL objects.

```
- val th1 = ASSUME ``p ==> q``;
> val th1 =  [.] |- p ==> q : thm
- val th2 = ASSUME ``p:bool``;
> val th2 =  [.] |- p : thm
- val th3 = MP th1 th2;
> val th3 =  [..] |- q : thm
- DISCH_ALL th3;
> val it = |- (p ==> q) ==> p ==> q : thm
```

The inference rules in Figure 9 are low-level forward inference rules. They are known as such because they take relatively small proof steps. HOL also has high-level decision procedures that take large steps. For example, the function `PROVE` takes a list of theorems and attempts to prove the term to which it is applied. In the example below, `PROVE` is used to show that $x + 0 + y + z = y + (z + x)$ given a list of theorems on addition (associativity, symmetry, and identity).

```
- ADD_ASSOC;
> val it =
    |- !m n p. m + (n + p) = m + n + p : thm
- ADD_SYM;
> val it = |- !m n. m + n = n + m : thm
- ADD_CLAUSES;
> val it =
     |- (0 + m = m) /\ (m + 0 = m) /\
        (SUC m + n = SUC (m + n)) /\
        (m + SUC n = SUC (m + n)) : thm
- PROVE
   [ADD_ASSOC, ADD_SYM, ADD_CLAUSES]
   ``x+0+y+z = y+(z+x)``;
> val it = |- x + 0 + y + z = y + (z + x) : thm
```

HOL is a rich system with many capabilities, particularly in the areas of defining new recursive types and structural induction. There are numerous theories ranging from mathematics (e.g., sets, relations, and real numbers) to applications such as the operational semantics of popular microprocessors such as the x-86 and ARM.

With the above as introduction, what follows is an overview of our implementation of the access-control logic in HOL. We first define the syntax of the logic followed by its semantics and inference rules.

B. Syntax

The syntactic definitions of principals (**Princ**), integrity levels, security levels, and formulas (**Form**) as shown in Section III-A is mirrored in HOL. The definitions (what is supplied to HOL as inputs) are shown in Figure 10.

HOL supports *polymorphism*, which allows the same definition to be used over many different types. For example, in Figure 10(a) the type `Princ` of principals is recursively defined in terms of three cases: (1) `Name` applied to a term of type `'apn`, (2) `meet` applied to two terms of type `Princ`, and (3) `quoting` applied to two terms of type `Princ`. The HOL type `'apn` is a *type variable*, i.e., `'apn` can be instantiated with specific types such as numbers, strings, sets of Booleans, etc. (Note, in HOL all type variable names start with `'`).

For example, the HOL term ` ``Name (Alice:string)`` ` has type `string Princ`, whereas

``Name (Alice:num)`` has type num Princ and ``Name (Alice:'pName)`` has type 'pName Princ. Compound principals arise from the use of meet (called *with* in the logic) and quoting. Of course, the types of the terms to which meet and quoting are applied must match, e.g., ``(Name (Bob:'pName)) quoting (Name (Alice:'pName))``.

Figure 10(a) also describes the syntax of integrity and security levels IntLevel and SecLevel, respectively. Note, that in both cases the datatypes are parametrized by type variables 'apn for principal names, 'il for integrity classification, and 'sl for security classification.

As an example, consider the standard military security classification given by SClass below.

```
val _ = Hol_datatype
 `SClass = Unclassified | Confidential |
           Secret | TopSecret`;
```

We see from Figure 10(a) that the type SecLevel is parametrized by the type variables 'sl for the security classification type and 'apn for a simple principal name type. For example, ``sLab Secret :('pName, SClass) SecLevel`` states that the security level sLab Secret is of type ('pName,SClass)SecLevel. Similarly, ``(sl (Alice :'pName) :('pName, SClass) SecLevel)`` states that Alice's security level sl (Alice:'pName) is of type ('pName,SClass)SecLevel.

Figure 10(b) corresponds to the definition of **Form** in Section III-A. One feature of the HOL definition is that primitive propositions are parametrized by the type variable 'aavar. This feature provides a way for operations on objects to be represented as primitive propositions in access-control formulas.

For example, say we represent subjects and objects as principals in the access-control logic, that we have read and write operations on objects, and we define actions to be operations on objects. We define datatypes Op and Action in HOL by:

```
val _ = Hol_datatype `Op = rd | wrt`;

val _ =
 Hol_datatype `Action = Act of (Op # 'apn Princ)`;
```

With the above definition, we can represent Alice's request to read the object File by the following in HOL:

```
- ``(Alice:'pName Princ) says
     (prop (Act (rd,(File:'pName Princ))))``;
<<HOL message:
  inventing new type variable names: 'a, 'b>>
> val it =
  ``(Alice :'pName Princ) says
    (prop (Act (rd,(File :'pName Princ))) :
    ('pName Action, 'pName, 'a, 'b) Form)`` : term
```

The subject Alice and object File are both principals of type 'pName Princ. Actions are parametrized by 'pName (the type of simple principal names), primitive propositions are based on terms of type 'pName Action, and type variables 'a and 'b created by HOL correspond to the types of integrity and security levels, respectively.

The remaining portions of Figure 10(b) correspond directly to the definition of well-formed formulas in Section III-A. As necessary, operators are defined to be infix operators with appropriate precedence. For space reasons we do not show how this is done here. However, it is a simple declaration that includes specifying operator precedence.

In Section III-A the basic Kripke structure defined is a three-tuple $\langle W, I, J\rangle$, where W is a non-empty set of worlds, I is an interpretation function mapping primitive propositions to sets of worlds in which the given primitive proposition is true, and J a mapping function from principals to a relation relating each world to a (possibly empty) set of worlds. In HOL, we define a data type Kripke out of components that include I and J. The specification of set of worlds W is done by parameterizing I and J in terms of the type variable 'aaworld. Thus, I will be a mapping of primitive propositions to terms of type 'aaworld set. Since all types in HOL are non-empty, we are guaranteed that 'aaworld set is non-empty.

Figure 10(c) shows the definition of the Kripke data type in HOL. KS is the constructor function and it is applied to the following four terms in order:

1) An interpretation function of type 'aavar -> ('aaworld set). This corresponds to interpretation function I mapping primitive terms of type 'aavar to sets of worlds of type 'aaworld set where the terms are true.
2) A relation of type 'aaworld -> ('aaworld set) indexed by simple principal names (parametrized by type variable 'apn). This corresponds to the relation on worlds J, whose type is 'apn -> ('aaworld -> ('aaworld set)).
3) A mapping of simple principals of type 'apn to integrity levels of type 'il.
4) A mapping of simple principals of type 'apn to security levels of type 'sl.

The last two parameters map simple principal names to integrity or security labels. For example, il (Alice:'apn) eqi iLab (Silver:'il) says that Alice's integrity level is Silver and sl (File:'apn) eqs sLab (TopSecret:'sl) says that File's security label is TopSecret.

Figure 10(d) defines accessor functions intpKS, jKS, imapKS, and smapKS to extract the interpretation function I, the relation on worlds indexed by simple principals J, and the mappings of simple principals to integrity and security labels imapKS and smapKS, respectively. These accessor functions return the underlying components of terms of type ('world,'propVar,'pName,'Int,'Sec)Kripke, i.e., Kripke structures parametrized by type variables for worlds ('world), propositions ('propVar), simple principals ('pName), integrity labels ('Int), and security labels ('Sec).

C. Semantics

Using the definitions of principal expressions, integrity and security levels, access-control logic formulas, Kripke structures, and access functions for Kripke structures, we define the

```
val _ = Hol_datatype
    `Princ = Name of 'apn
           | meet of Princ => Princ
           | quoting of Princ => Princ;

     IntLevel = iLab of 'il | il of 'apn;

     SecLevel = sLab of 'sl | sl of 'apn`;
```

(a) Syntax of Principal Expressions, Integrity Levels, and Security Levels

```
val _ = Hol_datatype
 `Form =
    TT
  | FF
  | prop of 'aavar
  | notf of Form
  | andf of Form => Form
  | orf of Form => Form
  | impf of Form => Form
  | eqf of Form => Form
  | says of 'apn Princ => Form
  | speaks_for of 'apn Princ => 'apn Princ
  | controls of 'apn Princ => Form
  | reps of 'apn Princ => 'apn Princ => Form
  | domi of ('apn, 'il) IntLevel => ('apn, 'il) IntLevel
  | eqi of ('apn, 'il) IntLevel => ('apn, 'il) IntLevel
  | doms of ('apn, 'sl) SecLevel => ('apn, 'sl) SecLevel
  | eqs of ('apn, 'sl) SecLevel => ('apn, 'sl) SecLevel
  | eqn of num => num
  | lte of num => num
  | lt of num => num`;
```

(b) Syntax of Formulas

```
val _ = Hol_datatype
 `Kripke =
  KS of ('aavar -> ('aaworld set)) =>
        ('apn -> ('aaworld -> ('aaworld set))) =>
        ('apn -> 'il) => ('apn -> 'sl)`;
```

(c) Definition of Kripke Structures

```
val intpKS_def =
    Define
    `intpKS(KS Intp Jfn ilmap slmap) = Intp`;

val jKS_def =
    Define
    `jKS(KS Intp Jfn ilmap slmap) = Jfn`;

val imapKS_def =
    Define
    `imapKS(KS Intp Jfn ilmap slmap) = ilmap`;

val smapKS_def =
    Define
    `smapKS(KS Intp Jfn ilmap slmap) = slmap`;
```

(d) Accessor Functions for Kripke Structures

Fig. 10: Syntax Definitions in HOL

```
val Jext_def =
    Define `(Jext (J:'pn -> 'w ->'w set) (Name s) = J s) /\
            (Jext J (P1 meet P2) = ((Jext J P1) RUNION (Jext J P2))) /\
            (Jext J (P1 quoting P2) = (Jext J P2) O (Jext J P1))`;

val Lifn_def =
    Define `(Lifn M (iLab l) = l) /\ (Lifn M (il name) = imapKS M name)`;

val Lsfn_def =
    Define `(Lsfn M (sLab l) = l) /\ (Lsfn M (sl name) = smapKS M name)`;
```

(a) Definition of Extended Relation on Worlds and Mappings of Names to Integrity and Security Labels

```
val Efn_def =
    Define
    `(Efn (Oi:'il po) (Os:'is po) (M:('w,'v,'pn,'il,'is) Kripke) TT = UNIV) /\
     (Efn Oi Os M FF = {}) /\
     (Efn Oi Os M (prop p) = ((intpKS M) p)) /\
     (Efn Oi Os M (notf f) = (UNIV DIFF (Efn Oi Os M f))) /\
     (Efn Oi Os M (f1 andf f2) = ((Efn Oi Os M f1) INTER (Efn Oi Os M f2))) /\
     (Efn Oi Os M (f1 orf f2) = ((Efn Oi Os M f1) UNION (Efn Oi Os M f2))) /\
        ....
     (Efn Oi Os M(P says f) = {w | Jext (jKS M) P w SUBSET (Efn Oi Os M f)}) /\
        ....
     (Efn Oi Os M (secl1 doms secl2) = (if repPO Os (Lsfn M secl2) (Lsfn M secl1) then UNIV else {})) /\
     (Efn Oi Os M (secl2 eqs secl1) = (if repPO Os (Lsfn M secl2) (Lsfn M secl1) then UNIV else {}) INTER
                                      (if repPO Os (Lsfn M secl1) (Lsfn M secl2) then UNIV else {})) /\
        ....
```

(b) Definition of Kripke Semantics

Fig. 11: Semantics in HOL

semantics of the logic as shown in Figure 11. The definition of `Jext` in Figure 11(a) corresponds to the definition of $\hat{J}$ in Section III-A. Simple principals in HOL have the form `Name s`; $P1$ & $P2$ in HOL is `P1 meet P2`; $P1 \mid P2$ in HOL is `P1 quoting P2`. Union ($\cup$) and relational composition ($\circ$) are `RUNION` and `O` in HOL.

The mapping of labels and names to integrity and security levels is defined by `Lifn` and `Lsfn` in Figure 11(a). The integrity level of an integrity label `iLab l` is `l`. The integrity level of a simple principal `il name` is `imapKS M name`, where `M` is a Kripke structure. The corresponding function `Lsfn` for security levels is similarly defined.

With all of the above, we define the Kripke semantics as shown in Figure 11(b). The HOL definition of `Efn` corresponds precisely to the definition of $\mathcal{E}_{\mathcal{M}}[\![-]\!]$ in Section III-A.

D. Partial Orders

How partial orders are incorporated into our HOL implementation of the access-control logic is the last remaining detail to disclose. In HOL, we can introduce a new type from elements of an existing type. The elements of an existing type are *representations* of new type. This capability coupled with HOL's support for polymorphism enables us to devise inference rules for *every* partial ordering of integrity and security levels. Our implementation of the logic introduces a new type `('a) po`, i.e., a type consisting of partial orderings of terms of type `'a`. The steps to introduce a new type `'a po` in this fashion are as follows.

1) Use the HOL predicate `WeakOrder` to select partial orderings from relations of type `'a -> 'a -> bool`. The definition of `WeakOrder` is:
   $\vdash \forall Z.$ WeakOrder $Z \iff$
   reflexive $Z \wedge$ antisymmetric $Z \wedge$ transitive Z
2) Prove that `'a po` is non-empty. We prove this by showing that ``` ``=`` ``` satisfies the properties of `WeakOrder`. The theorem is named `WeakOrder_Exists`. For space reasons we omit its proof here.
3) Define the new type `'a po` by executing the ML function
```
val po_type_definition =
 new_type_definition ("po",WeakOrder_Exists);
```
   which returns the value
```
val po_type_definition =
|- ?(rep :'a po -> 'a -> 'a -> bool).
   TYPE_DEFINITION
   (WeakOrder :('a -> 'a -> bool) -> bool) rep
```
4) Prove that `'a po` is isomorphic to the set of partial orderings using the ML function `define_new_type_bijections`. The mapping functions to and from `'a po` to `'a` are the abstraction (constructor) and representation (destructor) functions `PO` and `repPO`. Executing the following
```
val po_bij = save_thm ("po_bij",
 (define_new_type_bijections
 {name="po_tybij", ABS="PO", REP="repPO",
  tyax=po_type_definition}));
```
   yields:
```
|- (!a. PO (repPO a) = a) /\
    !r. WeakOrder r <=> (repPO (PO r) = r : thm
```

As the members of type `'a po` include only relations `Z: 'a -> 'a -> bool` that satisfy `WeakOrder`, all members of type `'a po` must be partial orders, too.

E. Inference Rules

We create inference rules in HOL corresponding to the access-control logic rules as shown in Figures 2, 3, and 4.

$$\textit{MATCH\_MP}\quad \frac{A_1 \vdash \forall x_1..x_n.t_1 \supset t_2 \quad A_2 \vdash t_1'}{A_1 \cup A_2 \vdash \forall x_a..x_k.t_2'}$$

$$\textit{SPEC\_ALL}\quad \frac{A \vdash \forall x_1 \ldots x_n.t}{A \vdash t[x_1'/x_1]\ldots[x_n'/x_n]}$$

Fig. 13: HOL Inference Rules

These rules are typically are created using a combination of built-in HOL inference rules, such as `MATCH_MP` and `SPEC_ALL` shown in Figure 13, and theorems proved about the access-control logic, such as `Modus_Ponens` shown below. `MATCH_MP` pattern matches the antecedent $\mathtt{t_1}$ with $\mathtt{t_1'}$ by instantiating variables $\mathtt{x_1} \ldots \mathtt{x_n}$ appropriately. `SPEC_ALL` specializes all the universally quantified variables of a theorem.

```
val Modus_Ponens =
    |- !M Oi Os f1 f2.
         (M,Oi,Os) sat f1 ==>
          (M,Oi,Os) sat f1 impf f2 ==>
           (M,Oi,Os) sat f2 : thm
```

The function `ACL_MP` is the implementation of Modus Ponens in HOL. The implementation depends on the theorem `Modus_Ponens` above, which is specialized using `SPEC_ALL` and matched to theorem `th1` using `MATCH_MP`. The result is matched to theorem `th2` to by a second application of `MATCH_MP`.

```
fun ACL_MP th1 th2 =
 MATCH_MP(MATCH_MP (SPEC_ALL Modus_Ponens) th1) th2;
```

VI. Examples in HOL

We now show how the virtualization and confidentiality examples in Section IV are implemented and verified in HOL.

A. Virtual Machine

In Section IV-A we postulated and proved a derived inference rule justifying access to virtual address A in a segment addressed by $(base, bound)$ within a physical memory with q addresses. From this description we see that there are three kinds of addresses: (1) a virtual address, (2) a physical address in real memory, and (3) a segment address given by $(base, bound)$. For reasons of type consistency in HOL, we introduce a new type `Addr` that is made from these three address types.

```
val _ =
 Hol_datatype
 `Addr = VA of num | PA of num | SA of num#num`;
```

The particular example had three operations: (1) LDA @A—loading the accumulator ACC with the contents of virtual address A, (2) STO @A—storing the contents of ACC into virtual address A, and (3) *trapping* the execution of an operation. These three operations constitute the `Op` datatype in HOL.

```
val _ =
 Hol_datatype`Op = LDA of Addr | STO of Addr | trap`;
```

Registers such as the instruction register and memory address register can hold values corresponding to operations of type `Op` and addresses of type `Addr`. These values are of type `RVal` defined below.

$$\textit{ACL\_ASSUM}\ f\ \frac{}{(M,O_i,O_s)\ \text{sat}\ f \vdash (M,O_i,O_s)\ \text{sat}\ f} \qquad \textit{ACL\_ASSUM2}\ f\ O_i'\ O_s'\ \frac{}{(M,O_i',O_s')\ \text{sat}\ f \vdash (M,O_i',O_s')\ \text{sat}\ f}$$

$$\textit{ACL\_MP}\ \frac{A_1 \vdash (M,O_i,O_s)\ \text{sat}\ f_1 \quad A_2 \vdash (M,O_i,O_s)\ \text{sat}\ f_1\ \text{impf}\ f_2}{A_1 \cup A_2 \vdash (M,O_i,O_s)\ \text{sat}\ f_2}$$

$$\textit{CONTROLS}\ \frac{A_1 \vdash (M,O_i,O_s)\ \text{sat}\ P\ \text{controls}\ f \quad A_2 \vdash (M,O_i,O_s)\ \text{sat}\ P\ \text{says}\ f}{A_1 \cup A_2 \vdash (M,O_i,O_s)\ \text{sat}\ f}$$

$$\textit{SL\_DOMS}\ \frac{A_1 \vdash (M,O_i,O_s)\ \text{sat}\ (\textit{sl}\ P)\ \text{eqs}\ l_1 \quad A_2 \vdash (M,O_i,O_s)\ \text{sat}\ (\textit{sl}\ Q)\ \text{eqs}\ l_2 \quad A_3 \vdash (M,O_i,O_s)\ \text{sat}\ l_2\ \text{doms}\ l_1}{A_1 \cup A_2 \cup A_3 \vdash (M,O_i,O_s)\ \text{sat}\ (\textit{sl}\ Q)\ \text{doms}\ (\textit{sl}\ P)}$$

Fig. 12: Example Access-Control Inference Rules in HOL

```
val _ =
 Hol_datatype `RVal = Opval of Op | Adval of Addr`;
```

In Figure 6 the request from instruction register IR is:

$$IR\ \textsf{says}\ \langle \text{LDA}\ @\text{A}\rangle.$$

In HOL, the request is:

```
(Name IR) says (prop (Opval (LDA (VA vaddr)))).
```

Recall that HOL inference rules produce theorems in the form of sequents. Our HOL inference rules for the access-control logic produce terms of the form $\Gamma \vdash (\mathcal{M} \models \varphi)$, which have the form $\Gamma \vdash (M, O_i, O_s)$ `sat f` in HOL. Figure 14 is the proof of LDA @A gaining access to virtual address A. The proof corresponds exactly to the manual proof in Figure 6. Looking at the first assumption `a1`, which is the theorem produced by ACL_ASSUM, the result below corresponds to the first assumption in Figure 6.

```
[.] |- (M,Oi,Os) sat
       Name IR says prop (Opval (LDA (VA vaddr)))
```

The values `a1` through `a5` in the HOL proof in Figure 14 correspond to the first five assumptions of the proof in Figure 6. Values `th6` through `th9` correspond to lines 6 through 9 in Figure 6. Finally, the last value `th10` of the HOL proof is the theorem that supports an inference rule of the form $\frac{\mathtt{a}_1\ \mathtt{a}_2\ \mathtt{a}_3\ \mathtt{a}_4\ \mathtt{a}_5}{\mathtt{th}_{10}}$ corresponding to the derived inference rule

$$\frac{\begin{array}{c} IR\ \textsf{says}\ \langle \text{LDA}\ @\text{A}\rangle \qquad RR\ \textsf{says}\ \langle (base, bound)\rangle \\ RR\ \textsf{says}\ \langle (base, bound)\rangle \supset (base + A < q) \supset \\ (A < bound) \supset IR\ \textsf{controls}\ \langle \text{LDA}\ @\text{A}\rangle \\ base + A < q \qquad A < bound \end{array}}{\langle \text{LDA}\ @\text{A}\rangle.}$$

B. Confidentiality

Section IV-B gives an example partial ordering of confidentiality levels and a proof justifying approving Carol's read request on object O. In this section we show the proof in HOL. The HOL proof in Figure 15 closely follows the proof shown in Figure 8.

In HOL we define a new datatype `Op` corresponding to the read and write operations requested by subjects on objects.

```
val _ = Hol_datatype `Op = rd | wrt`.
```

Actions are operations `Op` on principals. The new datatype `Action` is defined below.

```
val _ = Hol_datatype
  `Action = Act of (Op # 'apn Princ)`.
```

This example uses a specific ordering `OSec` on a specified set of security levels. The definition of `OSec` is based on the combination of two partial orderings: `SCOrder` and `Subset`. `SCOrder` is an ordering on security labels `SClass` defined as:

```
val _ = Hol_datatype `SClass = Lo | Hi`;
```

The ordering `SCOrder` has `Hi` dominating `Lo`:

```
val SCOrder_def =
 Define `SCOrder y x =
 if x = Hi then T else if y = Hi then F else T`;
```

`Subset` is used to order the power set of `Categories` defined as:

```
val _ = Hol_datatype`Categories = Bin1 | Bin2`;
```

The remaining steps to defining `OSec` are as follows:

1) Prove both `SCOrder` and `Subset` are partial orderings, i.e., that both satisfy `WeakOrder`. This is straightforward and we skip their proofs for space reasons.
2) With `SCOrder` and `Subset` as partial orderings, we know they are members of `'a po`. Thus, we define
   ```
   |- SCOrder_PO = PO SCOrder

   |- Subset_PO = PO $SUBSET
   ```
   (Note: `$SUBSET` is the prefix version of the infix operator `SUBSET`).
3) We define `OSec` as the composition of `SCOrder_PO` and `Subset_PO`:
   ```
   |- OSec = prod_PO SCOrder_PO Subset_PO,
   ```
   where `prod_PO` is defined as
   ```
   |- !PO1 PO2.
      prod_PO PO1 PO2 =
       PO (RPROD (repPO PO1) (repPO PO2))
   ```
4) Finally, we can prove that `OSec` is a partial ordering as shown in Figure 7.

Our HOL proof corresponding to Figure 8 is shown in Figure 15. The HOL proof introduces four the five assumptions using `ACL_ASSUM2`. We use this instead of `ACL_ASSUM` because we wish to explicitly specify the security partial order `OSec` corresponding to Figure 7. As we have a specific set of labels and a specified ordering, we can prove the following theorem corresponding to the fifth assumption in Figure 8:

```
l2_doms_l1 =
|- (M,Oi,OSec) sat
   sLab (Hi,{Bin1; Bin2}) doms sLab (Lo,{Bin1}) : thm
```

```
- val a1 =
    ACL_ASSUM
    ``((Name IR) says (prop (Opval (LDA (VA vaddr))))):(RVal, 'pName,'Int,'Sec)Form``;
> val a1 =  [.] |- (M,Oi,Os) sat Name IR says prop (Opval (LDA (VA vaddr))) :
  thm
- val a2 =
    ACL_ASSUM
    ``((Name RR) says (prop (Adval (SA (base,bound))))):(RVal, 'pName,'Int,'Sec)Form``;
> val a2 =  [.] |- (M,Oi,Os) sat Name RR says prop (Adval (SA (base,bound))) :
  thm
- val a3 =
    ACL_ASSUM
    ``(((Name RR) says (prop (Adval (SA (base,bound))))) impf
        ((base + vaddr) lt q) impf (vaddr lt bound) impf
         ((Name IR) controls (prop (Opval (LDA (VA vaddr)))))):(RVal, 'pName,'Int,'Sec)Form``;
> val a3 =
     [.]
    |- (M,Oi,Os) sat
       Name RR says prop (Adval (SA (base,bound))) impf
       (base + vaddr) lt q impf vaddr lt bound impf
       Name IR controls prop (Opval (LDA (VA vaddr))) : thm
- val a4 =
    ACL_ASSUM
    ``((base + vaddr) lt q):(RVal, 'pName,'Int,'Sec)Form``;
> val a4 =  [.] |- (M,Oi,Os) sat (base + vaddr) lt q : thm
-  val a5 =
    ACL_ASSUM
    ``(vaddr lt bound):(RVal, 'pName,'Int,'Sec)Form``;
> val a5 =  [.] |- (M,Oi,Os) sat vaddr lt bound : thm
- val th6 = ACL_MP a2 a3;
> val th6 =
     [..]
    |- (M,Oi,Os) sat
       (base + vaddr) lt q impf vaddr lt bound impf
       Name IR controls prop (Opval (LDA (VA vaddr))) : thm
- val th7 = ACL_MP a4 th6;
> val th7 =
     [...]
    |- (M,Oi,Os) sat
       vaddr lt bound impf Name IR controls prop (Opval (LDA (VA vaddr))) : thm
- val th8 = ACL_MP a5 th7;
> val th8 =
     [....] |- (M,Oi,Os) sat Name IR controls prop (Opval (LDA (VA vaddr))) :
  thm
- val th9 = CONTROLS th8 a1;
> val th9 =  [.....] |- (M,Oi,Os) sat prop (Opval (LDA (VA vaddr))) : thm
- val th10 = DISCH_ALL th9;
> val th10 =
    |- (M,Oi,Os) sat Name RR says prop (Adval (SA (base,bound))) ==>
       (M,Oi,Os) sat Name IR says prop (Opval (LDA (VA vaddr))) ==>
       (M,Oi,Os) sat (base + vaddr) lt q ==>
       (M,Oi,Os) sat vaddr lt bound ==>
       (M,Oi,Os) sat
       Name RR says prop (Adval (SA (base,bound))) impf
       (base + vaddr) lt q impf vaddr lt bound impf
       Name IR controls prop (Opval (LDA (VA vaddr))) ==>
       (M,Oi,Os) sat prop (Opval (LDA (VA vaddr))) : thm
```

Fig. 14: HOL Proof of LDA @A Access Request

The values a1 through a4 in the HOL proof in Figure 15 correspond to the first five assumptions of the proof in Figure 8. The theorem l2_doms_l1 corresponds to line 5 in Figure 8. Values th5 through th7 correspond to lines 6 through 8 in Figure 8. Finally, the last value th8 of the HOL proof is the theorem that supports an inference rule of the form $\frac{\mathtt{a_1\ a_2\ a_3\ a_4}}{\mathtt{th_7}}$ corresponding to the derived inference rule

$$\frac{\begin{array}{c} Carol\ \mathsf{says}\ \langle read, O_3\rangle \quad \mathsf{sl}(O_2) =_s (\textsc{lo}, \{Bin_1\}) \\ \mathsf{sl}(Carol) =_s (\textsc{hi}, \{Bin_1, Bin_2\}) \\ \mathsf{sl}(Carol)\ dom\ \mathsf{sl}(O_3) \supset (Carol\ \mathsf{controls}\ \langle read, O_3\rangle) \\ (\textsc{hi}, \{Bin_1, Bin_2\})\ dom\ (\textsc{lo}, \{Bin_1\}) \end{array}}{\langle read, O_3\rangle.}$$

VII. Related Work

We are indebted to the work of Abadi, et. al. for their original formulation of an access-control logic, [13]. Our modifications to their formulation of the logic include:

```
- val a1 =
    ACL_ASSUM2
    ``((Name Carol) says (prop (Act (rd, Name O3))))
       :('pName Action, 'pName,'Int,SClass#(Categories set))Form`` ``Oi:('Int po)`` ``OSec``;
> val a1 =  [.] |- (M,Oi,OSec) sat Name Carol says prop (Act (rd,Name O3)) : thm
- val a2 =
    ACL_ASSUM2
    ``((sl (O3:'pName)) eqs (sLab (Lo,{Bin1}))) :('pName Action, 'pName,'Int,SClass#(Categories set))Form``
    ``Oi:('Int po)`` ``OSec``;
> val a2 =  [.] |- (M,Oi,OSec) sat sl O3 eqs sLab (Lo,{Bin1}) : thm
-  val a3 =
    ACL_ASSUM2
    ``((sl Carol) eqs (sLab (Hi,{Bin1;Bin2}))) :('pName Action, 'pName,'Int,SClass#(Categories set))Form``
    ``Oi:('Int po)`` ``OSec``;
> val a3 =  [.] |- (M,Oi,OSec) sat sl Carol eqs sLab (Hi,{Bin1; Bin2}) : thm
- val a4 =
    ACL_ASSUM2
    ``(((sl Carol) doms (sl O3)) impf ((Name Carol) controls (prop (Act (rd,(Name O3))))))
      :('pName Action, 'pName,'Int,SClass#(Categories set))Form`` ``Oi:('Int po)`` ``OSec``;
> val a4 =
     [.] |- (M,Oi,OSec) sat sl Carol doms sl O3 impf Name Carol controls prop (Act (rd,Name O3)) : thm
- val th5 = SL_DOMS a2 a3 l2_doms_l1;
> val th5 =  [..] |- (M,Oi,OSec) sat sl Carol doms sl O3 : thm
- val th6 = ACL_MP th5 a4;
> val th6 = [...] |- (M,Oi,OSec) sat Name Carol controls prop (Act (rd,Name O3)) : thm
- val th7 = CONTROLS th6 a1;
> val th7 =  [....] |- (M,Oi,OSec) sat prop (Act (rd,Name O3)) : thm
- val th8 = DISCH_ALL th7;
> val th8 =
    |- (M,Oi,OSec) sat Name Carol says prop (Act (rd,Name O3)) ==>
       (M,Oi,OSec) sat sl Carol doms sl O3 impf Name Carol controls prop (Act (rd,Name O3)) ==>
       (M,Oi,OSec) sat sl O3 eqs sLab (Lo,{Bin1}) ==>
       (M,Oi,OSec) sat sl Carol eqs sLab (Hi,{Bin1; Bin2}) ==>
       (M,Oi,OSec) sat prop (Act (rd,Name O3)) : thm
```

Fig. 15: Read Access Proof in HOL

1) adding inference rules as described in this paper,
2) adding delegation in the form of P reps Q on φ as defined here,
3) the elimination of their notion of roles and in its place showing how roles consistent with role-based access control [14] can be described using speaks for and delegation, and
4) the addition of integrity and confidentiality partial orderings as described here.

The logic and examples described here, plus many more examples, appear in our textbook *Access Control, Security, and Trust: A Logical Approach* [15]. This book is a result of our research and teaching efforts for the Air Force Research Laboratory Information Directorate's Advanced Course in Engineering (ACE) Cyber Security Boot Camp [3][4]. Earlier versions of [15] were developed and tested on ACE students. Their facility with Kripke structures and access-control concepts is reported in [16].

We have applied our logic to the analysis of interoperable credentials for JP Morgan Chase [17][18]. This work analyzes the trust relationships in high-value commercial transactions.

For the Air Force Research Laboratory, we have used the methods described here to specify and verify concepts of operations [19]. This work involves trust establishment and preserving integrity of command and control of Air Force systems.

The HOL implementation described here has not been previously disclosed. A much earlier version, which does not use type variables nor partial orders for integrity and confidentiality levels, is described in [20].

VIII. Conclusion

Our overall objective is to provide a logic, method, and tools that are accessible by undergraduate students and practicing engineers to enable them to reason precisely about access control, security, and trust. To date, over 226 ROTC cadets from over 40 universities, over 25 active-duty officers and civilian contractors, and many more Syracuse University students have learned and applied the logic described here and [15] to reason about security. Based on this experience, we conclude that it is reasonable to expect newly-graduated engineers to use mathematics and logic to reason about security. The benefits to students and practitioners include the benefits of logic in general: clear and precise definitions and statements with a system of inference rules to verify conclusions.

The adoption of computer-assisted reasoning tools such as HOL [5] has been slow in part due to the lack of familiarity with functional programming languages and predicate calculus. Our recent experience shows that students who have a rudimentary capability in functional programming can accomplish a significant amount of verification in HOL with just a few weeks of class time. What is even more surprising is that students do not need a specialized course in logic to do credible HOL proofs. We are now incorporating the logic and tools described here into a larger pilot educational program at Syracuse University focusing on the engineering of secure systems.

ACKNOWLEDGMENTS

The work reported here is a result of joint work with Professors Susan Older and Lockwood Morris, Department of Electrical Engineering and Computer Science, Syracuse University. The author gratefully acknowledges their help and contributions. This work was partially supported by the Air Force Research Laboratory and Serco-NA.

REFERENCES

[1] Jerome Saltzer and Michael Schroeder, "The Protection of Information in Computer Systems," *Proceedings IEEE*, 1975.

[2] Matt Bishop, *Computer Security: Art and Science*, Addison Wesley Professional, 2003.

[3] Dan Carnevale, "Basic training for anti-hackers: An intensive summer program drills students on cybersecurity skills," *The Chronicle of Higher Education*, September 23 2005.

[4] Kamal Jabbour and Susan Older, "The advanced course in engineering on cyber security: A learning community for developing cyber-security leaders," in *Proceedings of the Sixth Workshop on Education in Computer Security*, July 2004.

[5] M.J.C. Gordon and T.F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, New York, 1993.

[6] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic verification of concurrent systems using temporal logic specifications," *ACM Trans. on Programming Languages and Systems*, vol. 8, no. 2, 1986.

[7] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison Wesley, 1980.

[8] Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Starrk, and Gordon T. Hamachi, *1990 DECWRL/Livermore Magic Release*, Digital Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA, September 1990, available at http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-90-7.html.

[9] W. Banzhaf, *Computer-Aided Circuit Analysis Using PSpice*, Prentice-Hall, 1992, 2nd edition, 1992.

[10] "Mosis," Available at http://www.mosis.com/.

[11] D. E. Bell and L. J. La Padula, "Secure computer systems: Mathematical foundations," Tech. Rep. Technical Report MTR-2547, Vol. I, MITRE Corporation, Bedford, MA, March 1973.

[12] K. Biba, "Integrity considerations for secure computer systems," Tech. Rep. MTR-3153, MITRE Corporation, Bedford, MA, June 1975.

[13] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 265–310, November 1992.

[14] David F. Ferraiolo, Ravi S. Sandhu, Serban I. Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli, "Proposed NIST Standard for Role-Based Access Control," *ACM Transaction on Information and System Security*, vol. 4, no. 3, pp. 224–274, August 2001.

[15] Shiu-Kai Chin and Susan Older, *Access Control, Security, and Trust: A Logical Approach*, CRC Press, 2011.

[16] Shiu-Kai Chin and Susan Older, "A rigorous approach to teaching access control," in *Proceedings of the First Annual Conference on Education in Information Security*. 2006, ACM.

[17] Glenn Benson, Shiu-Kai Chin, Sean Croston, Karthick Jayaraman, and Susan Older, "Interoperable credentials management for wholesale banking," Tech. Rep. Technical Report 171, Department of Electrical Engineering and Computer Science, Syracuse University, 2011, http://surface.syr.edu/eecs/171/.

[18] Glenn S. Benson, Shiu-Kai Chin, Sean Croston, Karthick Jayaraman, and Susan Older, "Credentials management for high-value transactions," In Kotenko and Skormin [21], pp. 169–182.

[19] Shiu-Kai Chin, Sarah Muccio, Susan Older, and Thomas N. J. Vestal, "Policy-based design and verification for mission assurance," In Kotenko and Skormin [21], pp. 125–138.

[20] Thumrongsak Kosiyatrakul, Susan Older, Polar Humenn, and Shiu-Kai Chin, "Implementing a calculus for distributed access control in higher order logic and HOL," in *Second International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*, Vladimir Gorodetsky, Leonard Popyack, and Victor Skormin, Eds., 2003, vol. 2776.

[21] Igor V. Kotenko and Victor A. Skormin, Eds., *Computer Network Security, 5th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS 2010, St. Petersburg, Russia, September 8-10, 2010. Proceedings*, vol. 6258 of *Lecture Notes in Computer Science*. Springer, 2010.

Science of Mission Assurance

Sarah L. Muccio and Thomas N.J. Vestal
Air Force Research Laboratory
Information Directorate
Rome, NY

Abstract *– We present a scientific framework for assuring mission essential functions in a contested cyber environment.*

1 Introduction

We plan to present a scientific framework for assuring mission essential functions in a contested cyber environment. Mission assurance resides in the avoid, survive and recover stages of the defense cycle. Performing the vulnerability assessment and mitigation steps of the mission assurance process allows us to avoid many threats that adversaries could exercise. Mission assurance is also vital in the survive stage of defense. We must insure our mission will succeed by fighting through attacks and we define this as mission survival. Mission assurance also lives in the recovery phase of the cycle. If you ever make it to the survive stage of an attack, the mission assurance process did not mitigate all of the vulnerabilities. There are now lessons learned and more information on what system vulnerability the attacker exploited; which means you must repeat the cycle all over again to mitigate these newly discovered vulnerabilities. We describe this as the mission recovery phase.

For each stage of the mission apply the following steps.

Prioritization: The first step is to create a detailed outline of the mission essential functions. Subject matter experts (SME) prioritize the mission essential functions according to their effect on the outcome on the mission. The outline must include the level of granularity with which this mission requires visibility into its components. Each mission essential function breaks down into smaller cyber elements. However, depending on their importance in the overall mission prioritization, they may require more or less decomposition. The mission sets of our collaboration partners provide an opportunity to efficiently prioritize realistic mission scenarios.

Mission Mapping: During this next step a mission separates into lower level mission essential functions. The mission essential functions break down further into basic components. The cyber components includes Air Force owned assets, other DoD owned assets and commercial assets used during the mission. A mission essential function rises in importance during the phase of the mission that utilizes it. However, that importance is relative to the mission and the current phase. Additionally, the SMEs narrow the focus to those assets that are most critical for the success of the mission. Essential to understanding and striving toward mission assurance is defining the connection between cyber components in relation to the mission.

Distribution Statement A - Approved for Public Release - Distribution Unlimited Document #88ABW-2011-2500, dated 3 May 2011

Vulnerability Assessment: Throughout the stages of the mission, cyber assets have varying impact of the overall measure of mission assurance. The impact ranges from no effect, mission degradation requiring fight through capabilities, to mission failure. A formal examination of the cyber assets previously mapped out allow for a complete vulnerability assessment.

Mitigation: After identification of the potential vulnerabilities, current research, tools, techniques and mitigation strategies reduce the exposure of the mission. The challenges not solved pave the way for future research and breakthroughs.

2 Acknowledgement

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the U.S. Air Force Research Laboratory, United States Air Force, Department of Defense, or the United States Government. This material is based upon work supported by the Air Force Office of Scientific Research under the Laboratory Research Independent Research Program under grant number FA8750-10-C-0116.

3 Author Information

Sarah L. Muccio is a Mathematician in the Cyber Science branch of the Air Force Research Laboratory's Information Directorate. She received her Ph.D in Applied Mathematics from North Carolina State University in 2007.

Thomas N.J. Vestal is a Computer Engineer in the Cyber Science branch of the Air Force Research Laboratory’s Information Directorate. He is a Computer Engineering Ph.D candidate at Syracuse University and a Member of the IEEE.

A Channel-theoretic Account of Separation Security

Gerard Allwein[1] **and William L. Harrison**[2]
[1]Code 5540, Naval Research Laboratory, Washington, D.C. 20375, USA
[2]Department of Computer Science, University of Missouri, Columbia, MO 65211, USA

Abstract—*It has long been held that information flow security models should be organized with respect to a theory of information, but typically they are not. The appeal of a information-theoretic foundation for information flow security seems natural, compelling and, indeed, almost tautological. This article illustrates how channel theory—a theory of information based in logic—can provide a basis for noninterference style security models. The evidence presented here suggests that channel theory is a useful organizing principle for information flow security.*

1. Introduction

It has long been believed that information flow security should be characterized in terms of a theory of Shannon-style [20] information flow. The problem with this is that the quantity "mutual information" is measured; there is no direction. *Channel theory* [3] is a logical or qualitative theory of information flow; direction is represented explicitly and it is capable of supporting a Shannon-theoretic analysis [2] (although we do not do so here). We submit that channel theory is a natural setting for characterizing information flow security policies and mechanisms, and we support it through the specification of a classic model of information flow security within channel theory in an elegant manner.

Most information flow security models are based either on the *noninterference* model of Goguen and Meseguer [7] or on variants of it [21]. Such security models specify end-to-end system security policies in terms of the "views" of the system as a whole by groups of users/processes. Views are typically characterized by partitioning global system inputs and outputs and associating groups of users/processes with these partitions. These input and output partitions determine the view of its associated group. Noninterference-based security policies will require that, for example, changes in a high level security input partition will result in no change to a low level output partition.

Channel theory is also known as the "logic of distributed systems", where "distributed systems" is interpreted in a very broad sense. In channel theory, each subsystem is formulated as a *local logic* which may be intuitively understood as characterizing the subsystem's view of the distributed system as a whole. Connecting these subsystems is a *channel* that governs how theorems in one local logic may be transferred to another local logic. The intuitive parallel with noninterference-based security is explicated here and made formal. The view according to security level is determined here by a local logic and a channel that governs information flow between levels.

By combining several formalisms from logic and semantics, this paper admittedly places demands on the reader beyond what is perhaps usually expected. Channel theory, in particular, is not broadly known and consequently the presentation here must introduce a number of its key concepts before proceeding. The security property we consider—separation—is simple, but as this is the first application of Channel theory to information security, simplicity is a virtue. But more importantly, this paper elaborates the necessary channel theoretic underpinnings (especially co-channels) and sets the stage for further applications of Channel Theory to information security.

Channel theory combines syntax and semantics and hence, the proofs are necessarily semantic in content. This has the sense of soundness from logic, i.e., one shows a particular (Gentzen) sequent holds semantically. The sequents are in a second order logic which quantifies (semantically) over all actions in a program. The combination of the use of co-channels to distribute an invariant and the invariant being a sequent in a simple second order logic is what allows the statement and proof of separation to be so clean.

This article proceeds from research characterizing the classic, noninterference-style security design of Rushby (known as a separation kernel [18]) in terms of monadic language semantics [11], [10]. A *separation kernel*, K (pictured in Figure 1), mediates communication between the high and low level domains, H and L, respectively. Domains H and L may only communicate with the kernel via the channels f and g, respectively. The security policy boils down to demonstrating that inputs to H have no impact on the outputs of L. The security property associated with the separation kernel in Figure 1 is based on the notion that any operation executing in the high security domain H should have no effect whatsoever on the threads executing in the low security domain L.

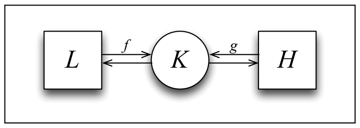

$$\begin{tikzcd} L \arrow[r, leftrightarrow, "f"] & K \arrow[r, leftrightarrow, "g"] & H \end{tikzcd}$$

Fig. 1: Sep. Kernel as a Channel Theory Flow Diagram.

In terms of individual atomic operations, this can be further specified as follows. Note that any system execution consists of a sequence of H and L operations determined by some scheduling strategy. The security specification requires that any system execution, $h_0\ ;\ l_0\ ;\ \cdots\ ;\ h_n\ ;\ l_n$, has precisely the same effect on the L domain as this same execution stripped of H operations, $l_0 ; \cdots ; l_n$. This security regime is not, in general, process isolation. Operations in H may not affect those in L, but operations in L are free to influence those in H. Separation requires the converse as well.

The Contributions of This Paper.

Monadic transformers [13], [12] are shown to provide a layering as a scaffold for channel theoretic objects and the relationships among the layers are represented using the morphisms of channel theory (the double-pairs of maps f and g from Figure 1). Through channel theory, by a judicious use of limits, colimits, and a free-variable second-order logic, separation kernels [11], [10] are show to validate the necessary logical statements that express separation. The resulting distributed logical apparatus of channel theory can then be seen to provide an straightforward proof of separation without collapsing the information in the monadic layers into a single but hard to manipulate formal apparatus. The burden of verification is thus made easier.

a) Related Work.: Channel theory is not a logic but, rather, it is a logical framework that allows separation of concerns (in the sense of Dijkstra [5]) at the level of logical specification. It is similar to institutions [6] in that both have similar objects and morphisms. In contrast to institutions, channels are the central organizing principle of channel theory, where channels and co-channels are used for the transfer of theorems between local logics. Separation logic [17] introduces separation of concerns at the level of logical connectives. The channel theoretic characterization of a monadic separation kernel presented here is factored according to the monadic layers underlying the kernel's construction, although channel theory does not focus on monadic specifications exclusively (in contrast to evaluation logic [16], HasCasl [19], or observational program specification [9]). Abadi, et al., [1] formulate notions of dependency (including noninterference) in terms of the *dependency core calculus*. Crary et al. [4] consider a logical characterization of information flow security that, like DCC, has Moggi's computational lambda calculus [14] at its core. The second author's monadic encapsulation of separation [11], [10] is more semantic and model-theoretic than either of these more logical and type-theoretic approaches. The present article answers an open question by making the relationship between such semantic and logical views apparent and precise. Channel theory has not, to the authors' best knowledge, ever been applied to information flow security.

2. Background

This article makes a connection between a number of different formalisms across logic, denotational semantics and

$$
\begin{array}{ll}
\mathit{StateT}\ \Sigma\ M\ X & = \ \Sigma \to M(X \times \Sigma) \\
\eta_S\ v & = \lambda\sigma.\ \eta_M\,(v,\sigma) \\
x\ \star_S\ f & = \lambda\sigma_0.\,(x\ \sigma_0) \star_M \lambda(v,\sigma_1).\,f\ v\ \sigma_1 \\
\mathsf{g}\ :\ S\ \Sigma & = \lambda\sigma.\ \eta_M\,(\sigma,\sigma) \\
\mathsf{u} : (\Sigma \to \Sigma) \to S() & \\
\mathsf{u}\,f & = \lambda\sigma.\ \eta_M((), f\ \sigma) \\
\mathit{lift}\ \varphi & = \lambda\sigma.\,\varphi\ \star_M\ (\eta_M \circ (\lambda v.\,(v,\sigma))) \\
 & \\
x \gg y & = x\ \star\ \lambda d.\,y \quad \text{— “null” bind}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{ResT}\ M\ X & = \mathrm{fix}\,\xi.\ X\ +\ M\ \xi \\
\eta_R\ v & = \delta\ v \\
(\delta\ x)\ \star_R\ f & = f\ x \\
(\rho\,\varphi)\ \star_R\ f & = \rho\,(\varphi\ \star_M\ \lambda\kappa.\ \eta_M\,(\kappa\ \star_R\ f)) \\
\mathit{step}\ \varphi & = \rho\,(\varphi\ \star_M\ (\eta_M \circ \eta_R)) \\
\mathit{run}\ (\delta\ v) & = \ \eta_M\ v \\
\mathit{run}\ (\rho\,\varphi) & = \varphi\ \star_M\ \mathit{run}
\end{array}
$$

Fig. 2: Monad Transformers

information security and, to make it as self-contained as possible, overviews of the necessary background material is presented here. First, noninterference security and separation are discussed and then it is summarized how separation may be realized via monadic language semantics [13], [12]. A brief overview of modular monadic semantics is presented as well. It is assumed of necessity that the reader is familiar with monadic semantics, and, for those wishing more background on monadic semantics, please consult the references. In particular, for an overview of monads for concurrency (i.e., resumption monads), please consult Harrison [8].

Separation Security & Noninterference.

This security regime, which we will refer to as *separation*, is an instance of Goguen and Meseguer noninterference. Separation considers the "static case" where there is no passing of capabilities. Such policies formulate information security in terms of "views" or perspectives with respect to processes or users at the low security level. We assume, without loss of generality that there are precisely two such levels, high and low). If two system inputs, i and i', are the same from the low view (sometimes written $i \approx_L i'$), then, for any system execution consisting of a sequence of high and low operations, $h_0; l_0; \cdots; h_n; l_n$, the following system outputs are the same from the low view: $\mathit{out}\ (h_0; l_0; \cdots; h_n; l_n) \approx_L \mathit{out}\ (l_0; \cdots; l_n)$. Noninterference specifications are typically formulated in terms of abstract state machines.

3. Definition of Channel Theory

The basic structures of channel theory are deceptively simple. The things that are distributed in a distributed system are contexts called *classifications*. The classifications are connected by *infomorphisms*. The relevant definitions follow.

A classification contains two distinct collections of objects, tokens and types. They could be anything that makes sense in using a classification as a model. However, most of modern language theory tends to use the term *types* in a different sense. In this paper, channel theory's types are always termed *propositions*. The tokens are analogous to states or program actions. Bold slanted typeface is always used to denote classifications.

Definition 3.0.1 A *classification*, $\boldsymbol{X}$, is a pair of sets, $Tok(\boldsymbol{X})$, and $Prop(\boldsymbol{X})$, and a relation, $\models_{\boldsymbol{X}} \subseteq Tok(\boldsymbol{X}) \times Prop(\boldsymbol{X})$ written in infix, e.g., $x \models_{\boldsymbol{X}} A$. $x \models_{\boldsymbol{X}} A$ is the qualitative unit of information that flows in channel theory.

Classifications occur wherever models of formal systems are found. Channel theory has its own notion of morphism, called an infomorphism. It is similar to a pair of adjoint functors in that it is a pair of opposing arrows with a condition similar to the adjoint's bijection.

Definition 3.0.2 An *infomorphism* $h : \boldsymbol{X} \to \boldsymbol{Y}$ of classifications is a pair of contravariant maps, $\overrightarrow{h}$ and $\overleftarrow{h}$ such that $\overrightarrow{h} : Prop(\boldsymbol{X}) \to Prop(\boldsymbol{Y})$ and $\overleftarrow{h} : Tok(\boldsymbol{Y}) \to Tok(\boldsymbol{X})$, and for all x and A, the following condition is satisfied, $x^h \models_{\boldsymbol{X}} A$ iff $x \models_{\boldsymbol{Y}} A^h$,. For ease of presentation, $\overleftarrow{h}(x)$ is displayed as x^h and $\overrightarrow{h}(A)$ as A^h.

A commuting *finite cocone* consists of a graph homomorphism G from a finite graph to the category of classifications, a vertex classification $\boldsymbol{C}$, and a collection of arrows $g_i : G(i) \to \boldsymbol{C}$. It is required that for all $f : i \to j$, $g_i = g_j \circ G(f)$. The base of the cocone is the objects and arrows identified by G. There will also be a need for cones: a *finite cone* consists of a graph homomorphism G from a finite graph to the category of classifications, a vertex classification $\boldsymbol{C}$, and a collection of arrows $g_i : G(i) \to \boldsymbol{C}$. It is required that for all $f : j \to i$, $g_i = G(f) \circ g_j$. Just reverse all the arrows in the preceding diagram.

$$\begin{tikzcd}
 & & & \boldsymbol{C} & & \\
G(1) \arrow[urrr, "g_1"] & \cdots & G(i) \arrow[ur, "g_i"'] \arrow[r, "G(f)"'] & G(j) \arrow[u, "g_j"] & \cdots & G(n) \arrow[ull, "g_n"']
\end{tikzcd}$$

Definition 3.0.3 An *information channel* is a co-cone in the category of classifications and infomorphisms. An *information co-channel* is a cone in the category of classifications and infomorphisms. $\boldsymbol{C}$ in the diagram is call the core of the (co)channel.

The smallest channel over a base is a colimit. Frequently, the smallest channel is not the most useful because a channel is used as a model. The smallest channel would simply connect the base with no additional modeling apparatus. A colimit in the category of classifications is a colimit on propositions and a limit on tokens.

Assuming a fixed classification $\boldsymbol{C}$, a *Gentzen sequent*, $\Gamma \Vdash_{\boldsymbol{C}} \Delta$ is two sets of propositions connected by a relation $\Vdash$. A *valid* sequent has the force of a meta-level implication of form: for all tokens x, if $x \models_{\boldsymbol{C}} A$ for all of the propositions A in Γ, then at $x \models_{\boldsymbol{C}} B$ for at least proposition in Δ. In this paper, all our sequents will be rather simple and of the form $A \Vdash B$. Sequents are used to represent constraints for a classification. In the core of a channel, sequents underwrite information flow in the core.

Definition 3.0.4 A *local logic* $\mathfrak{L} = \langle \boldsymbol{C}, \Vdash_{\mathfrak{L}}, N_{\mathfrak{L}} \rangle$ consists of a classification $\boldsymbol{C}$, a set $\Vdash_{\mathfrak{L}}$ of sequents involving the types of $\boldsymbol{C}$, and a subset $N_{\mathfrak{L}} \subseteq Tok(\boldsymbol{C})$ called the *normal tokens* of $\mathfrak{L}$, which satisfy all the constraints $\Vdash_{\mathfrak{L}}$. A local logic $\mathfrak{L}$ is sound if every token is normal; it is complete if every sequent that holds of all normal tokens is in the consequence relation $\Vdash_{\mathfrak{L}}$.

Each classification supports a local logic, including cores of channels. A non-normal token represents a counter-example to the theory. In this paper, only normal tokens are used. Non-normal tokens can be used to introduce conditional probabilities associated with the sequents. Typically, the sequents are required to follow certain *structural rules* but these will not concern us in this paper. These non-structural rules allow for the movement of logics forward along an infomorphism $f : \boldsymbol{X} \to \boldsymbol{Y}$, where Γ^{-f} is an abbreviation for $\overrightarrow{f}^{-1}(\Gamma)$, i.e., the inverse image of Γ under f and Δ^f is the direct image of Δ under f. There are two equivalent forms for each; f-Intro preserves validity and f-Elim preserves non-validity.

$$\frac{\Gamma \Vdash_{X} \Delta}{\Gamma^{f} \Vdash_{Y} \Delta^{f}}\ (\text{f}-Intro)$$

$$\frac{\Gamma \Vdash_{Y} \Delta}{\Gamma^{-f} \Vdash_{X} \Delta^{-f}}\ (\text{f}-Elim)$$

Channels and co-channels are used to hold channel logics. In the core of a channel, a logic can be used to underwrite or authorize information transfer among the side classifications. Co-channels, as they are used in this paper, are used to distribute a common logic in the core to the side classifications.

4. A Channel Account of Separation

The channel theory diagram System Classification (see Figure 3) will be annotated with theorems where $Tok(\boldsymbol{Hi})$ and $Tok(\boldsymbol{Lo})$ are the set of program states for Hi and Lo respectively, $\boldsymbol{Hi}_k$ represents the k-th operation of Hi, $\boldsymbol{C}_H$ is a co-channel which is used for distributing a constraint (expressed as a sequent), $\boldsymbol{H}$ is constructed using the monad $H = StateT\ Hi\ Id$, $\boldsymbol{K}$ is constructed using the monad $K = \mathit{StateT}\ \mathit{Hi}\ (\mathit{StateT}\ \mathit{Lo})\ \mathit{Id}$, $\boldsymbol{R}$ is constructed using the resumption monad. There is a similar diagram below $\boldsymbol{L}$ (using the Lo versions of $\boldsymbol{C}_H$, etc.) as below $\boldsymbol{H}$. Let two program statements be $D := D + 1$ and $D := D - 1$. The first is the Hi operation and the second is the Lo operation. These statements are the actions $D \mapsto D + 1$ and $D \mapsto D - 1$. The other computations needed are $D \mapsto D$ denoted $1_{\boldsymbol{Hi}}$ and $1_{\boldsymbol{Lo}}$ for the respective sides; these are needed for internal housekeeping in the sequel. There are only two $\boldsymbol{Hi}_k$'s in the sample system, namely $\boldsymbol{Hi}_1$ for $1_{\boldsymbol{Hi}}$ and $\boldsymbol{Hi}_2$ for the computational interpretation of $D := D + 1$.

$$\begin{tikzcd}
 & & & & \boldsymbol{R} & & \\
 & & \boldsymbol{K} \arrow[urr] & \cdots & \boldsymbol{K} \arrow[u] & \cdots & \boldsymbol{K} \arrow[ull] \\
 & & \boldsymbol{C}_H \arrow[r] \arrow[dl] \arrow[dr] & \boldsymbol{H} \arrow[ur, "\iota_H"'] & & \boldsymbol{L} \arrow[ul, "\iota_L"] & \\
 & \boldsymbol{Hi}_1 & \cdots & \boldsymbol{Hi}_k & \cdots & & \\
 & & \boldsymbol{Hi} \arrow[ur, "\iota_1"'] & & \boldsymbol{Hi} \arrow[ul, "\iota_2"] & &
\end{tikzcd}$$

Fig. 3: Sys. Class./Monadic Layers

Notice that the fragment of the diagram containg $\boldsymbol{C}_H, \boldsymbol{Hi}_1, \boldsymbol{Hi}_2, \ldots$ has its arrows pointing downward in contradistinction to the rest of the arrows pointing upward. This is an instance of a cone whereas the other fragments, all of whose arrows have a common target, are cocones. $\boldsymbol{C}_H, \boldsymbol{Hi}_1, \boldsymbol{Hi}_2, \ldots$ is an example of a co-channel. The lone sequent that will be in $\boldsymbol{C}_H$ will be distributed to the $\boldsymbol{Hi}_k$.

Let $\mathfrak{a}_2 = \lambda v.v + 1$, the sequent for $\boldsymbol{Hi}_2$, $\iota_1(D = V) \Vdash_{\boldsymbol{Hi}_2} \iota_2(D = \mathfrak{a}_2 V)$ will be denoted as: $D = V \Vdash_{\boldsymbol{Hi}_2} \mathfrak{a}_2(D = V)$, eliding the ι_i and making the $\mathfrak{a}$ appear as modal operator. Now for the cone with vertex $\boldsymbol{C}_H$ and base the $\boldsymbol{Hi}_k$. There is an infomorphism (Figure 4) where ϕ_k is the identity map and $Prop(\boldsymbol{C}_H) = Prop(\boldsymbol{Hi}_k)$ for all k. The state pairs $\langle s, s' \rangle$ in $Tok(\boldsymbol{Hi}_k)$ are just those state pairs that satisfy $s \xrightarrow{\mathfrak{a}_k} s'$, where $\xrightarrow{\mathfrak{a}_k}$ symbol means s goes to s' under the action $\mathfrak{a}_k$. Let $\psi_k(\langle s, s' \rangle) = \langle \mathfrak{a}_k . \langle s, s' \rangle \rangle \in Tok(\boldsymbol{C}_H)$. We define $\langle \mathfrak{a}_k, \langle s, s' \rangle \rangle \models_{\boldsymbol{C}_H} Q$ iff $\langle s, s' \rangle \models_{\boldsymbol{Hi}_k} Q$ for any proposition Q. This is a "safe" definition since it will lead to no new sequents holding as constraints in $\boldsymbol{C}_H$ that do not already hold in each of the $\boldsymbol{Hi}_k$; the proposition sets are all identical. This defines $\langle \varphi_k, \psi_k \rangle$ as an infomorphism.

$$\begin{tikzcd}
Prop(\boldsymbol{C}_H) \arrow[rr, "\phi_k"] \arrow[d, dash, "\models"'] & & Prop(\boldsymbol{Hi}_k) \arrow[d, dash, "\models"] \\
Tok(\boldsymbol{C}_H) & & Tok(\boldsymbol{Hi}_k) \arrow[ll, "\psi_k"]
\end{tikzcd}$$

Fig. 4: $\boldsymbol{C}_H$ to $\boldsymbol{Hi}_k$ Infomorphism

To pull the sequent back along ϕ might, given the rules for sequents, result in an invalid sequent in $\boldsymbol{C}_H$. However, notice that this sequent holds for all k under the condition that each $\boldsymbol{Hi}_k$ interprets $\mathfrak{a}$ to the operation $\mathfrak{a}_k$. In this way, the sequent, $D = V \Vdash_{C_H} \mathfrak{a}(D = V)$, becomes a sequent of second-order free-variable logic. However, the sequent still retains its modal character, just as in $\boldsymbol{Hi}_k$. The job of ϕ_k is to distribute this sequent to each $\boldsymbol{Hi}_k$.

Consider each channel $(\boldsymbol{Hi}_k, \boldsymbol{Hi})$ to be a model for a modal logic with the operator $\mathfrak{a}_k$ defining the modality. Each $\boldsymbol{Hi}_K$ contains all pairs of states $\langle s, s' \rangle$ relating s to s' under the action $\mathfrak{a}_k$. Each s and s' must provide a value for D and V. Some values are such $D \neq V$ in which case the antecedent of the sequent is false and hence the sequent evaluates as true; the sequent is then satisfied *spuriously*. In the cases where $D = V$, the consequent follows because $\boldsymbol{Hi}_k$ is the core of the channel for $\mathfrak{a}_k$. Hence the sequent is valid. So $\boldsymbol{Hi}_k$ contains all the models for the logic and the action.

The arrow ψ_k is constructing part of a (flattened) second order model structure in $Tok(\boldsymbol{C}_H)$. It is easier to think of there being a single token in $Tok(\boldsymbol{C}_H)$ for each $\boldsymbol{Hi}_k$. This token needs to pull out a pair from its modal relation to evaluate a proposition, i.e.,

$$\begin{array}{lll}
\langle a_k, \langle s, s' \rangle \rangle \models_{\boldsymbol{C}_H} \iota_1(D = V) & & \\
\quad \text{iff } \psi_k(\langle s, s' \rangle) \models_{\boldsymbol{C}_H} \iota_1(D = V) & & \text{def. of } \psi_k \\
\quad \text{iff } \langle s, s' \rangle \models_{\boldsymbol{Hi}_k} \phi_k(\iota_1(D = V)) & & \text{infomorphism} \\
\quad \text{iff } \langle s, s' \rangle \models_{\boldsymbol{Hi}_k} \iota_1(D = V) & & \text{def. of } \phi_k \\
\quad \text{iff } \pi_1(\langle s, s' \rangle) \models_{\boldsymbol{Hi}} D = V & & \text{infomorphism} \\
\quad \text{iff } s \models_{\boldsymbol{Hi}} D = V & & \text{def. of } \pi_1
\end{array}$$

In this way, the work load of evaluating propositions, and hence sequents, is distributed via channel theory.

In the sequel, $\mathrm{op}(\mathit{class})$ will yield the set of operations of the classification class. Hence $\mathrm{op}(\boldsymbol{C}_H) = \{\mathfrak{a}_k \mid \langle \mathfrak{a}_k, \langle s, s' \rangle \rangle \in Tok(\boldsymbol{C}_H)\}$. Similarly, $\mathrm{op}(\mathit{token})$ will yield the operation hiding in a single token.

The token set $Tok(\boldsymbol{C}_H)$ is then a disjoint union of interpretations, each partition contains an entire second-order logic model. The second-order logic used is tightly constrained for the application in this paper. There is no explicit quantifying over function variables (or predicates), hence the moniker *free-variable*. The implicit universal quantification over free function variables is bounded by the classifications needed to evaluate the propositions, it is bounded by the structure of the system being considered.

4.1 The Classifications $\boldsymbol{H}$ and $\boldsymbol{L}$

We carry out the definitions for the H side, the L side is analogous. The $\boldsymbol{H}$ has the same proposition set as $\boldsymbol{C}_H$ and as tokens pairs of the form $\langle \mathfrak{u}_H\ \mathfrak{a}_k, \langle s, \langle (), s' \rangle \rangle \rangle$. The computation $\mathfrak{u}_H \mathfrak{a}_k$ is a computation in S. The definitions

$$\mathfrak{u}_H\ \langle \mathfrak{a}_k, \langle s, s' \rangle \rangle \overset{def}{=} \langle \mathfrak{u}_H\ \mathfrak{a}_k, \langle s, s' \rangle \rangle, \quad deHf \overset{def}{=} \lambda v.\pi_2(f\ v)$$

(where π_2 projects the second element of a pair) are used to construct the infomorphism from $\boldsymbol{C}_H$ to $\boldsymbol{H}$ which is the identity on the propositions and is a projection $\pi_{C_H} : Tok(\boldsymbol{H}) \to Tok(\boldsymbol{C}_H)$ on the tokens with the definition

$$\pi_{C_H}\langle \mathsf{u}_H\ \mathfrak{a}_k, \langle s, \langle (), s'\rangle\rangle\rangle \overset{def}{=} \langle deH\ (\mathsf{u}_H\ \mathfrak{a}_k), \langle s, s'\rangle\rangle,$$

where $s \xrightarrow{\mathfrak{a}_k} s'$. It is easily seen that $\pi_{C_H} \circ \mathsf{u}_H = 1_{Tok(C_H)}$ and $\mathsf{u}_H \circ \pi_{C_H} = 1_{Tok(\boldsymbol{H})}$. That this is an infomorphism is by stipulation. The proposition sets are the same and the token sets are isomorphic. No new constraints will hence be generated and the old one is preserved (see the $H = StateT\ Hi\ Id$ rule below).

In the sequel, $\mathrm{op}(\boldsymbol{H}) = \{f \mid \langle f, \langle s, \langle (), s'\rangle\rangle\rangle \in Tok(\boldsymbol{H})\} \subseteq S()$. Computations used in $\mathrm{op}(\boldsymbol{H})$ and $\mathrm{op}(\boldsymbol{L})$ are of type $a \to t \times a$. The two statements $D := D + 1$ and $D := D - 1$ are compiled into

$$\begin{aligned} inc &= [\![D+1]\!] \star_s \lambda v.\mathsf{u}_H(D \mapsto v),\\ dec &= [\![D-1]\!] \star_s \lambda v.\mathsf{u}_L(D \mapsto v)\end{aligned}$$

and hence $\mathrm{op}(\boldsymbol{H}) = \{inc, \eta_\mathsf{S}\}$ and $\mathrm{op}(\boldsymbol{L}) = \{dec, \eta_\mathsf{S}\}$. One might consider treating the construction of $\boldsymbol{H}$ to be analogous to the monad transformer $StateT$. The $\boldsymbol{Hi}_k$ is a construction diagram where the projections deconstruct the channel. Thought of in this way, then

$$\frac{D = V \Vdash_{C_H} \mathfrak{a}(D = V)}{D = V \Vdash_{\boldsymbol{H}} \mathfrak{a}(D = V)}\ StateT\ Hi\ Id$$

becomes simply the sequent in $\boldsymbol{C}_H$ moved forward (the sound direction) along an infomorphism.

4.2 The Classification $\boldsymbol{K}$

Let $\boldsymbol{X} \in \{\boldsymbol{H}, \boldsymbol{L}\}$, and for all tokens β in $\boldsymbol{X}$, let $\beta \models_{\boldsymbol{X}} \Phi_{\boldsymbol{X}}$. The sequents

$$\begin{aligned}\Phi_{\boldsymbol{X}} \wedge \boldsymbol{H}(D = V) \wedge \boldsymbol{L}(D = U) \Vdash_{\boldsymbol{K}}\\ \boldsymbol{H}(\mathfrak{a}(D = V)) \wedge \boldsymbol{L}(\mathfrak{b}(D = U))\end{aligned}$$

will be evaluated using the monad K where $K = StateT\ Hi\ (StateT\ Lo)\ Id$. The tags $\boldsymbol{H}$ and $\boldsymbol{L}$ indicate from which classification the propositions came and are the result of the infomorphisms from $\boldsymbol{H}$ and $\boldsymbol{L}$ being injections on propositions. We let the $\Phi_{\boldsymbol{X}}$ be unaltered by the injections from $\boldsymbol{H}$ and $\boldsymbol{L}$ and note that

$$\Phi_{\boldsymbol{X}} \wedge D = V \Vdash_{\boldsymbol{X}} \mathfrak{x}(D = V)$$

holds in $\boldsymbol{X}$ where $\mathfrak{x} = \mathfrak{a}$ if $\boldsymbol{X} = \boldsymbol{H}$, and $\mathfrak{x} = \mathfrak{b}$ if $\boldsymbol{X} = \boldsymbol{L}$. The operations in K will be restricted to the operations injected by $outer \circ \mathsf{u}_H$ and $inner \circ \mathsf{u}_L$.

The construction for the channel $\boldsymbol{K}$ acts like the monad K. Specifically, the sequents of $\boldsymbol{H}$ are operated on by "wrapping them" with $\boldsymbol{L}$ sequents. In our simple case, the sequent transformation can be constructed with the rule:

$$\frac{\begin{array}{c}\Phi_{\boldsymbol{H}} \wedge D = V \Vdash_{\boldsymbol{H}} \mathfrak{a}(D = V) \text{ or}\\ \Phi_{\boldsymbol{L}} \wedge D = U \Vdash_{\boldsymbol{L}} \mathfrak{b}(D = U)\end{array}}{\begin{array}{c}\Phi_{\boldsymbol{X}} \wedge \boldsymbol{H}(D = V) \wedge \boldsymbol{L}(D = U) \Vdash_{\boldsymbol{K}}\\ \boldsymbol{H}(\mathfrak{a}(D = V)) \wedge \boldsymbol{L}(\mathfrak{b}(D = U))\end{array}}\ K$$

This is two rules with Φ_X being appropriately $\Phi_{\boldsymbol{H}}$ or $\Phi_{\boldsymbol{L}}$. The computations necessary for separation are those injected into the monad K using $outer(\mathsf{u}_H\ \mathfrak{a})$ and $inner(\mathsf{u}_L\ \mathfrak{b})$. These computations are of type $a \to b \to t \times a \times b$. Tokens in $\boldsymbol{K}$ are of the form $\langle f, \langle s, q, \langle\langle (), s'\rangle, q'\rangle\rangle\rangle$. In our sample system, $f \in \{outer\ inc, outer\ \eta_\mathsf{S}, inner\ dec, inner\ \eta_\mathsf{S}\}$. More generally,

$$\begin{aligned}\mathrm{op}(\boldsymbol{K}) = \{outer\ \ f \mid f \in \mathrm{op}(\boldsymbol{H})\} \cup\\ \{inner\ \ f \mid f \in \mathrm{op}(\boldsymbol{L})\}.\end{aligned}$$

The infomorphism $\mathbb{\iota}_{\boldsymbol{H}} : \boldsymbol{H} \to \boldsymbol{K}$ has a token morphism $\pi_{\boldsymbol{H}}$.

Definition 4.2.1 Let $prj_H(t, a, b) = (t, a)$ and $prj(t, a, b) = (t, b)$ and

$$\begin{aligned} deK_H f &\overset{def}{=} \lambda v.prj_H(f\ v\ undefined),\\ deK_L f &\overset{def}{=} \lambda v.prj_L(f\ v\ undefined),\end{aligned}$$

then $\pi_{\boldsymbol{H}} : Tok(\boldsymbol{K}) \to Tok(\boldsymbol{H})$ and $\pi_{\boldsymbol{L}} : Tok(\boldsymbol{K}) \to Tok(\boldsymbol{L})$ are defined as

$$\begin{aligned}\pi_{\boldsymbol{H}}\langle f, \langle s, q, \langle\langle (), s'\rangle, q'\rangle\rangle\rangle &\overset{def}{=} \langle deK_H\ f, \langle s, \langle (), s'\rangle\rangle\rangle,\\ \pi_{\boldsymbol{L}}\langle f, \langle s, q, \langle\langle (), s'\rangle, q'\rangle\rangle\rangle &\overset{def}{=} \langle deK_L\ f, \langle q, \langle (), q'\rangle\rangle\rangle.\end{aligned}$$

These definitions reveal the objects $K()$ and $Tok(\boldsymbol{K})$ act like products. Since $\mathrm{op}(\boldsymbol{K}) \subseteq K$, these projections may be restricted to $\mathrm{op}(\boldsymbol{K})$. The classification $\boldsymbol{K}'$ where $Prop(\boldsymbol{K}') = Prop(\boldsymbol{K})$, $Tok(\boldsymbol{K}') = Tok(\boldsymbol{H}) \times Tok(\boldsymbol{L})$ is the co-limit of $\boldsymbol{H}$ and $\boldsymbol{L}$ (recall colimits in the category of classifications entails being a limit on tokens). Hence there is a pair of infomorphisms $k : \boldsymbol{H} \to \boldsymbol{K}'$ and $k' : \boldsymbol{L} \to \boldsymbol{K}'$. It is an easy observation that the restriction of an infomorphism from restricting the domain of the token map is still an infomorphism. Since $Tok(\boldsymbol{K}) \simeq Tok(\boldsymbol{H}) + Tok(\boldsymbol{L})$, the restriction of k and k' to $Tok(\boldsymbol{K})$ yields infomorphisms connecting $\boldsymbol{H}$ with $\boldsymbol{K}$ and $\boldsymbol{L}$ with $\boldsymbol{K}$.

Theorem 4.2.2 *For all $\beta \in Tok(\boldsymbol{K})$, β satisfies*

$$\begin{aligned}\Phi_{\boldsymbol{X}} \wedge \boldsymbol{H}(D = V) \wedge \boldsymbol{L}(D = U) \Vdash_{\boldsymbol{K}}\\ \boldsymbol{H}(\mathfrak{a}(D = V)) \wedge \boldsymbol{L}(\mathfrak{b}(D = U))\end{aligned}$$

From the fact that the morphisms from $\boldsymbol{H}$ and $\boldsymbol{L}$ to $\boldsymbol{K}$ are infomorphisms, the two sequents

$$\begin{aligned}\Phi_{\boldsymbol{H}} \wedge \boldsymbol{H}(D = V) \Vdash_{\boldsymbol{K}} \boldsymbol{H}(\mathfrak{a}(D = V)),\\ \Phi_{\boldsymbol{L}} \wedge \boldsymbol{L}(D = U) \Vdash_{\boldsymbol{K}} \boldsymbol{L}(\mathfrak{b}(D = U))\end{aligned}$$

are constraints holding in $\boldsymbol{K}$. The usual rules of classical logic then underwrite the theorem.

Theorem 4.2.3

$$deK_H \circ outer = 1_{\mathrm{op}(\boldsymbol{H})};$$
$$deK_L \circ inner = 1_{\mathrm{op}(\boldsymbol{L})}.$$

This theorem says that $outer$ and $inner$ are reversible using the projections from $\mathrm{op}(\boldsymbol{K})$. A consequence of this is that $\mathrm{op}(\boldsymbol{K})$ act like a disjoint sum with $outer$ and $inner$ being the injections:

Corollary 4.2.4 *Let $g : \mathrm{op}(\boldsymbol{H}) \to D$ and $g' : \mathrm{op}(\boldsymbol{L}) \to D$, then there is a unique $k : \mathrm{op}(\boldsymbol{K}) \to D$ such that g factors into $k \circ outer$ and g' into $k \circ inner$.*

Theorem 4.2.5

$$(deK_H \circ\ inner \circ \mathsf{u}_L)f = \eta_{\mathsf{S}}();$$
$$(deK_L \circ\ outer \circ \mathsf{u}_H)f = \eta_{\mathsf{S}}().$$

This says that injecting a computation from $\mathrm{op}(\boldsymbol{L})$ into $\mathrm{op}(\boldsymbol{K})$ results in a computation whose $\boldsymbol{H}$ component is the identity computation in $\mathrm{op}(\boldsymbol{H})$, i.e., $\eta_{\mathsf{S}}() = \mathsf{u}_H\ 1_{Tok(\boldsymbol{Hi}_1)}$. A similar statement holds for $\boldsymbol{H}$. Notice that $1_{\mathrm{op}(\boldsymbol{H})} : \mathrm{op}(\boldsymbol{H}) \to \mathrm{op}(\boldsymbol{H})$ while $\eta_{\mathsf{S}}() \in \mathrm{op}(\boldsymbol{H})$.

Theorem 4.2.6

$$\beta \in \boldsymbol{K} \text{ implies}$$
$$(\mathrm{op} \circ \pi_{\boldsymbol{H}}(\beta) \neq \eta_S() \text{ and } \mathrm{op} \circ \pi_{\boldsymbol{L}}(\beta) = \eta_S()) \text{ or}$$
$$(\mathrm{op} \circ \pi_{\boldsymbol{H}}(\beta) = \eta_S() \text{ and } \mathrm{op} \circ \pi_{\boldsymbol{L}}(\beta) \neq \eta_S())$$

This theorem is an easy consequence of $outer$ and $inner$ acting like injections to the parameterized disjoint sum $\mathrm{op}(\boldsymbol{K})$ and the projections $\pi_{\boldsymbol{H}}$ and $\pi_{\boldsymbol{L}}$ revealing the components of the elements injected.

Corollary 4.2.7 *For all tokens $\beta \in \boldsymbol{K}$, β satisfying*

$$\Phi_{\boldsymbol{X}} \wedge \boldsymbol{H}(D = V) \wedge \boldsymbol{L}(D = U) \Vdash_{\boldsymbol{K}}$$
$$\boldsymbol{H}(\mathfrak{a}(D = V)) \wedge \boldsymbol{L}(\mathfrak{b}(D = U))$$

implies either $\boldsymbol{X} = \boldsymbol{H}$ and $\mathfrak{b} = 1_{\boldsymbol{Lo}}$, or $\boldsymbol{X} = \boldsymbol{L}$ and $\mathfrak{a} = 1_{\boldsymbol{Hi}}$.

Incidentally, the sequents in the conclusion of the theorem can be seen in model theoretic form in [18] although they were developed independently from the structure of the classifications involved.

4.3 The Classification $\boldsymbol{R}$

The computations $R()$ are on a bijective correspondence with lists of operations from $K()$. The injection $step : K \to R$ with the definition:

$$step\ \ x \overset{def}{=} \rho(x \star_{\mathsf{K}} (\eta_{\mathsf{K}} \circ \delta))$$

is not fundamentally changing the computation x but merely adding some bookkeeping. The following theorem shows that $step$ is reversible:

Theorem 4.3.1 $run \circ step = 1_{S()}$.

Let the infomorphism from $\boldsymbol{K}$ to $\boldsymbol{R}$ be the identity on types. Tokens of $\boldsymbol{R}$ are of the finite and infinite sequences $\langle \sigma_0, \ldots \sigma_{n-1} \rangle, 0 \leq n \leq \infty$ such that each σ_i has the form $\langle step\ \ f, \langle s, q, \langle\langle (), s' \rangle, q' \rangle\rangle\rangle$ for $\langle f, \langle s, q, \langle\langle (), s' \rangle, q' \rangle\rangle\rangle$ a token $\boldsymbol{K}$. Also, it is required for

$$\sigma_i = \langle f, \langle s, q, \langle\langle (), s' \rangle, q' \rangle\rangle\rangle, \quad \text{and}$$
$$\sigma_{i+1} = \langle f', \langle \hat{s}, \hat{q}, \langle\langle (), \hat{s}' \rangle, \hat{q}' \rangle\rangle\rangle,$$

that

(i) $s' = \hat{s}$ and $q' = \hat{q}$ and
(ii) $s(D) = s(V)$ and $q(D) = q(U)$.

(i) allows for only valid computation sequences to appear in $Tok(\boldsymbol{R})$. (ii) allows us to disregard states for which any other values will cause the sequents

$$\Phi_{\boldsymbol{X}} \wedge \boldsymbol{H}(D = V) \wedge \boldsymbol{L}(D = U) \Vdash_{\boldsymbol{K}}$$
$$\boldsymbol{H}(\mathfrak{a}(D = V)) \wedge \boldsymbol{L}(\mathfrak{b}(D = U))$$

to hold spuriously by making the antecedent false.

The function $step$ can be extended to work on $Tok(\boldsymbol{K})$ by

$$step\langle f, \langle s, q, \langle\langle (), s' \rangle, q' \rangle\rangle\rangle \overset{def}{=} \langle step\ \ f, \langle s, q, \langle\langle (), s' \rangle, q' \rangle\rangle\rangle$$

For any tokens $\gamma \in Tok(\boldsymbol{R})$ such that $step\ \ \beta = \gamma$, we let $\gamma \models_{\boldsymbol{R}} Q$ iff $\beta \models_{\boldsymbol{K}} Q$. This will not cause any new propositions to hold in $\boldsymbol{R}$. Now let $\sigma \models_{\boldsymbol{R}} Q$ iff for all $i, \sigma_i \models_{\boldsymbol{R}} Q$. run can be extended and the function run_i defined thusly:

$$run\langle step\ \ f, \langle s, q, \langle\langle (), s' \rangle, q' \rangle\rangle\rangle \overset{def}{=} \langle f, \langle s, q, \langle\langle (), s' \rangle, q' \rangle\rangle\rangle,$$
$$run_i\langle \sigma_0, \ldots \sigma_{n-1} \rangle = run\ \sigma_i, \quad i < n.$$

Theorem 4.3.2 *$\langle step, run_i \rangle$ is an infomorphism for all i.*

The following Lemma says that for any token $\sigma \in \boldsymbol{R}$ is composed of a sequence of actions from the High or Low side and that each High action is paired with the identity for the Low side and visa versa:

Theorem 4.3.3 *For all tokens $\sigma \in \boldsymbol{R}$, for all i, σ_i satisfies either* $\mathrm{op} \circ \pi_{\boldsymbol{H}} \circ run(\sigma_i) \neq \eta_{\mathsf{S}}$, $\mathrm{op} \circ \pi_{\boldsymbol{L}} \circ run(\sigma_i) = \eta_{\mathsf{S}}$, *or* $\mathrm{op} \circ \pi_{\boldsymbol{L}} \circ run(\sigma_i) \neq \eta_{\mathsf{S}}$, $\mathrm{op} \circ \pi_{\boldsymbol{H}} \circ run(\sigma_i) = \eta_{\mathsf{S}}$.

The following theorem is valid by pushing the analogous sequent in $\boldsymbol{K}$ forward along the infomorphism from $\boldsymbol{K}$ to $\boldsymbol{R}$ and the previous Lemma.

Corollary 4.3.4 (Separation) *For all tokens $\sigma \in \boldsymbol{R}$, for all i, σ_i satisfies*

$$\Phi_{\boldsymbol{X}} \wedge \boldsymbol{H}(D = V) \wedge \boldsymbol{L}(D = U) \Vdash_{\boldsymbol{R}}$$
$$\boldsymbol{H}(\mathfrak{a}(D = V)) \wedge \boldsymbol{L}(\mathfrak{b}(D = U))$$

and either $\boldsymbol{X} = \boldsymbol{H}$ and $\mathfrak{b} = 1_{\boldsymbol{Lo}}$, or $\boldsymbol{X} = \boldsymbol{L}$ and $\mathfrak{a} = 1_{\boldsymbol{Hi}}$.

All that is left to to define that the sequents in the theorem hold of all tokens $\sigma \in \boldsymbol{R}$ just when it holds for all σ_i. These sequents are a second-order invariants over all tokens of $\boldsymbol{R}$; the tokens represent all valid computation sequences augmented with some valuation information for logic formulas. It is by accident that these sequents are satisfied by all the $\sigma \in Tok(\boldsymbol{R})$ as opposed to the individual σ_i. This happened by forcing the condition (ii) in the specification of $Tok(\boldsymbol{R})$. (ii) is necessary to define the infomorphism from $\boldsymbol{K}$ to $\boldsymbol{R}$.

Generalizing from Isolation to Noninterference.

Separation in this paper might more accurately be termed "isolation" where High and Low have no interference with each other. It is easy to change this for High having complete access to its D and low's D and Low having only access to its D. Put quickly, one half of the logical work disappears as there would then be counter-examples to theorems implying High has no access to low's D.

The relation $\approx_L$ of Background can be defined on $Tok(\boldsymbol{R})$ by using the projections taking $Tok(\boldsymbol{R})$ to $Tok(\boldsymbol{K})$ to $Tok(\boldsymbol{L})$ and then projecting out the operation using op. The fact that the high and low variables D are used in this paper is immaterial; they could be replaced by streams of values read from an outside environment. Allowing the k index of $\boldsymbol{Hi}_k$ and $\boldsymbol{Lo}_k$ to be larger provides for more operations than the simple increment and decrement.

5. Conclusion

The channel theory used in the proof of separation was able to track the monadic construction of the system. We feel this is an important organizing principle akin to a natural deduction system. This allows channel theory to distribute the workload of the proof over its classification structure much like a natural deduction system allows one to distribute the workload into subproofs. Each classification was relatively simple. The token sets from $\boldsymbol{H}, \boldsymbol{L}$ and above were extracted from the monad applications. The token sets for $\boldsymbol{Hi}_k, \boldsymbol{Lo}_k$ and below were taken from some pre-monadic and standard Floyd-Hoare soundness conditions. The central driving force for the simple logic statement came from $\boldsymbol{C}_H$ and the recognition that the statement of separation could use a second-order free-variable logic sequent. This sequent arose by using a co-channel to represent the common abstraction leading to the second order sequent.

The choice of second-order free-variable logic simplifies the exposition of separation as a meta-statement about the system. The implicit quantification inherent in any free-variable formal system becomes tamed by the use of channel theory to bound the quantification to operations used in the system.

The system analyzed could have been made more complex without substantial changes to either the overall proof or the way the information flows in the System Classification diagram. There are two flows of interest here, or rather one flow and a lack of flow. Information flows from bottom to top in that diagram. It is a logical flow *about* the system. The lack of flow between $\boldsymbol{H}$ and $\boldsymbol{L}$ is really the meta-statement about the system that is the essence of separation. This was a lack a flow of information *in* the system. Complicating the system with shared kernel variables (or processor registers) would be confined to $\boldsymbol{K}$. Once the basic separation properties could be proven there, they would immediately flow to $\boldsymbol{R}$.

A planned extension is probability analysis. The kernel computations in K might include "leakage". This will be modeled in $\boldsymbol{K}$ using sequents that are not entirely valid, but only partially valid. The required change for channel theory is to include this notion via non-normal tokens in $Tok(\boldsymbol{K})$ which are counter-examples to the theorem expressing non-information flow from $\boldsymbol{H}$ to $\boldsymbol{L}$. Probabilities come either defined mathematically or empirically via tests. If $\boldsymbol{K}$ can be entirely mathematically defined, the probabilities are computed ahead of time. This would allow different design decisions to be made. If the probabilities are empirically determined, say if the system must perform a lot of action with an environment giving rise to the leakage, then steps can be taken for redesign of the system.

Another area of extension is in hardware-software codesign. With a mathematically powerful tool such as monads, the interaction between hardware and software can be abstracted. Channel theory will provide the logical layer for formally proving security properties about such designs. The goal is modular designs where proofs of system properties are similarly modularized by following the monadic structure of the system.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proceedings of the Twenty-sixth ACM Symposium on Principles of Programming Languages*, pages 147–160, January 1999.

[2] G. Allwein A Qualitative Framework for Shannon Information Theories. In *Proceedings of the New Security Paradigms Workshop, 2004*, ACM Press, 2005.

[3] J. Barwise and J. Seligman. *Information Flow: The Logic of Distributed Systems*. Cambridge University Press, 1997. Cambridge Tracts in Theor. Comp. Sci. 44.

[4] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(2), Mar. 2005.

[5] E. W. Dijkstra. My recollections of operating system design. *SIGOPS Oper. Syst. Rev.*, 39(2):4–40, 2005.

[6] J. A. Goguen. Institutions: Abstract model theory for specification and programming. *CLSI Research Reports*, 85-30:1–73, 1985.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy (SSP '82)*, pages 11–20. IEEE Computer Society Press, Apr. 1990.

[8] W. Harrison. The essence of multitasking. In *11th International Conference on Algebraic Methodology and Software Technology (AMAST 2006)*, pages 158–172, July 2006.

[9] W. Harrison. Proof abstraction for imperative languages. In *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS06)*, pages 97–113, 2006.

[10] W. Harrison and J. Hook. Achieving information flow security through precise control of effects. In *18th IEEE Computer Security Foundations Workshop (CSFW05)*, pages 16–30, Aix-en-Provence, France, June 2005.
[11] W. Harrison and J. Hook. Achieving information flow security through monadic control of effects. Invited submission to: *Journal of Computer Security*, 2008. 46 pages. Accepted for Publication.
[12] S. Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, 1998.
[13] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990.
[14] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
[15] S. Peyton Jones, editor. *Haskell 98 Language and Libraries, Revised Report*. Cambridge Univ. Press, Apr. 2003.
[16] A. M. Pitts. Evaluation logic. In *Proc. of the IVth Higher Order Workshop*, pages 162–189, 1990.
[17] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, 2002.
[18] J. Rushby. Proof of separability: A verification technique for a class of security kernels. In *Proceedings of the 5^{th} International Symposium on Programming*, pages 352–362, Berlin, 1982. Springer-Verlag.
[19] L. Schäder and T. Mossakowski. *Monad-independent Hoare Logic in HasCasl*, volume Fundamental Approaches to Software Engineering, LNCS 2621, pages 261–277. Springer–Verlag, 2003.
[20] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 1948.
[21] S. Zdancewic. Challenges for information-flow security. In *Proceedings of the First International Workshop on Programming Language Interference and Dependence (PLID'04)*, 2004.

"It Takes a Village" (to create a science): From crypto science to security science

S. A. Borbash, W. B. Martin, R. V. Meushaw
Information Assurance Research, National Security Agency
Fort Meade, MD, USA

Abstract - *Government funding for the solutions to security-related problems has increased significantly in the last decade, along with interest in these problems from researchers from many fields. Much of the funding has been in near-term applications to specific systems (health care records; automotive security; remote controls; military applications, etc.), while there has not been enough in support of more general solutions. Many people have agreed recently that it would be useful to incentivize more theory or "basic science" to support security. This presentation will review current government efforts to build stronger scientific foundations for cyber security.*

Keywords: security, science, theory

1 Introduction

U. S. Government funding for the solutions to security-related problems has increased significantly in the last decade, along with interest in these problems from researchers from many fields. Much of the funding has been in near-term applications to specific systems (health care records; automotive security; remote controls; military applications; etc.), while there has not been enough in support of more general solutions. Many people have agreed recently that it would be useful to incentivize more theory or "basic science" to support security. This presentation will review current government efforts to build stronger scientific foundations for cyber security.

The "science of security" may be defined in different ways, and certainly there are subtopics of security that have scientific foundations, such as cryptography. However, cryptography is inadequate to address all the current security problems. Many have argued that a science of security, since it is concerned ultimately with people, will have to address economic and usability issues. Therefore some effort has been needed to assemble the research community to better define what to pursue to improve security science. These have included both an NSA, NSF and IARPA sponsored workshop in 2008 in Berkeley to bring together researchers from a broad array of fields to catalyze new thinking on security at a more fundamental level, and "Science of Cyber-Security," an overview report on the problem of undergirding current practice with science, produced by the JASON group of the Defense Department in November 2010.

There are also funds available to seed promising research. For example, "Science of Cyber Security" is a new Air Force Office of Scientific Research MURI, and NSA has provided funding to the NSF TRUST program to emphasize security science. In the longer term the federal Comprehensive National Cybersecurity Initiative is expected to be a source of more funds.

NSA is working to help coordinate these research efforts. One effort is to provide a collaboration environment where people can participate remotely in program review meetings, view current problem descriptions, and find relevant research publications. This "virtual organization" of security science researchers is currently being created as a website with the NSF Cyber Physical Systems program. Part of the goal of this site will be to connect many of the science-of-security-related research programs run within the federal government at this time. This should make it easier for all researchers, and all program managers, to see what the others are doing.

2 References

[1] NSF/IARPA/NSA Workshop on the Science of Security, Berkeley CA Nov. 2008. Retrieve from sos.cs.virginia.edu

[2] "Science of Cyber-Security," JASON report JSR-10-102. Retrieve from www.fas.org/irp/agency/dod/jason/cyber.pdf. Nov. 2010.

[3] NSF TRUST Science and Technology Center. Visit www.truststc.org

[4] Science of Cyber Security, AFOSR FY2011 MURI Topic #16. ONR Broad Agency Announcement 10-026, p.51

[5] Comprehensive National Cybersecurity Initiative. Visit www.whitehouse.gov/cybersecurity/comprehensive-national-cybersecurity-initiative

SESSION

INVITED SESSION - RC + LBS: THE CONFLUENCE OF SECURE HARDWARE AND PROGRAMMING LANGUAGES

Chair(s)

PROF. WILLIAM L. HARRISON

3-D Extensions for Trustworthy Systems

(Invited Paper)

Ted Huffmire*, Timothy Levin*, Cynthia Irvine*, Ryan Kastner† and Timothy Sherwood‡
*Department of Computer Science
Naval Postgraduate School, Monterey, CA 93943
Email: {tdhuffmi,levin,irvine}@nps.edu
†Department of Computer Science and Engineering
University of California, San Diego, La Jolla, CA 92903
Email: kastner@cs.ucsd.edu
‡Department of Computer Science
University of California, Santa Barbara, Santa Barbara, CA 93106
Email: sherwood@cs.ucsb.edu

Abstract

Trustworthy system development entails a high non-recurring engineering (NRE) cost together with a low volume of units over which to amortize that cost. For example, the potential for developmental and operational attacks against hardware requires countermeasures that make it very expensive to design and manufacture custom hardware used to build high assurance systems. To address these problems, we propose an approach to trustworthy system development based on 3-D integration, an emerging chip fabrication technique in which two or more integrated circuit dies are fabricated individually and then combined into a single stack using vertical conductive posts. With 3-D integration, a general-purpose die, or computation plane, can be combined with a special-purpose die, or control plane. We discuss the security advantages of using 3-D integrated hardware in sensitive applications, where security is of the utmost importance, and we outline problems, challenges, attacks, solutions, and topics for future research.

I. Introduction

Hardware-oriented security is growing in importance as attackers increasingly target the lowest level of system abstraction. 3-D integration is an emerging technology for designing efficient chips by stacking two or more integrated circuit (IC) dies and connecting them with conductive posts. Unlike traditional coprocessors, a 3-D integration design approach offers the ability to monitor and even override internal structures of a processor. For example, on-chip bus traffic can be monitored, and bus connections can be disabled. With these capabilities, 3-D integration can be used to provide secure alternate services (e.g., cryptographic processing at much higher bandwidth than a coprocessor), isolation, and passive monitoring for mass-produced processors.

In our basic paradigm, a 3-D chip consists of one die that is a commodity microprocessor and another die that contains application-specific security functionality; we refer to the commodity die as the *computation plane* and the custom die as the *control plane* (or *resource*[1] plane in situations where resources are simply made available). Figure 1 shows an example system in which multiple CPU cores reside in the computation plane, and routers reside in the control plane. The control plane and computation plane can be fabricated at separate foundries and conjoined in a third facility. When the control plane contains mechanisms that enforce a policy on the computation plane, to achieve a requisite level of trust, the fabrication of the control plane and the conjoining operation can take place in a trusted foundry.

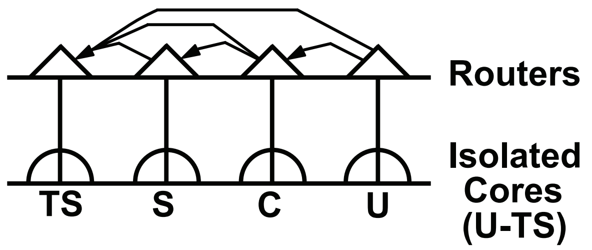


Fig. 1. Isolated cores, surrounded by 2-D moats, and network-on-chip routers. Here, using two IC planes, the routers help to enforce a Multilevel Security (MLS) policy on information flow in the lower plane. In this example, each core has been assigned a label of either TOP SECRET (TS), SECRET (S), CONFIDENTIAL (C), or UNCLASSIFIED (U).

Developing high assurance systems is costly. Our approach has the potential to reduce the cost of developing hardware for high assurance systems by joining a mass-produced computation plane with a custom control plane. Our approach provides several advantages, including (1) dual use of the computation plane, which can be optionally combined with a control plane housing application-specific security functions; (2) physical isolation and logical disentanglement of security functions in the control plane from the non-security circuitry in the computation plane; (3) controlled lineage (e.g., use of a trusted foundry to manufacture the control plane); (4)

[1]Resources are the totality of all active and passive entities on a chip, across a wide range of abstractions. For example, a storage buffer, an accumulator, and a bus are resources.

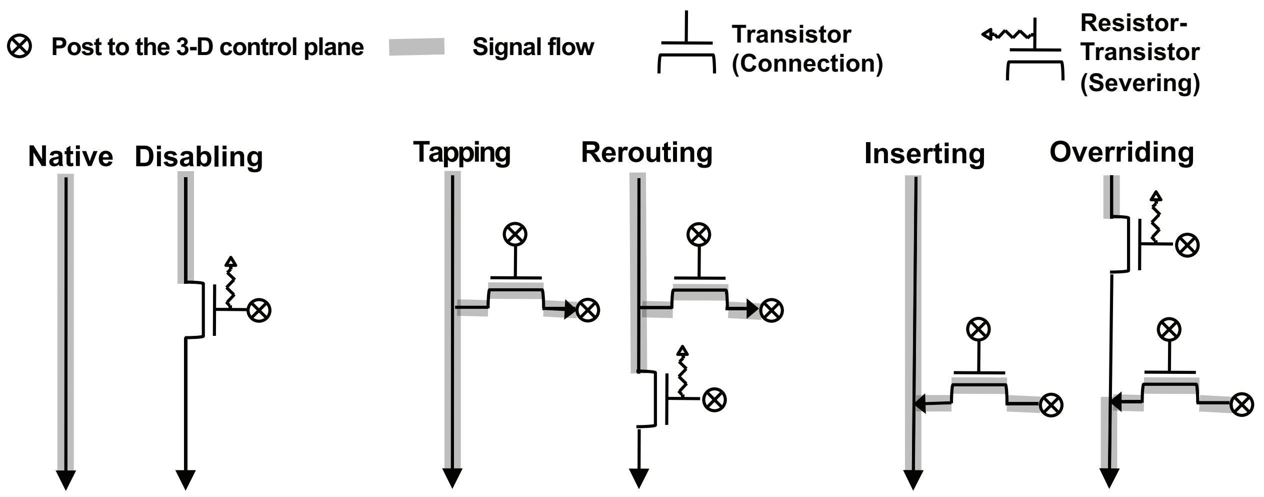


Fig. 2. Circuit-level primitives for trustworthy 3-D design [28]. The *disabling* circuit can stop a signal in the computation plane from flowing, based on the control plane's command, which is sent through a dedicated post. The *tapping* circuit copies a signal from the computation plane to the control plane. Two posts are needed: one to carry the signal to the control plane and another for the command to connect the signal. The *rerouting* circuit combines tapping and disabling so that the original signal only goes to the control plane. The *inserting* circuit carries a signal from the control plane to a circuit on the computation plane. The *overriding* circuit combines inserting and disabling, first disabling the original signal in the computation plane and then introducing a new signal from the control plane.

high bandwidth communication and low latency between the computation plane and components in the control plane such as coprocessors, memory, or other devices; and (5) direct, granular access by the control plane to internal structures in the computation plane.

The threat model we address is that of malicious hardware and software in the computation plane, although for this work we assume that the primitives we introduce on the computation plane remain intact, e.g., in the face of various malicious inclusions and probing of the computation plane.

In this paper, we present concepts and ideas for 3-D design based on a system security architecture that supports a variety of policies including those that specify legal communication between policy equivalence classes[2]. We explore minor modifications to the 3-D design flow to support these methods. We also describe new circuit-level primitives to support this technique and introduce the use of distinct layers available in a 3-D IC as a primitive for the physical isolation of hardware components. Specifically, the contributions of this paper are:

- The application to 3-D IC design of a proven design practice based on a system security architecture.
- A general-purpose circuit-level primitive to support this design approach by allowing different control planes to be conjoined with the same computation plane (or vice-versa).
- A diode circuit-level primitive to support this approach by enforcing the one-way flow of information in a 3-D IC.
- A design approach that uses distinct IC layers, with separate lineage and developmental assurance, to achieve physical separation of hardware components, providing secure application capabilities even in the presence of an untrusted processor and OS software.
- Requirements for automated 3-D IC design tools for the physical layout of components. Since fully automated Electronic Design Automation (EDA) for 3-D circuit design and layout are still evolving, this paper provides high-level requirements analysis aimed at influencing their development so that security is adopted as a principal constraint in the practice of 3-D IC design.
- Offloading of testing circuitry to removable test planes to reduce the cost of design for test.

[2]The system resources are partitioned into separate classes, where each member of a class is treated equivalently with respect to the security policy. Technically, an equivalence class is formed by a set and binary relation. Here, the set is all system resources, and the relation is "has the same policy as."

II. Standardization of Interfaces

The ability to conjoin different control planes with the same computation plane allows a variety of application-specific security enhancements (e.g., policy enforcement mechanisms) while reusing a mass-produced computation plane. To accomplish this, however, requires a standard interface to the computation plane. A standardized interface also enables a mass-produced control plane (e.g., a 3-D crypto coprocessor) to be conjoined to a variety of computation planes. Achieving standardization requires overcoming several challenges:

- Standard placement of posts
- Diverse manufacturing processes (e.g., face-to-face vs. face-to-back bonding)
- Diverse electrical and timing properties of dies
- Diverse sizes and form factors of dies
- Diverse packaging options for 3D-ICs
- Standardization places constraints on 3-D floor planning and layout of TSVs

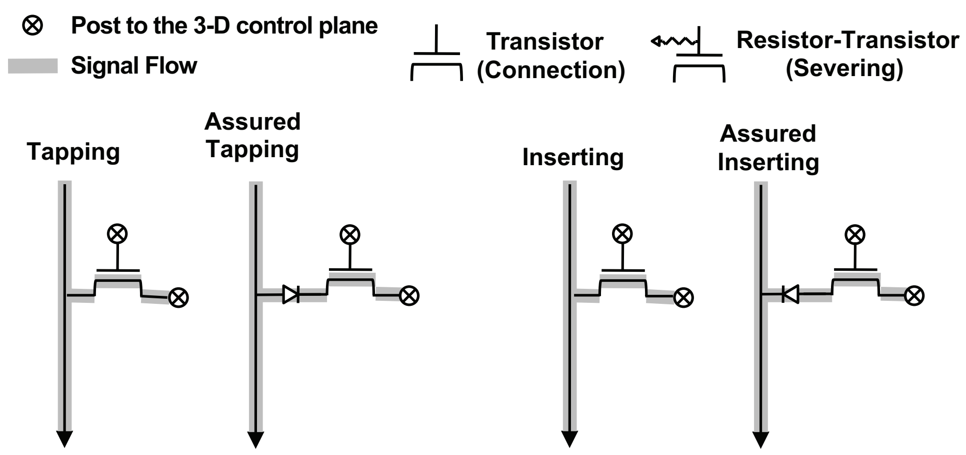


Fig. 3. Primitive TSV functions and corresponding selectively refined functions. The diode is placed between the native (computation plane) circuit and the TSV receptacle, such that the diode controls the flow regardless of how the TSV receptacle is used by the control plane.

The basic idea is the identification of a standard set of logical TSV receptacles on the computation plane, each related to a basic function (e.g., tapping the internal bus, tapping the instruction pipeline, etc.). As long as the basic set of TSV receptacles is present, then variance in their physical layout from processor type to processor type can be accommodated in simple realignment (e.g., with a hardware adaption layer) of the control plane. Furthermore, reusing the precise physical layout of either a control plane or a computation plane is both more challenging and less interesting than reusing the higher-level design of a custom plane.

A. Standard Primitives

In this section we introduce two novel circuit-level primitives, a *diode* and a *general-purpose TSV receptacle*, that support our design approach. Previous work introduced five primitive functions for controlling information flow between planes: disabling, tapping, rerouting, inserting, and overriding, as shown in Figure 2. First, we introduce the diode and two assured functions that result from its application.

1) Diode: A diode, as shown in Figure 3, allows information to flow in only one direction from one component to another[3]. Such an arrangement can enforce a policy requiring that information can flow from a low confidentiality component to a high confidentiality component[4] but not vice-versa. In other words, in a system whose resources have been partitioned into policy equivalence classes, a diode can be applied to a primitive to enforce the one-way flow of information between these classes. Diodes provide a granularity of enforcement related to the granularity of components that they connect. Diodes can be static or programmable, and they can ensure that vertical posts do not violate the inter-plane policy, e.g., by placing diodes between a post and the circuitry of a plane. We expect, however, that this hard-wired enforcement can have unforeseen effects, e.g., on the performance of bidirectional communication protocols, and that *programmable diodes* would provide flexibility in the designs supported.

The primitives each provide an environment for receiving one or two posts. We refer to this computation plane environment as a *TSV receptacle/socket*.

2) Generic TSV Receptacle: A general-purpose TSV receptacle can be used to support multiple control plane applications with the same computation plane, e.g., when different control planes are used or when the applications on a given control plane are reconfigured. These generic TSV receptacles are used to anchor posts on the computation plane to minimize the number of TSV receptacle types that the processor manufacturer must produce and allows a given TSV receptacle to be used for different purposes from application to appplication. Supporting these features requires identifying standard locations for posts on the computation plane such that generality is balanced against the hardware resources (e.g., posts) required (see future work).

Figure 4 shows a general-purpose TSV receptacle that supports any of the five basic circuit-level primitives. To make use of a TSV receptacle, signals on the control plane determine which primitive is active at a given point in time. Thus, different 3-D applications can use the same TSV receptacle in a different way. Only one configuration of the Generic TSV Receptacle is required even though, for example, one application might read from a TSV receptacle, and another might override circuits with it.

A generic TSV receptacle provides design flexibility at the cost of additional circuitry and posts. The Generic TSV Receptacle accepts four posts (three control and one data) implementing four primitive features internally: one tapping or insertion feature and two disabling features. These are combined to implement disabling (i), tapping (ii and iii), rerouting (ii, iii, and iv), inserting (ii and iii), overriding (i, ii, and iii), and native (none of the posts), which has no effect.

[3]Note that one-way communication can be enforced using other electrical techniques besides a diode

[4]Given a lattice of confidentiality markings, "high" markings are those that are closer to the top (the universal upper bound), and "low" markings are those that are closer to the bottom (the universal lower bound).

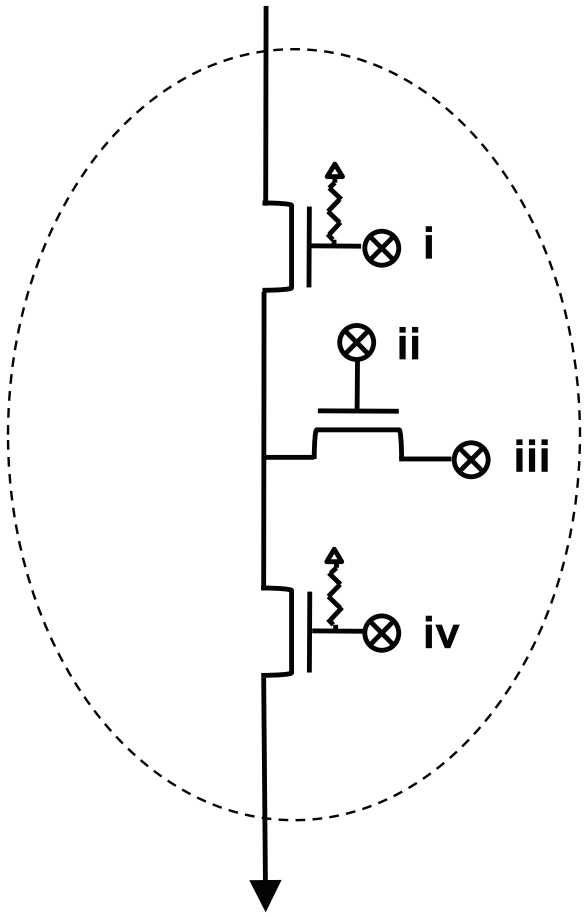


Fig. 4. Generic TSV Receptacle. A general-purpose TSV receptacle supports any of the basic circuit-level primitives (disabling, tapping, rerouting, inserting, and overriding). Signals from the control plane determine which primitive is active at a given point in time. Thus, the application on each different control plane could use a given TSV receptacle in a different way: one application might tap a TSV receptacle, and another might override it. The control posts are i and ii, and the data posts are iii and iv.

The Generic TSV Receptacle could include *diodes* to assure that the information flows of each post are precisely controlled, in which case the diode for post iii could be programmable to support reading or writing.

3) Application Classes: This section describes several categories of 3-D applications that can be built using our framework [28], [8]:

- Secure Alternate Service. This category provides a trustworthy enhancement to the service provided by the computation plane. Examples include ciphers, key storage, compression, and network-on-chip (NoC) routers.
- Isolation and Protection. This category actively overrides the computation plane to enforce access control, eliminate points of interference, or disable communication. Examples include the 3-D cache eviction monitor described in [28] and enforcing a policy on buses or NoC routers in the computation plane, as shown in Figure 6.
- Passive Monitoring. This category passively monitors the computation plane. Examples include audit, information flow tracking, and runtime checks.

III. Isolation of Hardware Equivalence Classes

Arranging system components to structurally support a security policy results in a *security architecture* [14]. Realization of a multi-level security (MLS) policy in a 3-D system requires establishing (1) policy equivalence classes; (2) isolation of components according to those classes; and (3) controlled interaction between classes according to an inter-class communication policy.

4) Policy Equivalence Classes: Grouping similar entities into a domain or *equivalence class* helps simplify the design of secure systems.

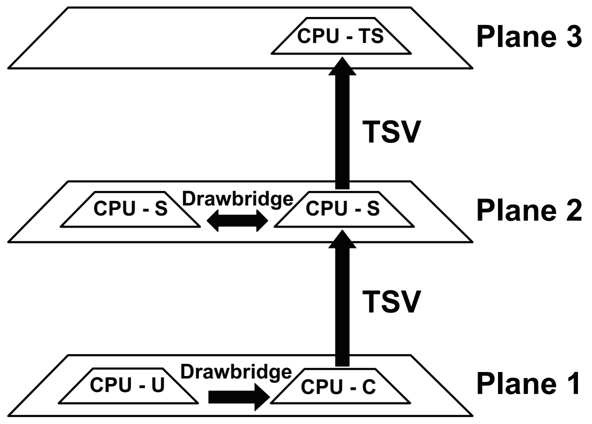


Fig. 5. A hypothetical layout of CPUs on several computation planes, where each CPU is a point in the lattice. This system consists of three layers: the lower layer contains a CPU with an UNCLASSIFIED (U) label and a CPU with a CONFIDENTIAL (C) label; the middle layer contains two CPUs with a SECRET (S) label; and the upper layer contains a CPU with a TOP SECRET (TS) label.

Consider a system security policy for a 3-D IC in which the computation plane contains several cores: a given application workload may require that two cores devoted to a sensitive application be isolated from the rest. To achieve isolation, computational components belonging to the same equivalence class can be placed on the same layer (die). On the other hand, multiple equivalence classes may reside on a layer (see the lower plane in Figure 5) if they are spatially or otherwise separated. Thus, the dies and cores can be partially ordered with strong separation, as shown in Figure 5. To achieve controlled sharing, only specified inter-die posts are permitted, and the 3-D design is statically checked to ensure that posts do not violate the policy. In other words, the physical separation between layers and between cores on a layer result in a conceptual moat, and connections that cross moat boundaries (viz., *drawbridges*), must conform to the policy, as was shown to be effective in [7]. For generality, the diodes and junctions in Figure 5 would be programmable, to support a wide range of policies.

3-D integration offers the unique capability to physically isolate hardware components by arranging circuitry into distinct layers. We extend the idea of a 2-D moat into 3-D,

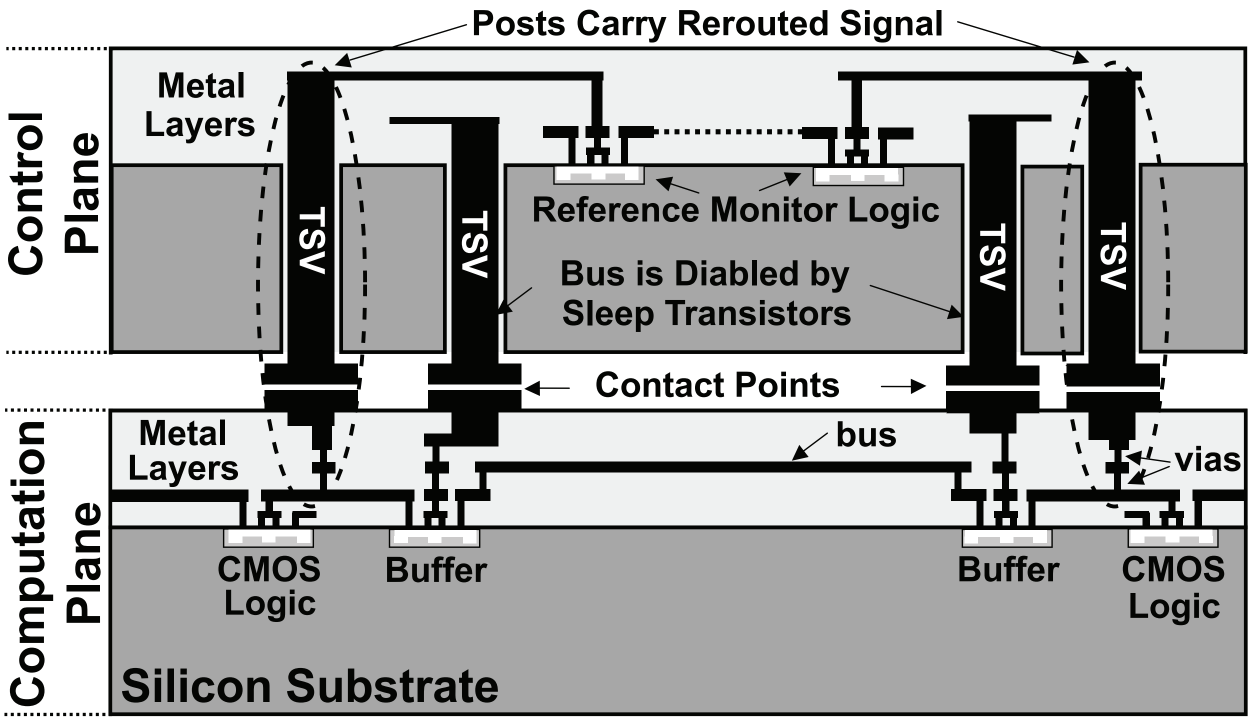


Fig. 6. Example of rerouting bus signals in the computation plane through the control plane [28].

in which physical isolation possible with separate layers is similar to a 2-D moat.

We can use a combination of moats and 3D planes to isolate the equivalence classes. *Moats* separate CPUs on a given plane, and each plane is physically isolated from other planes. *Drawbridges* connect different moats on the same plane.

The isolation need not be complete. The standard MLS policy for information flow allows a "downward" flow in the lattice. Diodes and other electrical mechanisms can enforce such a one-way flow and we can use automated analysis techniques such as information flow tracking [27], [25] to verify that the flows conform to the policy.

To facilitate the controlled interaction of isolated layers, it is necessary to ensure that only specified vertical connections exist between these layers, where the specification includes directionality of information allowed between layers consistent with the MLS sensitivity of the layers. We also extend the idea of a 2-D drawbridge into 3-D, in which the architectural integrity of vertical connections between layers must be statically checked, similar to a 2-D drawbridge.

Furthermore, it is possible to use 2-D moats and drawbridges within an individual layer, when design requirements (e.g., cost limitations) dictate that more than one equivalence class must reside on a layer. In order to separate the cores residing on a given chip, various structures must be partitioned and/or virtualized, including on-chip memory and computational components. Other *points of interference* between cores that must be constrained include interconnect (e.g., on-chip bus or on-chip network), I/O devices, and dependencies (e.g., power, I/O, privilege, etc.). Other structures (e.g., micro-connections not visible at a high level of system abstraction) may also need to be partitioned.

5) Core Isolation: Isolation is a foundational concept in computer security [24]. At the hardware level, it is possible to isolate circuitry spatially. Hardware functionality can be physically isolated by using air gaps or other techniques described as moats and drawbridges [7], [9]. To facilitate the controlled interaction between isolated equivalence classes, it is necessary to ensure that only specified connections exist between these domains. In the case where system components communicate over a shared bus (or on-chip network), the interconnect must be designed to prevent illegal information flow between equivalence classes, e.g., using diodes or similar arrangement to control the direction of flow.

6) Inter-Class Communication Policy: At the highest level of abstraction, an inter-class communication policy for a 3-D system must first specify what the equivalence classes are and what information flows are permissible between the classes. Then, the system resources should be partitioned with respect to the equivalence classes. Figure 5 shows an example of a lattice policy. Barring the use of a time sharing approach, the policy should also specify which equivalence class each core is assigned. A given equivalence class may span multiple dies and even off-chip I/O and memory traffic. This high-level policy must be mapped to enforcement mechanisms at a lower level of abstraction, either by a talented human designer or by automated tools that assist the designer in visualizing the 3-D security architecture and exploring the design space of

3-D mechanisms. Finally, the flows allowed between classes are chosen, providing a partial order relation required of the lattice. For example, in an MLS policy, we order the classes (TS $\geq$ S $\geq$ C $\geq$ U) and allow a flow from class A to class B only if B $\geq$ A [3].

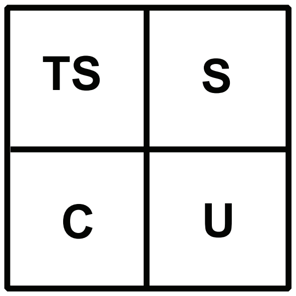


Fig. 7. Top view of a hypothetical multi-core integrated circuit. Four regions of the chip correspond to the hierarchical General Service (GENSER) equivalence classes of TOP SECRET (TS), SECRET (S), CONFIDENTIAL (C), and UNCLASSIFIED (U).

Figures 7 and 8 show examples of more complex 3-D security architectures. The system shown in Figure 8 obeys an MLS policy, as described above. Diodes enforce the one-way flow of information. Junctions, which are two-way connections, connect memory and cores with equal labels. Programmable junctions can be set to act as a diode for which the layer reads the TSV, a diode for which the TSV reads the layer, or a two-way connection.

To avoid the security issues of sharing a CPU between equivalence classes (e.g., side channels and other covert channels) we dedicate each CPU to an equivalence class in our lattice-based policy, and then ensure that unwanted interference between CPUs is impossible.

Included in the equivalence class of a given CPU are its dedicated memory (L1) and any dedicated board-resident devices. L2 memory may be shared with another CPU only if cache side channel interference has been eliminated, effectively virtualizing the L2 cache [29], as shown in Figures 9 and 10.

7) 3-D Design Flow: As the standards for 3-D Electronic Design Automation (EDA) tools are still emerging, a complete standard design flow is not yet available, as described in [1], [21], and [15]. Existing design flows are limited to a specific 3-D fabrication technology and a fixed number of planes, and there is no standard fabrication technology. Therefore, talented human designers must balance multiple constraints simultaneously to achieve the desired design properties. In a 3-D design flow, the positioning of components on different layers may perturb the properties achieved on the individual layers, e.g., vertical proximity may increase thermal factors.

3-D design is very challenging because the large number of interacting constraints result in a complex optimization problem. Specifically, the designer must balance several factors in achieving overall properties such as performance, bandwidth, yield, cost, and testability:

- thermal limits, power density, and electrical interference

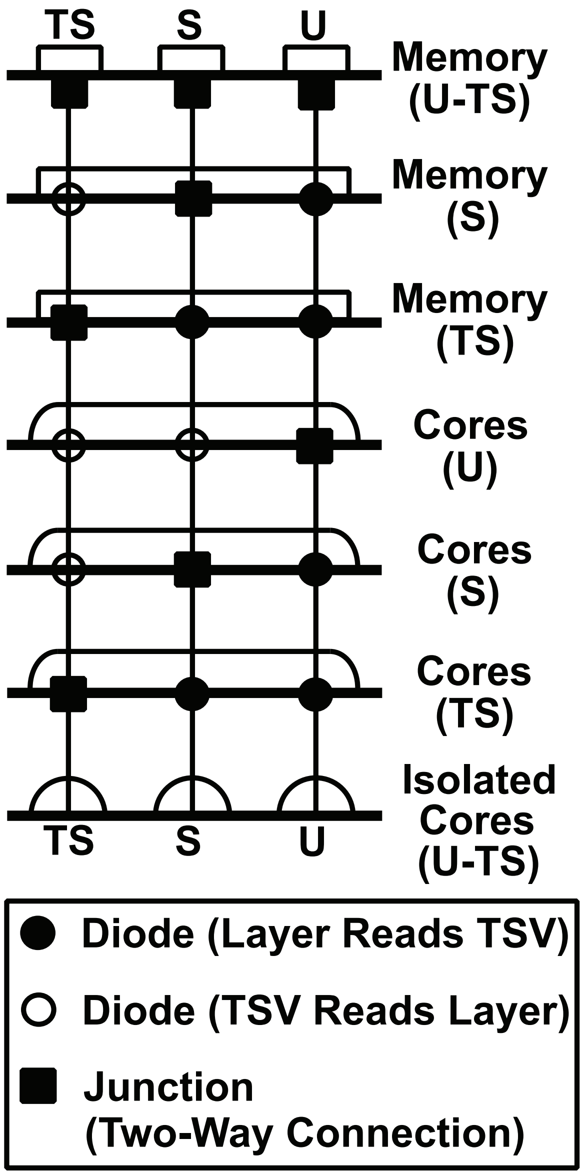


Fig. 8. Side view of a hypothetical 3-D integrated circuit with seven layers.

- power distribution (current delivery) [11] and power limits
- clock delivery, timing analysis, and dissimilar clock rates
- packaging, handling, wafer alignment, and bonding
- mechanical stress [2] and metal resistivity/expansion
- via density, via size, via count, and via location
- the number of vertical connections required and the length of the posts
- variation across planes: dissimilar via resistance and pitch
- I/O, die-to-die signaling, bus fan-in, and bus fan-out

Despite these daunting challenges, numerous 3-D systems have been built (see Section VII: Related Work), including a complex processor with stacked memory fabricated at Georgia Tech [17], [16].

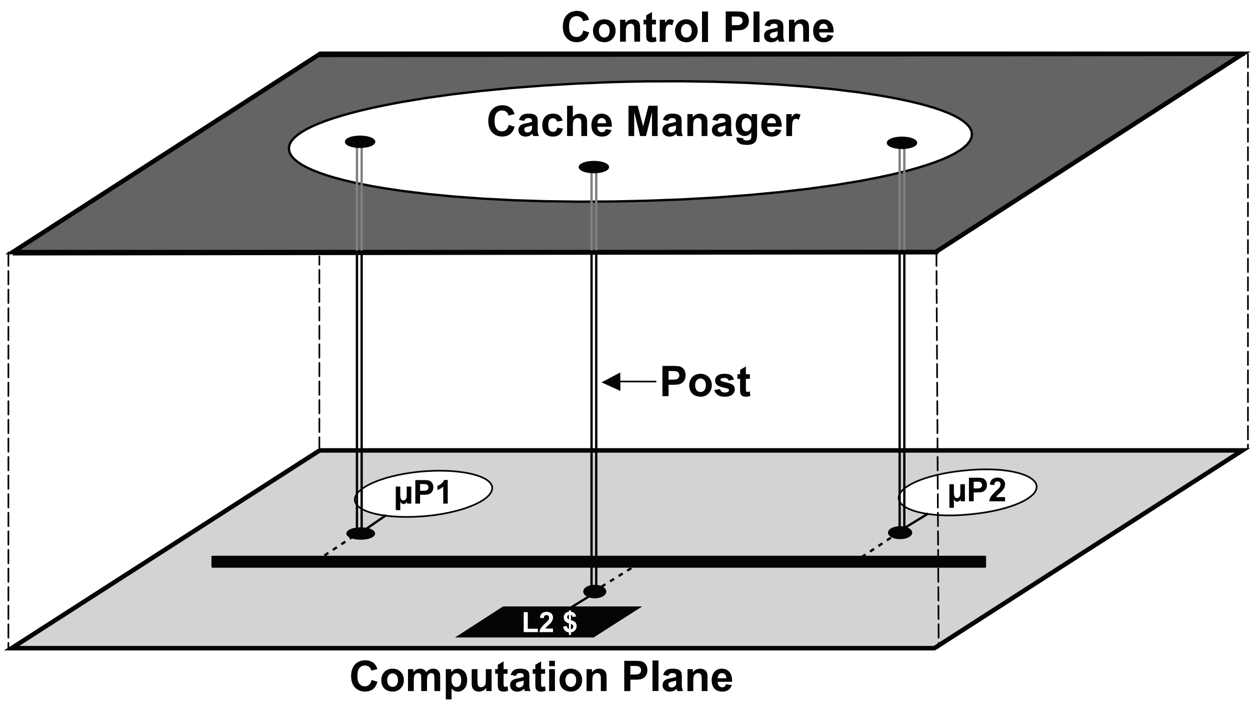


Fig. 9. System-level view of a two-tier 3-D IC. The computation plane contains a dual-core chip multi-processor and shared L2 cache, all connected to a shared bus. The control plane contains a cache manager that partitions the shared L2 cache.

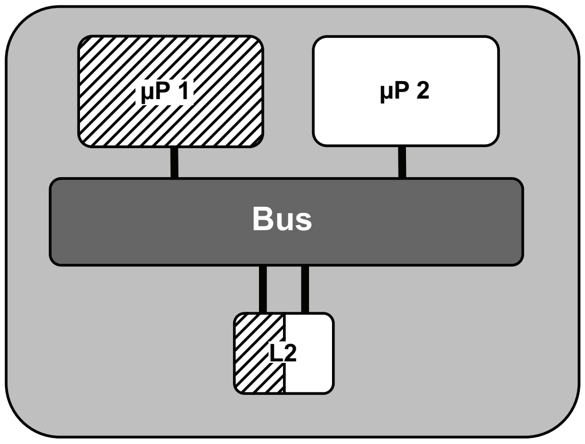


Fig. 10. System-level view of the computation plane of the two-tier 3-D IC shown in Figure 9.

8) Design Flow Modifications: Our technique involves making minor modifications to the 3-D IC design flow, specifically the stage referred to as *floor planning*, which is performed after logic synthesis and the validation of the functional correctness of the circuitry. In other words, the design is initially agnostic about floor planning, yet partitioning of certain resources among layers (to respect a policy-based partial ordering of resources) may be a first-order constraint.

To most effectively use 3-D technology, the designer must consider the ordering and connections of the 3-D security architecture as constraints when performing floor planning. Prior to floor planning, the designer must already have a coarse-grained understanding of (1) what computational resources will reside on which layers; (2) the vertical connections permitted between layers; (3) the transitive closure of flows induced by posts; (4) the horizontal connections permitted between components within a layer; and (5) constraints on 2-D and 3-D interconnect (e.g., bus or router), such as enforcing a Time Division Multiple Access (TDMA) scheme.

Arriving at this coarse-grained picture requires the designer to refine the security architecture. Automated tools can assist the designer in exploring the design space of 3-D security architectures and to validate that the security architecture correctly supports the policy.

A multistep floor planning approach is described in [21], which involves first assigning blocks to planes (i.e., *partitioning*) and then allowing blocks to be rearranged within a plane (i.e., *intra-plane moves*). Allowing simultaneous inter- and intra-plane moves results in an unmanageable optimization problem size. Working within this multistep paradigm, security constrains the first step when, for example, blocks are assigned to planes according to the functional relationship between planes and blocks. Security also constrains the second step when blocks are rearranged within a plane to achieve the specified security properties. For example, intra-plane rearrangement of blocks results in the positioning of a TS memory region in one plane directly above a TS core in another plane and may require the arrangement of air gaps between blocks within a plane. To decrease the complexity of the optimization problem, the design tools could be instructed to only allow the partial ordering of components rather than the more stringent requirement of assigning components to specific planes.

Floor planning establishes the precise geometric boundaries of (1) the computational cores on the layers; (2) the vertical connections between layers; and (3) the horizontal connections within a layer. At this stage, it is necessary to statically check that horizontal connections that cross 2-D moat boundaries and vertical connections that cross layers (both extended with transitive relations) do not violate the system security policy (e.g., checking for extraneous connections between equivalence classes). If these geometric boundaries of elements determined during the layout process are specified in the GDS II database file format, a static analysis program can check the design for policy conformance by analyzing the GDS II file.

A. Other Uses besides MLS

1) Interconnect Built to Support MLS Policy: As discussed earlier, an MLS security policy is based on a *lattice* of sensitivity labels, e.g., TOP SECRET (TS), SECRET (S), CONFIDENTIAL (C), UNCLASSIFIED (U). Each resource controlled by the policy is assigned a label, such that the labels partition the resources. All of the resources with the same label are said to be in the same *equivalence class*.

Lattice-based policies can be generalized beyond those with national security labels, as long as there is a means of determining in which equivalence class a resource resides, and the equivalence classes form a lattice.

2) Self-Protection and Dependency Layering: A secure application must be protected from attack and must not depend on any components that are less trustworthy. In a 3-D system, self-protection requires that the computation plane cannot short-circuit or surge the power of the control plane, make requests to the control plane that cause buffers to overflow, or modify the control plane. Dependency layering requires that the control plane not request service from or wait on the computation plane.

This paper does not address package-level concerns related to I/O and power other than the isolation provided by separate layers within the enclosure of the package. A more thorough discussion of the axioms of self-protection and dependency layering is applied to 3-D IC design in [8].

B. Topology Considerations

The form factor of the dies is a security architecture design consideration. For example, a small memory tile could be stacked on top of one moated region belonging to an equivalence class. Provided that its form factor allows it to remain within the bounds of the moated region on which it is placed, it should be separate from another memory die stacked on top (but within the bounds of) another moated region belonging to a different equivalence class. A wide variety of 3-D structures are possible in this manner.

IV. Power and I/O Independence

A variety of methods for achieving I/O independence for the control plane are possible [8]. First, the control plane can be located closest to the I/O and power pins, such that it provides these services to the computation plane. Otherwise, wireless methods include capacitive/inductive coupling, short-range RF, short-range optical, and simply attaching EEPROM. Wired options include JTAG interface, serial cable, dedicated pins, TDMA over HyperTransport, and dedicated memory ranges. Providing an independent source of power to the control plane can be achieved using the same circuit-level primitives described above for computation signals. Wireless power transmission technology is another option.

V. Offloading of Testing Circuitry

Design for test circuitry consumes significant die area and therefore plays a major role in the cost of mass-produced processors[5], [26]. DFT must not impact performance; must have a small area cost; allow multiple uses when possible; and be integrated into the design from the beginning [5].

While 3D-ICs have their own set of daunting test challenges [13], [30], [19], we argue that significant parts of the DFT circuitry can be offloaded from the computation plane to the control plane to reduce area impact. Furthermore, a 3-D approach can mitigate some security concerns associated with DFT circuitry. For example, Yang et al. showed that crypto processors are vulnerable to attacks that use scan-based design for test to steal crypto secrets [31]. With a 3-D approach, the test interface does not have to ship with production systems; instead, only a select number of computation planes are joined with testing planes to support factory testing.

VI. Experimental Framework

We are in the process of preparing a test bench for experimentally validating and demonstrating the effectiveness of our circuit-level primitives to determine how well they work when fabricated. The 3-D chip will include a test harness which will manage the invocation of and vary the inputs to each circuit-level primitive and will read the outputs to verify that the expected behavior occurs. The experiment will build the primitives in different configurations. The frequency at which a circuit can operate without modification will be compared against: (1) the frequency at which it can operate with the circuit-level modifications; and (2) the frequency at which it an operate with both the circuit-level modifications and the addition of a control plane.

VII. Related Work

While 3-D integration is an emerging technology, a variety of 3-D applications have been realized, including imaging [32], medicine [10], particle physics [6], reconfigurable hardware [23], as well as high-performance microprocessors, as described in [4], [22], [17], [16], and [12],

A 3-D cache monitor designed to mitigate access-driven cache side channel attacks in a simultaneous multithreading processor's shared memory [29] has negligible computation plane overhead in terms of area, delay, and performance [28]. The control plane maintains a data structure that records whether cache lines are protected and for what process. Cache evictions of a protected line are denied based on the security policy. Experiments used an FPGA synthesis tool to collect area and timing information, comparing the case involving just the computation plane, the case involving just the control plane, and the case involving the combined computation and control plane. Evaluation of the impact of the delay of the posts was based on data from Loi et al. [18], which found that the worst-case delay between opposite corners of a chip to be approximately .29ns, which is too small to affect the performance of the critical path of the 3-D cache eviction monitor. Analysis by Mysore et al. showed that the additional area required for vias is small for 3-D applications that perform introspection and profiling of the computation plane [20].

This paper builds on earlier work presented in [28] and [8] that applied 3-D integration to the problem of hardware-oriented security and trust as well as on the *moats and drawbridges* technique developed for Field Programmable Gate Arrays (FPGAs) presented in [7] and [9]. In addition to the 3-D circuit-level primitives presented in [28] and [8], in this paper we discuss an additional circuit-level primitive, the *diode*, and we build upon this diode primitive to support policy enforcement in a 3-D IC. We also introduce a *general-purpose TSV receptacle* that can implement any of the primitives described in earlier work, supported as necessary by the diode primitive introduced in this paper.

VIII. Future Work

The use of disabling posts to selectively disable (by removing power) wires in the computation plane that violate the isolation of components could break some designs, in which case the circuitry could remain unchanged but its use audited by other posts. Determining which connections to disable could utilize analytical tools from graph theory, e.g., determination of *cut points*. 3-D posts connected to these cut points could be used to selectively disable the connections without affecting other portions of the circuit. Ideally, the native computation plane designer eliminates all points of interference between the cores of a multi-core processor. However, some connections are needed by some applications but not others. It is possible that these connections could be selectively disabled according to the policy. We also leave the following to future work:

- The use of a reference monitor for providing fine-grained policy enforcement in 3-D chips.
- The use of disabling posts in the context of dynamic runtime policy changes, e.g., to allow moats that are configurable at runtime, i.e., movable moats.
- The use of configurable diodes in the context of reconfigurable policy changes.
- The use of disabling and inserting posts to modify the behavior of interconnect in the computation plane.
- The use of rerouting and overriding posts to force bus traffic in the computation plane to take a detour to an alternate bus in the control plane where various policies can be enforced.
- Using disabling posts to partition a computation plane consisting of entwined cores that are not already spatially isolated.
- The use of a module in the control plane to erase architectural state in the computation plane in order to address data remanence.
- Identification of standard locations for TSV receptacles on general-purpose processors.

IX. Conclusion

Security must become a first-order design constraint in the engineering of 3-D systems. We have described a design approach based on a 3-D system security architecture. To support this approach, we have described two novel circuit-level primitives: (1) a general-purpose TSV receptacle that allows the same posts to be reused in different ways by different applications and (2) a diode that restricts the vertical flow of information to one direction. Our design approach also takes advantage of the physical isolation provided by distinct layers. Finally, we describe modifications to the floor planning stage of the 3-D design flow that are necessary to support our design approach. We strongly recommend that the 3-D EDA community incorporate features in commercial design tools for the hardware-oriented security and trust community to constrain the floor planning of components in a 3-D IC. Design flows should provide researchers and practitioners the flexibility to, for example, achieve isolation of and programmable partial ordering of selected components.

X. Acknowledgments

This research was funded by National Science Foundation Grants CNS-0910734, CNS-0910389, and CNS-0910581.

References

[1] Global Semiconductor Alliance. Tour guide to 3D-IC design tools and services. In *Presentation at the 3D/TSV Technology Reception: Education, Benefits, and Solutions at the Design Automation Conference (DAC)*, Anaheim, CA, June 2010.

[2] Krit Athikulwongse, Ashutosh Chakraborty, Jae-Seok Yang, David Z. Pan, and Sung Kyu Lim. Stress-driven 3D-IC placement with TSV keep-out zone and regularity study. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, November 2010.

[3] D.E. Bell and L.J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, Hanscom Air Force Base, Bedford, MA, USA, March 1976.

[4] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die stacking (3D) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Orlando, FL, December 2006.

[5] Adrian Carbine and Derek Feltham. Pentium pro processor design for test and debug. *IEEE Design and Test of Computers*, 15(3), July–September 1998.

[6] Marcel Demarteau, Yasuo Arai, Hans-Gunther Moser, and Valero Re. Developments of novel vertically integrated pixel sensors in the high energy physics community. In *IEEE International Conference on 3D System Integration*, San Francisco, CA, September 2009.

[7] Ted Huffmire, Brett Brotherton, Gang Wang, Tim Sherwood, Ryan Kastner, Timothy Levin, Thuy Nguyen, and Cynthia Irvine. Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2007.

[8] Ted Huffmire, Timothy Levin, Michael Bilzor, Cynthia E. Irvine, Jonathan Valamehr, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Hardware trust implications of 3-D integration. In *Proceedings of the 5th Workshop on Embedded Systems Security (WESS)*, Scottsdale, AZ, October 2010.

[9] Ted Huffmire, Timothy Levin, Thuy Nguyen, Cynthia Irvine, Brett Brotherton, Gang Wang, Timothy Sherwood, and Ryan Kastner. Security primitives for reconfigurable hardware-based systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(2), May 2010.

[10] Y. Kaiho, Y. Ohara, H. Takeshita, K. Kiyoyama, K-W Lee, T. Tanaka, and M. Koyanagi. 3D integration technology for 3D stacked retinal chip. In *IEEE International Conference on 3D System Integration*, San Francisco, CA, September 2009.

[11] Nauman H. Khan, Syed M. Alam, and Soha Hassoun. System-level comparison of power delivery design for 2D and 3D ICs. In *Proceedings of the IEEE International Conference on 3D System Integration (3DIC)*, San Francisco, CA, September 2009.

[12] Jongman Kim, Chrysostomos Nicopoulos, Dongkook Park, Reetuparna Das, Yuan Xie, N. Vijaykrishnan, Mazin S. Yousif, and Chita R. Das. A novel dimensionally-decomposed router for on-chip communication in 3D architectures. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[13] Hsien-Hsin S. Lee and Krishnendu Chakrabarty. Test challenges for 3d integrated circuits. *IEEE Design and Test of Computers*, 26(5), September/October 2009.

[14] Timothy E. Levin, Cynthia E. Irvine, Clark Weissman, and Thuy D. Nguyen. Analysis of three multilevel security architectures. In *Proceedings of the ACM Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, November 2007.

[15] Zhouyuan Li, Xianlong Hong, Qiang Zhou, Shan Zeng, Jinian Bian, Hannah Yang, Vijay Pitchumani, and Cheng-Kuan Cheng. Integrating dynamic thermal via planning with 3D floorplanning algorithm. In *Proceedings of the International Symposium on Physical Design (ISPD)*, San Jose, CA, April 2006.

[16] Gabriel H. Loh. 3-D stacked memory architectures for multi-core processors. In *International Symposium on Computer Architecture (ISCA)*, Beijing, China, June 2008.

[17] Gabriel H. Loh, Yuan Xie, and Bryan Black. Processor design in 3D die-stacking technologies. *IEEE Micro*, 27(3), May-June 2007.

[18] Gian Luca Loi, Banit Agrawal, Navin Srivastava, Sheng-Chih Lin, Timothy Sherwood, and Kaustav Banerjee. A thermally-aware performance analysis of vertically integrated (3-D) processor-memory hierarchy. In *Proceedings of the 43rd Design Automation Conference (DAC)*, San Francisco, CA, July 2006.

[19] Erik Jan Marinissen. Testing tsv-based three-dimensional stacked ics. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE)*, Dresden, Germany, March 2010.

[20] S. Mysore, B. Agrawal, S.C. Lin, N. Srivastava, K. Banerjee, and T. Sherwood. Introspective 3-D chips. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 2006.

[21] Vasilis F. Pavlidis and Eby G. Friedman. *Three-Dimensional Integrated Circuit Design*. Morgan Kaufmann, Boston, MA, 2009.

[22] K. Puttaswamy and G.H. Loh. Thermal analysis of a 3D die-stacked high-performance microprocessor. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI'06)*, Philadelphia, PA, May 2006.

[23] Seyyed Ahmad Razavi, Morteza Saheb Zamani, and Kia Bazargan. A tileable switch module architecture for homogeneous 3D FPGAs. In *Proceedings of the IEEE International Conference on 3D System Integration*, San Francisco, CA, September 2009.

[24] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Communications of the ACM*, 17(7), July 1974.

[25] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, Boston, MA, October 2004.

[26] Kenneth M. Thompson. Intel and the myths of test. *IEEE Design and Test of Computers*, 13(1), Spring 1996.

[27] Mohit Tiwari, Hassan Wassel, Bita Mazloom, Shashidhar Mysore, Frederic Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, March 2009.

[28] Jonathan Valamehr, Mohit Tiwari, Timothy Sherwood, Arash Arfaee, Ryan Kastner, Ted Huffmire, Cynthia Irvine, and Timothy Levin. Hardware assistance for trustworthy systems through 3-D integration. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Austin, TX, December 2010.

[29] Z. Wang and R. Lee. New cache designs for thwarting cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[30] Xiaoxia Wu, Paul Falkenstern, Krishnendu Chakrabarty, and Yuan Xie. Scan-chain design and optimization for three-dimensional integrated circuits. *ACM Journal on Emerging Technologies in Computing Systems*, 5(2), July 2009.

[31] Bo Yang, Kaijie Wu, and Ramesh Karri. Secure scan: A design-for-test architecture for crypto chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10), October 2006.

[32] Hiroshi Yoshikawa, Atsuko Kawasaki, Tomoaki Iiduka, Yasushi Nishimura, Kazumasa Tanida, Kazutaka Akiyama, Masahiro Sekiguchi, Mie Matsuo, Satoru Fukuchi, and Katsutomo Takahashi. Chip scale camera module (CSCM) using through-silicon via (TSV). In *IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, February 2009.

Declarative FPGA Circuit Synthesis using Kansas Lava

Andy Gill
Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas
Email: andygill@ittc.ku.edu

Abstract—*Designing and debugging hardware components is challenging, especially when performance requirements demands a complex orchestra of cooperating and highly synchronized computation engines. New language-based solutions to this problem have the potential to revolutionize how we think about and build circuits. In this paper, we describe our language-based approach to semi-formal co-design. Using examples, we will show how generative techniques, high-level interfaces, and refinement techniques like the worker/wrapper transformation can be used to take descriptions of specialized computation, and generate efficient circuits. Kansas Lava, our high-level hardware description language built on top of the functional language Haskell, acts as a bridge between these computational descriptions and synthesizable VHDL. Central to the whole approach is the use of Haskell types to express communication and timing choices between computational components. Design choices and engineering compromises during co-design become type-centric refinements, encouraging architectural exploration.*

Keywords: Hardware, Synthesis, Functional Programming

1. Introduction

Generated VHDL offers many advantages over hand-written VHDL. Kansas Lava is a framework for expressing synthesizable hardware and generating VHDL, with the express purpose of exploring computational models for describing hardware level concerns. In this paper, we will describe the benefits of generated code, supported models of computation, and a look at a number of applications we have constructed using Kansas Lava.

Reconfigurable computing and co-design provides the opportunity to be flexible about how and where something is computed. The larger fabrics provided by modern FPGAs open the option of non-specialists writing VHDL programs, much in the same way early microprocessors allowed assembly language programming for efficient numerical computation. For highly customized parallel computations, well designed FPGA fabrics can perform at breathtaking scale, unmatched even by modern multi-cores. Yet tool support for migrating highly-computational tasks to IP cores on FPGAs is rudimentary, at best, and from a tool perspective, certainly comparable with early microprocessor development tools.

When considering generating circuit descriptions, VHDL and Verilog are sibling languages, at least from a language semantic point of view, and many FPGA tools support both as input to be compiled for FPGA fabrics. In this paper, we will examine a VHDL generator, and use VHDL as our language in examples. The same ideas could be retargeted to use Verilog with only nominal engineering effort.

VHDL provides both a way of expressing structure, or connections between existing components, as well a Register Transfer Level (RTL) style of expressing how a component acts based on register contents and a state machine. Both are used in practice by engineers programming FPGAs. There are other, more-advanced computational programming models, for example the atomic transaction model used by BlueSpec [1]. Kansas Lava provides a straightforward way of expressing the basic model of connected components, and a highly customizable methodology for providing RTL and other models on top of the connectivity. In this way, Kansas Lava becomes as test bed of ways of customizing models for expressing solutions to hardware-level problems. Ultimately, the vision is having custom languages, tailored to specific problem classes, that allow experts in the problem classes to program FPGAs effectively.

Hardware components communicate using buses and wires. VHDL uses a signal abstraction to represent almost all such communications. As such, a VHDL signal is fundamentally physical. Using Kansas Lava, we can explicitly represent higher-level communication abstractions, including partial values, bus protocols, and wiring inside multi-clock or hyper-clocked environment. This richness and depth of communication options brings opportunities that would be completely unreasonable to expect an engineer to directly manage; the Kansas Lava abstractions build on top of primitives provided in VHDL instead.

The analogy of VHDL being an assembly language for FPGAs is an timely one. As a community, we are providing new computational and communication abstractions for FPGA programmers, and hope it will lead to a critical mass of useful applications and happy users. In this paper, we will overview our specific system, Kansas Lava, and explain how it both helps users and provides abstraction opportunities. We will not go into details about the *internals* of Kansas Lava; we refer interested readers to our earlier explanations [2], [3] for specifics. Instead, we focus on using Kansas Lava. To do so, some knowledge of functional programming and Haskell [4] is needed, because this is the language-based foundation on which we build our tools.

2. Haskell

Haskell is the premier pure functional programming language. Haskell supports many forms of abstraction, and provides a robust foundation for building Domain Specific Languages (DSLs). In this section, we give a terse introduction to Haskell, sufficient to make this paper self-contained.

Haskell is all about *types*. Types in Haskell, like types in other languages, are constraining summaries of structural values. For example, in Haskell `Bool` is the type of the values `True` and `False`, `Int` is the type of machine-sized words, `Double` is the type of double precision floating point values; and this list goes on in the same manner as `C`, `C++`, `Java` and other traditional languages. All these type names in Haskell start with an upper-case letter.

On top of these basic types, Haskell has two syntactical forms for expressing compound types. First, pairs, triples and larger structures can be written using tuple syntax, comma separated types inside parenthesis. So `(Int,Bool)` is structure with both an `Int` and a `Bool` component. Second, lists have a syntactical shortcut, using square brackets. So `[Int]` is a list of `Int`.

Haskell also has other container types. A container that *may* contain one `Int` has the type `Maybe Int` which is read `Maybe` of `Int`. These container names also start with upper-case letters. Types can be nested to any depth. For example, we can have a `[(Maybe (Int,Bool))]`, read as list of `Maybe` of (`Int` and `Bool`).

Polymorphic values, which are analogous to the type `Object` in Java, or `void*` pointers in C, are expressed using lower-case letters. These polymorphic values can have constraints expressed over them, using the Haskell equivalent of an object hierarchy. Finally, a Haskell function that takes a list, and returns a list is written as using an arrow: `[a] -> [a]`.

We can now give an example of a Haskell function.

```
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = sort before ++ [x] ++ sort after
  where
        before = filter (<= x) xs
        after  = filter (> x) xs
```

This function sorted a list, using a variant of quicksort in which the pivot is the front of the list.

- The first line is the type for `sort`. This is ∀`a`, such that `a` can be `Ord`ered (admits comparisons like <=), the function takes and return a list of such `a`'s.
- The second line says that an empty list is already sorted.
- The remaining lines state that a (non-empty) list can be sorted by taking the first and rest of the list (called `x` and `xs`, respectively), and sorting the values before this pivot and after this pivot, and concatenating theses intermediate values together.
- Finally, intermediate values can be named using the `where` syntax; in this case the values of `before` and `after`.

Haskell is a concise and direct language. Structures in Haskell are described using types, constructed and deconstructed, but never updated. The entire language functions by chaining together these structural processors, which ultimately take input, and produce output. Side-effects are described using a `do`-notation, for example:

```
main :: IO ()
main = do
  putStrLn "Hello"
  xs <- getLine
  print xs
```

In this example a *value* called `main` uses the `do`-notation to describe an interaction with a user. Actually, the `do`-notation captures this as a structure called a Monad; purity is not compromised. For more details about how `do`-notation and Monads can provide an effectful interface inside a pure language like Haskell, see [5]. For the purposes of Kansas Lava, `do`-notation is a way of providing syntax and structure that looks like interaction.

3. Kansas Lava Primer

Kansas Lava is a Haskell library that models circuits as well as generating VHDL. In this section, we introduce the two main value descriptor types, and system modeling capabilities of Kansas Lava, returning to to the actual *generation* VHDL in section 6.

3.1 Combinational Values

Operations on combinational values are the basis of hardware computations. Examples include an addition function operating over numerical values, or a nand gate operating over boolean values. To take an example, and code it in Kansas Lava, consider a half adder.

```
halfAdder :: Comb Bool
          -> Comb Bool
          -> (Comb Bool,Comb Bool)
halfAdder a b = (carry,sum)
 where carry = and2 a b
       sum   = xor2 a b
```

The type indicates that this function, `halfAdder` operates over two arguments, both `Comb Bool`, or combinational booleans. The result, the `carry` and `sum`, are return as a 2-tuple of `Comb Bool`. (The type would typically be written on a single line after the function name, but is

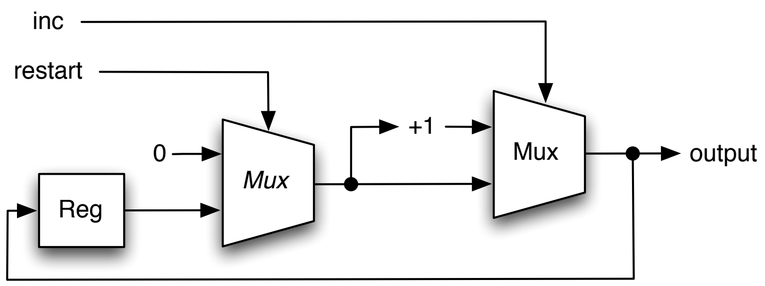


Fig. 1: `counter` schematic

folded over two lines to fit into a single column.) Inside the function definition, the intermediate values `carry` and `sum` are declared using the Haskell `where` keyword. This definition of a half adder is simple, direct, declarative and clear.

`Comb` is an *active* container for representing combinational values, where the type argument of `Comb` is the value being represented. `Comb Bool` is the type of a combinatorial boolean, in VHDL this would be a `std_logic`. `Comb Word8`, a combinatorial 8-bit value could be represented in VHDL using a `std_logic_vector`, or even an IEEE `signed`.

Executing our half adder example (as a simulation) is as simple as calling the `halfAdder` function. (The user interaction is notated starting with the `GHCi>` prompt.)

```
GHCi> halfAdder high low
(low,high)
GHCi> halfAdder high high
(high,low)
```

In summary, `Comb` x represents a single value x, computed using a combinatorial circuit.

3.2 Sequential Values and Clocks

Combinatorial logic by itself is completely uninteresting, and needs multiple invocations on different values, constructing sequential circuits, to do useful computations. Kansas Lava has a type `CSeq`, a mnemonic for Clocked Sequence, for representing a stream of values created over time, using sequential logic. A specific clock is used to interpret when to sample the values communicated over a sequence. We do not assume any global clock, rather we use types to attempt to contain their interpretive utility.

There are two Kansas Lava functions that are the primitive sequential circuit builders, `register`, and `delay`.

```
register :: (Clock c, sig ~ CSeq c)
         => a -> sig a -> sig a
delay    :: (Clock c, sig ~ CSeq a)
         =>      sig a -> sig a
```

`register` is a classical “D” edge-triggered flip-flop, where the clock is implicit. `register` takes two arguments, the initial value (at startup and after a reset), and the input sequence. `delay` is a version of `register` where the initial value is intentionally undefined. The syntax for the constraint on the type of both `register` and `delay` states there is a clock called `c`, and there is a signal called `sig`, which is interpreted using this clock. In this way, the types state that the input and output of both functions are in the same clock domain.

As an example of using `register`, consider:

```
counter :: (Rep a, Num a, Clock clk, CSeq clk ~ sig)
        => sig Bool -> sig Bool -> sig a
counter restart inc = loop
   where reg = register 0 loop
         reg’ = mux2 restart (0,reg)
         loop = mux2 inc (reg’ + 1, reg’)
```

This circuit connects two multiplexers, an adder, and a `register` to give a circuit that counts the number of clocked pulses on a the signal `inc`. The circuit takes two clocked signals, and returns a clocked signal that explicitly operates using same clock, because they share the same type. Figure 1 gives the circuit intended for this description.

We can execute sequential circuits with the same directness that combinational functions we invoked.

```
GHCi> toSeq (cycle [True,False,False])
T : F : F : T : F : F : T : F : F : ...
GHCi> counter low (toSeq (cycle [True,False,False]))
1 : 1 : 1 : 2 : 2 : 2 : 3 : 3 : 3 : ...
```

Both `Comb` and `CSeq` provide a direct representation of signals in VHDL, or more precisely, a meaning allowing interpreting a VHDL signal. Both `Comb` and `CSeq` also provide lifting, allowing the representation of unknown (in VHDL, `X`) values. On this basis, we build Kansas Lava.

3.3 Signals

In VHDL, signals are a generalization of `Comb` and `CSeq`. In Kansas Lava we allow a third class of value, a `Signal`, which is a Haskell class admitting both `Comb` and `CSeq`. The type for `and2` actually is:

```
and2 :: (Signal sig)
        => sig Bool -> sig Bool -> sig Bool
```

This use of `Signal` literally means either a `Comb` or a `CSeq` with a `Clock` can be used as a signal. In this way, `Signal` brings together combinational and sequential values, allowing overloaded primitives to be either.

Circuits are composed of sequential and combinational components, but values generated by sequential and combinational circuits have different types. We allow combinational circuits (which can only be run once) to be prompted, or lifted, into circuits that can be run sequentially many times. So we have have primitive combinational operators, sequential operators like `register`, and a way of *lifting* combinator functions into either combinatorial *or* sequential functions.

4. ROMs and Lifted Functions

As well as this trio of basic signal types, we can build circuits that operate on Haskell functions directly, provided the domain of the function is finite. We use the `Rep` constraint to signify that we can enumerate all possible representable values in a type, giving the funMap function.

```
funMap :: (Rep a, Rep b, Signal sig)
       => (a -> Maybe b)
       -> sig a -> sig b
```

The generated circuit is implemented using a ROM, and we can generate control logic directly in terms of Haskell functions and data-structures. As a example, consider a small ROM that stores the square of a value.

```
squareROM :: (Num a, Rep a, Signal sig)
             => sig a -> sig a
squareROM = funMap (\ x -> return (x * x))
```

In this way, direct Haskell functions can be lifted into `Signal` world. Notice how `squareROM` function is not specific about size, but is completely generic, as long as the argument stream, of type "a", is representable as a number.

`funMap` used directly behaves as an asynchronous ROM. A synchronous ROM can be constructed out of a `funMap` followed by a `register` or `delay`. The specific decision *how* to represent a ROM is postponed to VHDL generation time. The simulator simply stores and looks up values using the function directly.

5. Structures and Types

Kansas Lava has some general purpose types that turn out to be especially useful with constructing descriptions of hardware. Specifically Kansas Lava has support for generically sized fixed-width signed and unsigned numbers, and generically sized fixed-width matrixes. We use a sized type [6], notated as $\mathtt{X}_n$, to annotate sizes onto types. Examples include `Unsigned X4`, a 4-bit unsigned number, `Signed X8`, a 8-bit signed number, and `Matrix X6 Bool`, a (one-dimensional) matrix with 6 elements that are of type `Bool`. Use of irregular-sized values

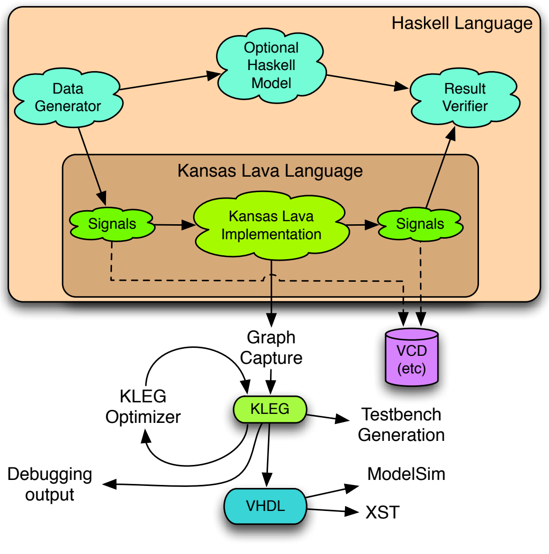


Fig. 2: Kansas Lava Architecture

is common in Hardware, while having a 14-bit adder is unheard-of in software. We also provide type shortcuts for common sizes, giving `U4` for the 4-bit unsigned number, and using $\mathtt{S}_n$ and $\mathtt{M}_n$ as names for signed numbers and matrixes, respectively.

On top of our signal types (`Comb`, `CSeq` and `Signal`) Kansas Lava builds a number of useful type-oriented utilities. The major one is the `pack` and `unpack` combinators. `pack` provides a way of taking a bundle of signals, and packing them into a single signal. `unpack` does the reverse operation, unpacking a single signal into a bundle of signals.

As a concrete example, consider a pairing of a `Bool`ean signal, and an 8-bit unsigned signal (`U8`), with the names `a` and `b`:

```
(a,b) :: (Signal sig) => (sig Bool, sig (U8))
```

We can use `pack` to pack this pairing into a single signal

```
pack (a,b) :: (Signal sig) => sig (Bool,U8)
```

As an example, we can define `(a,b)`, then pack the result inside the Kansas Lava simulation mode.

```
GHCi> let a = toSeq (cycle [True,False])
GHCi> a
T : F : T : F : T : F : T : F : ...
GHCi> let b = toSeq [1..]
GHCi> b
1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : ...
GHCi> pack (a,b)
(T,1) : (F,2) : (T,3) : (F,4) : (T,5) : ...
```

`pack` and `unpack` turn out to be extremely useful in practice, and support the objective of allowing highly flexible type representation for data flowing inside an implemented program.

Finally, there are structural types for expressing common protocols used in hardware. For example `Enabled a` means a value `a` that has an extra `Boolean` value signifying validity. Another example is `Write addr val`, which may contain a write request sent to a memory.

Overall, the type system and structures of the core Kansas Lava system described here provide a useful replacement for describing and simulating circuits, comparable to VHDL and ModelSim, but with richer and more descriptive types. Our earlier paper [2] gives detailed justification for each feature in this core from a language point of view. The interesting work is what can be built on top of this foundation.

6. Generating VHDL

The main purpose of Kansas Lava is to generate useful VHDL. As the earlier examples illustrated, the Haskell system can be used to directly simulate circuit behavior. Taking this behavior, and correctly mapping it to efficient VHDL is done using a technique called reification.

Figure 2 gives an overview of the the major components of a successful invocation of Kansas Lava. A Kansas Lava program is executed in terms of generated test vectors, or signals, and these signals can be optionally compared with output generated by a high-level model. The same program can be extracted [7], without needing the specific signal context, and captured as a KLEG (Kansas Lava Entity Graph), which is for all intents and purposes a simple net-list.

If we take our `counter` example from above, the question is how to turn this into a VHDL architecture (from the *body* of the function), and VHDL entity (from the *type* of the function). In VHDL, all arguments in an entity are named and unordered, in Haskell and Kansas Lava, they are unnamed and order matters. Rather than just inventing names (which turns out to be brittle to even minor changes) we explicitly name arguments use a mathematical construction called a monad.

A monad is a way of expressing computation. For the purposes of the discussion of this specific monad, we can think of `Fabric`, the monad in question, as an abstract data structure. The interface for specifying inputs and output in VHDL is concise, and summarized in figure 3.

In figure 3 `inStdLogic` names an input, giving a `Fabric` that returns `Seq Bool`. `outStdLogic` names an output, and provides a `Seq Bool`, and returns a `Fabric`. `inStdLogicVector` and `outStdLogicVector` are versions of `inStdLogic` and `outStdLogic` respectively, that operate on a vector rather than an individual bit.

```
inStdLogic        :: (sig ~ CSeq ())
                  -> String
                  -> Fabric (sig Bool)
inStdLogicVector  :: (Rep a, sig ~ CSeq ())
                  => String
                  -> Fabric (sig a)
outStdLogic       :: (sig ~ CSeq ())
                  -> String
                  -> sig Bool
                  -> Fabric ()
outStdLogicVector :: (Rep a, sig ~ CSeq ())
                  => String
                  -> sig a
                  -> Fabric ()
```

Fig. 3: Fabric API

We can build a fabric around our counter, using the following fabric-building code.

```
counterFabric :: Fabric ()
counterFabric = do
        restart <- inStdLogic "restart"
        inc     <- inStdLogic "inc"
        let res :: (Clock clk) => CSeq clk U4
            res = counter restart inc
        outStdLogicVector "output" res
```

This use of fabric *must* clarify the specific types being operated on; in this case we are returning a signal of `U4`.

As well as building a Fabric, this is the Kansas Lava method for specifying port names. We can reify (capture) this `Fabric`, and the call to `counter` inside it, using a function called `fabricReify`.

```
fabricReify :: Fabric () -> IO KLEG
```

`KLEG` is an abstract representation of our net-list, and can be printed, optimized, or written into VHDL. This program is all the code needed to generate valid VHDL.

```
main = do
  k <- fabricReify counterFabric
  writeVHDL "counter" "counter.vhd" k
```

The contents of `"counter.vhd"` are shown in Figure 4.

The summary regarding VHDL generation is a simple narrative. We build a thin veneer around our circuit functions called a `Fabric`, which specifies how our circuits are invoked, and what to name our ports. These declarations, and the behavior of the underlying circuit, is realized in the form of a VHDL entity and architecture. As already stated, there is no reason that Verilog could not be generated instead; indeed we use a generic netlist rendering tool that can already target VHDL or Verilog.

```
entity counter is
  port(rst : in std_logic;
       clk : in std_logic;
       clk_en : in std_logic;
       restart : in std_logic;
       inc : in std_logic;
       output : out std_logic_vector(3 downto 0));
end entity counter;
architecture str of counter is
  signal sig_2_o0 : std_logic_vector(3 downto 0);
  ...
begin
  sig_2_o0 <= sig_5_o0 when (inc = '1')
                       else sig_6_o0;
  sig_5_o0 <= std_logic_vector(...);
  sig_6_o0 <= "0000" when (restart = '1')
                     else sig_10_o0;
  sig_10_o0_next <= sig_2_o0;
  proc14 : process(rst,clk) is
  begin
    if rst = '1' then
      sig_10_o0 <= "0000";
    elsif rising_edge(clk) then
      if (clk_en = '1') then
        sig_10_o0 <= sig_10_o0_next;
  ....
end architecture;
```

Fig. 4: Fragments of VHDL Generated for `counter`

7. Example: parity checking

To take a front to back example, consider a block of logic that listens on a channel for parity errors, counting them. We want to know if there were *any* parity error, but for debugging and error reporting, we report a parity error count. There is a control signal, typed `sig DecodeCntl`, which encodes when to start and stop counting parity counts. Figure 5 gives an example implementation, taken from an implementation of a LDPC forward error corrector [8].

There are a number of typical illustrative aspects to this example.

- The use of `pack` and `unpack` is an integral part of the description.
- The call to `counter`, our earlier example, can be seen as a simple function invocation on the final line.
- `liftPred` uses a Haskell-level equality test, checking to see if the control signal is at the minimum (start) or maximum (termination) delimiter.
- The output is only valid on the `stop` cycle, with validity communicated using a `pack`ed boolean value.
- Finally, the *type* is a detailed description of what this component requires. In this case, a control signal and a set of write "packets", giving a 16-bit result signal, with a validity bit attached.

We can now simulate and independently test this circuit, by creating two input streams. For this test, we construct a smaller control data-structure, where the minimum `DecodeShare` value and the maximum `DecodeShare` value is 3. We can now execute our example.

```
parityCount :: (Clock clk, sig ~ CSeq clk)
            => sig DecodeCntl
            -> sig (Write SZ Bool)
            -> sig (Enabled U16)
parityCount control parityWrite = pack (stop,res)
  where (en,datam) = unpack parityWrite
        (addr,val) = unpack datam
        start = liftPred (== DecodeShare minBound)
                         (delay control)
        stop  = liftPred (== DecodeShare maxBound)
                         (delay control)
        res = counter start (en `and2` val)
```

Fig. 5: Parity Counting in LDPC

```
GHCi> let control = ...
           ++ [DecodeShare n | n <- [0..3]]
           ++ ...
GHCi> let writes = ...
          ++ [ Write (0,m)
             | m <- [True,False,True,False]
             ]
          ++ ...
GHCi> parityCount control writes
... : Nothing : Nothing : Nothing : Just 2 : ...
```

Once we are happy with this component, we can plug it into a larger Kansas Lava circuit, or render VHDL as a stand-alone VHDL component, by creating a `Fabric` round a call to `parityCount`.

8. Discussion

The core Kansas Lava system is an academic project, but mature and complete enough to engineer large examples. There are three principal things you can do with (a system like) Kansas Lava, beyond simply write code in Kansas Lava. You can (1) solve problems and express solutions using generative techniques; (2) build new models of computation and other abstractions on top of the existing interface; and (3) use the language as a target of refinements from a yet-higher level of specification of abstract behavior.

Generative Programming [9] is an idiomatic name for the general case of programs generating programs. In Kansas Lava, any circuit generator shares aspects of this powerful idiom. For example, a recursive divide and conquer algorithm can be used to generate a circuit, when the base-case generates small, hand crafted solutions, and the division invocations connect sub-circuits together. We discuss a specific example of using recursion for circuit generation in section 10.

A generalization of these ideas is build new abstractions on top of the the basic syntax. When constructing circuits in VHDL, often a Register Transfer Level (RTL) idiom is used, where hardware behavior is expressed as clocked transitions between states with additional memory. In section 9 we present an extension on top of Kansas Lava that provides an RTL interface abstraction, building on the Haskell do-notation.

```
rate :: forall x clk . (Clock clk, Size x) => Witness x -> Rational -> CSeq clk Bool
rate Witness n
  | step * 2 > 2^sz = error $ "bit-size " ++ show sz ++" too small for punctuate Witness " ++ show n
  | n <= 0  = error $ "can not have rate less than or equal zero"
  | n > 1 = error $ "can not have rate greater than 1, requesting " ++ show n
  | otherwise = runRTL $ do
     count <- newReg (0 :: (Unsigned x))
     cut   <- newReg (0 :: (Unsigned x))
     err   <- newReg (0  :: (Signed x))
     CASE [ IF (reg count .<. (step + reg cut - 1)) $
                count := reg count + 1
          , OTHERWISE $ do
                count := 0
                CASE [ IF (reg err .>=. 0) $ do
                         cut := 1
                         err   := reg err + nerr
                     , OTHERWISE $ do
                         cut := 0
                         err   := reg err + perr
                     ]
          ]
     return  (reg count .==. 0)
           where sz = fromIntegral (size (error "witness" :: x)) :: Integer
                 num = numerator n
                 dom = denominator n
                 step = fromIntegral $ floor (1 / n)
                 perr = fromIntegral $ dom - step       * num
                 nerr = fromIntegral $ dom - (step + 1) * num
```

Fig. 6: RTL-based sample pulse builder

Finally, Kansas Lava, with or without the generative programming extensions, can be a target language of a refinement methodology. A clear short description of an algorithm can be connected to a real implementation of a circuit, using a set of formal or semi-formal refinement steps. In section 11 we discuss a larger example of showing a relationship between a specification and implementation discussed in section 10.

9. Example: Register Transfer Language API

Sometimes state-based thinking leads to clear descriptions of behavior. The core Kansas Lava system is stream based, with conditionals handled by `mux2` and related combinators. It is possible to build a state-based API that models RTL on top of core Kansas Lava.

Consider the problem of generating a fractionally rated boolean signal. This may be used, for example to trigger a sampling of an input port. When sampling higher-rate data transfers, the ratio of clock cycles to sample rate within acceptable tolerance might not have an integral reciprocal of a fraction. For example, when using a 50MHz clock, sampling a signal at 6MHz requires waiting $8\frac{1}{3}$ clock cycles between samples; clearly not possible. Instead, a sample rate of 6MHz overall can be achieved by waiting 8 *or* 9 cycles between samples, and averaging the gap to $8\frac{1}{3}$ clock cycles. This can be solved with Bresenham's Line Algorithm, which has to approximate a fractional rate of a line using whole pixels.

Figure 6 gives a complete implementation of such a function, using the RTL extension. As can be seen, the example reflects the RTL style of VHDL state-based programming, but keeps the type-based approach of Kansas Lava intact. To explain the example, `rate` takes a witness (approximately the analog of a generic argument in VHDL) and a fractional argument, to return a clocked `Bool` that will be punctuated at the given rate. Again, the generative motive appears; this function takes runtime arguments, to generate a circuit.

After checking for various pre-conditions, `rate` allocates three registers, and conditionally either increments a counter, or resets it and records an error from ideal. The key function is `runRTL`, which has type:

```
runRTL :: (Clock c)
       => (forall s . RTL s c a) -> a
```

`runRTL` takes a structure, called a `RTL`, that looks like a list of assignments using `do`-notation, and converts this list guarded assignments into regular Kansas Lava signal-based code. `runRTL`, in a real sense, provided the semantics and implementation of the RTL abstraction. In this case, a straightforward clocked atomic transfer semantics have been encoded, but there is no reason why a more powerful semantics, like the semantics of the high-level modeling language BlueSpec [1] could be coded. Indeed, this framework allows the exploration of new RTL-based abstraction models with extremely low cost of development, because so much of the existing infrastructure is reused. For comparison purposes, the entire RTL abstraction, including `runRTL`, is implemented in around 150 lines of commented Haskell.

```
operateOnMatrix :: (Clock clk, sig ~ CSeq clk)
  => Options
  -> A (Int,Int)
  -> sig DecodeCntl
  -> [ sig (Write SZ FLOAT) ]
  -> [ Maybe (Int,sig (Write SZ FLOAT)) ]
 -> ( [ Maybe (Int,sig (Write SZ FLOAT)) ],
      [ Maybe (sig (Write SZ FLOAT)) ],
      [ Maybe (sig (Write SZ Bool)) ]
    )
```

Fig. 7: Type of the Recursive LDPC Fabric Generator

10. Example: Recursion and Circuit Generation

In a recent project, we needed to generate a computational fabric to perform Low Density Parity Check (LDPC) forward error correcting [10], based on operations to a large sparse matrix. Specifically, using the notation from the standard reference on error correcting codes [11], the LDPC fabric needed to compute three things as quickly as possible:

For each (m,n) where $A(m,n) = 1$:

$$\eta^{[l]}_{m,n} = -\tfrac{3}{4}\left((\!|\min^{\dagger}|\!)_{j\in\mathcal{N}_{m,n}} \; (\eta^{[l-1]}_{m,j} - \lambda^{[l-1]}_{j}) \right)$$

where

$$\min^{\dagger}(x,y) = sign(x) * sign(y) * min(|x|,|y|)$$

For each n:

$$\lambda^{[l]}_{n} = \lambda^{[0]}_{n} + \textstyle\sum_{m\in\mathcal{M}_n} \; \eta^{[l]}_{m,n}$$

For each n:

$$\hat{c}^{[l]}_{n} = 1, \text{if} \lambda^{[l]}_{n} > 0, \text{otherwise} = 0.$$

The input, $\lambda^{[0]}$ is a soft input, being a representation of likelihood of a symbol, with zero representing unknown. $\mathcal{M}_n$ represents $\{m : A_{m,n} = 1\}$, and $\mathcal{N}_{m,n}$ represents $\{n : A_{m,n} = 1\} \setminus n$.

A is a sparse matrix, approximately 7K by 3K, with a density of around 6 non-zero elements per row. After analyzing the problem domain and the (fixed) matrix, we decided to generate a two dimensional fabric of small unit blocks, that would be controlled by a central controller. These smaller blocks would communicate with each other in a serial manner, cutting down internal wiring overhead.

Figure 7 gives the type of our recursive implementation. Though involved, the type spells out exactly what gets passed to, and returned from, `operateOnMatrix`. Specifically, `operateOnMatrix` is a recursive function. `operateOnMatrix` checks to see if `A` is at a threshold. If the threshold is satisfied, `operateOnMatrix` generates a small circuit we call a cell, otherwise two recursive calls are made to `operateOnMatrix`, to generate two sub-circuits, and they are combined to return a larger circuit. The technical details of this implementation can be found in [8].

11. Example: Derivation of a Forward Error Corrector

As a final example, we connect together the LDPC example from section 10 to a specification of LDPC, also written in Haskell. Consider again the main equation in the LDPC algorithm.

For each (m,n) where $A(m,n) = 1$:

$$\eta^{[l]}_{m,n} = -\tfrac{3}{4}\left((\!|\min^{\dagger}|\!)_{j\in\mathcal{N}_{m,n}} \; (\eta^{[l-1]}_{m,j} - \lambda^{[l-1]}_{j}) \right)$$

In Haskell, a transliteration of the same operation *is* a semi-formal and executable specification, and was sufficiently fast to generate the required test vectors.

```
eta' = [ ( (m,n)
         , -0.75 * (fold minDagger
                        [ eta ! (m,j) - lam ! j
                        | j <- toList (rows a ! m)
                        , j /= n
                        ]))
       | (m,n) <- toList a
       ]
```

This is a classic example of Haskell being used as a specification, with the whole decoder being specified in around 20 lines of Haskell. The motivation for starting at this specification was one of cautiousness, because implementing LDPC is notoriously tricky. So a refinement exercise was undertaken, inside our text editor.

- The original Haskell-based specification was refined so that the computations were performed in specific computation cells, and cells were connected by streams of values.
- The individual sub-components of these cells were replaced systematically with Kansas Lava primitives, in a correctness preserving manner.
- Further refinements were done, to push out from these sub-components, until the whole original model was a large (around 800 line) Kansas Lava program, as previous discussed. At this point, the Kansas Lava program can be executed to generate the circuit.

In all, 17 distinct versions of the LDPC were developed, including the semi-formal model, each of which was executable. The final version, when executed, generated VHDL which was efficiently realizable in hardware. Every refined version was a cut-and-pasted version of its predecessor, applying by hand the refinements provided by worker/wrapper [12] and other fusion transformations. The whole exercise was time consuming, and did not allow for easy backtracking (any design change in an earlier version needed changed by hand in all subsequent programs). The only bugs found were missteps in our refinements, the final version was not compromised in terms of performance, with the hardware implementation of LDPC performing within 10% of the estimated clock-rate/performance.

Version 9

```
(...) => A (x,y)
      -> [ Stream (Matrix SZ FLOAT) ]
      -> [ Stream (Matrix SZ (Maybe FLOAT)) ]
      -> ( [ Stream (Matrix SZ (Maybe FLOAT)) ]
         , [ Stream (Matrix SZ (Maybe FLOAT)) ])
```

Version 10

```
(...) => A (x,y)
      -> [ Stream (Matrix SZ FLOAT) ]
      -> [ Maybe (Stream (Matrix SZ FLOAT)) ]
      -> ( [ Maybe (Stream (Matrix SZ FLOAT)) ]
         , [ Maybe (Stream (Matrix SZ FLOAT)) ])
```

Fig. 8: Refinement Source and Target Types

Critical to this whole refinement was the hand-use at every step of the worker/wrapper transformation, a mechanism for refining types based on an algebra of type coercing function. The path through the LDPC refinement chain was planned by deciding the specific types of communicating sub-components at each step. From these, type coercion functions are constructed, and simple fusion properties of the coercions were used to allow the new implementations to be derived. Most of the process was mundane, after the key type-based planning had been done.

To ground this refinement description, figure 8 gives an example of a refinement step on our key implementation function. This function takes the sparse matrix A, which represents the specific parity code, and generates a program that takes and returns data. At this point in the refinement (version 9 to version 10), we have two inputs and two outputs. To break down our type, reading from out to in, `[Stream (Matrix SZ (Maybe FLOAT))]` has the meaning

`[...]`	List representing the connection to a specific number of computational "cells";
`Stream ...`	of a stream of unclocked sequential values, one per iteration of LDPC;
`Matrix SZ ...`	of a matrix, size `SZ`;
`Maybe ...`	and an optional value;
`FLOAT`	of a custom sized, quantized, real number.

In this step, we want to move (or more specifically, commute) the `Maybe`, to between the list and the `Stream`. This step enables a key optimization of the implementation, specifically the ability to eliminate the circuitry of parity sub-maxtrixes that are encoded using the zero matrix. In the block-circuit versions of the LDPC, over half the cell-sized blocks are, by design, zeros. The preconditions for performing this specific type refinement using worker/wrapper hinges on making sure that a specific sub-block *always* generates the optional `FLOAT` value, or *never* generates an optional `FLOAT`. The step-specific coercions, as required by worker/wrapper, are straightforward to write, and using this pre-condition, also straightforward to validate for use. The worker/wrapper transformation can then be applied, and any sub-coercions can be pushed inside the definition of the implementation of version 9, deriving version 10 through equational reasoning and fusion laws.

A detailed description of this refinement can be found in [13]. The overarching narrative is that using the same underlying language for the specification (straight Haskell) and the implementation (Kansas Lava) has the benefit of allowing Haskell refinement techniques to use to connect, at least semi-formally, the implementation and specification.

12. Related Work

The original ideas for Lava can be traced back to the Ruby hardware description language [14] and prior to that, μFP [15]. A good summary of the principles behind Lava can be found in [16]. JHDL [17] is a hardware description language, embedded in Java, which shares many of the same ideas found in Lava.

The overarching use of refinement is inspired by the outstanding research undertaken by the Computing Laboratory at the University of Oxford. Specifically, the methodologies promoted by Bird, et. al. [18], [19] call for a brighter future for robust software and hardware development.

Kansas Lava is a modeling language as well a synthesis tool. It models communicating processes, via synchronous signals. There are several other modeling languages that share a similar basic computational basis, for example Esterel [20].

13. Summary and Conclusions

Kansas Lava is two years old. It has been used an number of projects, generating Forward Error Correctors, building communication bridges between UNIX hosts and FPGA computational engines, and other smaller components, including a double-buffered FIFO/interleaver VHDL component.

The core Kansas Lava implementation is mostly complete, and will be released as an open-source product in the near future. Experimentation with idioms, advanced APIs, and refinement engines continues. The long term goal of clear specifications, connected (via refinement or compilation) to implementation descriptions in being actively explored by the Kansas Lava project. The ideas here are general; there is no reason why there could not be a Kansas Lava analog for generating C or C++, for example, as is done in the CoPilot project [21]. Building descriptions of programs in languages like Haskell, and building further infrastructure on top of this has potential for a long-term impact to the way we build software, and design hardware.

References

[1] Arvind, R. Nikhil, D. Rosenband, and N. Dave, "High-level synthesis: an essential ingredient for designing complex asics," in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, nov. 2004, pp. 775 – 782.

[2] A. Gill, T. Bull, A. Farmer, G. Kimmell, and E. Komp, "Types and type families for hardware simulation and synthesis: The internals and externals of Kansas Lava," in *Proceedings of Trends in Functional Programming*, May 2010.

[3] A. Farmer, G. Kimmell, and A. Gill, "What's the matter with Kansas Lava?" in *Proceedings of Trends in Functional Programming*, May 2010.

[4] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries – The Revised Report*. Cambridge, England: Cambridge University Press, 2003.

[5] S. L. Peyton Jones and P. Wadler, "Imperative functional programming," in *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1993, pp. 71–84.

[6] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing Kansas Lava," in *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.

[7] A. Gill, "Type-safe observable sharing in Haskell," in *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.

[8] A. Gill, T. Bull, D. DePardo, A. Farmer, E. Komp, and E. Perrins, "Using functional programming to generate an LDPC forward error corrector," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, May 2011.

[9] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[10] K. S. Andrews, D. Divsalar, S. Dolinar, J. Hamkins, C. R. Jones, and F. Pollara, "The development of turbo and LDPC codes for deep-space applications," *Proc. IEEE*, vol. 95, no. 11, pp. 2142–2156, Nov. 2007.

[11] T. K. Moon, *Error correction coding : mathematical methods and algorithms*. Hoboken, N.J.: Wiley-Interscience, 2005. [Online]. Available: http://www.loc.gov/catdir/toc/ecip055/2004031019.html

[12] A. Gill and G. Hutton, "The worker/wrapper transformation," *Journal of Functional Programming*, vol. 19, no. 2, pp. 227–251, March 2009.

[13] A. Gill and A. Farmer, "Deriving an efficient FPGA implementation of a low density parity check forward error corrector," submitted to the 16th ACM SIGPLAN International Conference on Functional Programming.

[14] G. Jones and M. Sheeran, "Circuit design in ruby," in *Formal Methods for VLSI Design*, Staunstrup, Ed. Elsevier Science Publications, 1990.

[15] M. Sheeran, "mufp, a language for vlsi design," in *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*. New York, NY, USA: ACM, 1984, pp. 104–112.

[16] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *International Conference on Functional Programming*, 1998, pp. 174–184. [Online]. Available: citeseer.nj.nec.com/bjesse98lava.html

[17] P. Bellows and B. Hutchings, "JHDL - an HDL for reconfigurable systems," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, p. 175, 1998.

[18] R. S. Bird and O. De Moor, *Algebra of Programming*, ser. International Series in Computing Science. Prentice Hall, 1997, vol. 100.

[19] R. Bird, *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010. [Online]. Available: http://www.cambridge.org/gb/knowledge/isbn/item5600469

[20] G. Berry, "The constructive semantics of pure Esterel," 1999, http://www-sop.inria.fr/esterel.org/files/.

[21] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: A hard real-time runtime monitor," in *RV*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds., vol. 6418. Springer, 2010, pp. 345–359.

Towards Semantics-directed System Design and Synthesis

William L. Harrison[1], **Benjamin Schulz**[1], **Adam Procter**[1], **Andrew Lukefahr**[2] **and Gerard Allwein**[3]
[1]Department of Computer Science, University of Missouri, Columbia, MO, USA.
[2]Department of Electrical Engineering and Computer Science, University of Michigan-Ann Arbor, MI, USA.
[3]US Naval Research Laboratory, Code 5543, Washington, DC, USA.

Abstract—*High assurance systems have been defined as systems "you would bet your life on." This article discusses the application of a form of functional programming—what we call "monadic programming"—to the generation of high assurance and secure systems. Monadic programming languages leverage algebraic structures from denotational semantics and functional programming—monads—as a flexible, modular organizing principle for secure system design and implementation. Monadic programming languages are domain-specific functional languages that are both sufficiently expressive to express essential system behaviors and semantically straightforward to support formal verification.*

Keywords: Formal Methods, Computer Security, Programming Languages

1. Introduction

System software is notoriously difficult to reason about either formally or informally and this, in turn, greatly complicates the construction of high-assurance, secure systems. In our view, the difficulty stems from the conceptual distance between abstract models of secure systems and their concrete implementations. System models are formulated in terms of high-level abstractions while system implementations reflect the low-level and concrete details of hardware, machine languages and C. Typically, there is no apparent relationship between the system model and implementation to exploit in verifying critical system properties, and this disconnect impedes the construction of computer systems with verified security policies.

This paper describes ongoing research that pursues a novel approach towards bridging this conceptual gap: synthesizing implementations of systems directly from formal models of security in a manner verified to preserve the system security property. The particular systems we focus on are Rushby's classic security kernel design: *separation kernels* [1]. Separation kernels partition processes by security level into distinct "domains" (where all interdomain communication is mediated by the kernel) and enforce a non-interference-style security discipline called *domain separation* (see Figure 1). In a separation kernel, processes on different domains behave as if they are on distinct nodes in a networked distributed system while they, in fact, execute on shared hardware.

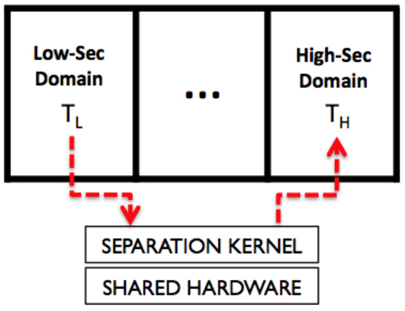


Fig. 1: A separation kernel mediates all inter-domain communication, thereby enforcing its security policy. The dotted arrow designates permitted information flows.

The methodology being explored is a calculational approach seeking the development, implementation and verification of compilation strategies that preserve domain separation. The main vehicle for our methodology is a domain-specific language, called HASK (for High-Assurance Security Kernel), and the aim of this approach is not to develop a single, specific security kernel, but rather a class of such kernels embodied as HASK programs.

This paper presents an overview of the design, implementation and verification of HASK programs. It does not report new technical innovations or scientific discoveries, but rather it presents an overview of the research, as well as the philosophy underlying the research, that is currently being pursued by the authors. Furthermore, an overview of both the philosophy that motivates our approach to high assurance and secure systems. The remainder of this section discusses and motivates the design of the HASK language. Section 2 places this research in the context of efforts to discover a science of cybersecurity. Section 3 describes two formalisms that provide a foundation for our work, monadic semantics [2] and channel theory [3].

1.1 Motivating the HASK Design

The "bread and butter" activities for any kernel include concurrency (e.g., task scheduling, synchronization, etc.), managing IO, and handling asynchronous exceptions and system calls. Implementing a kernel means supporting these functionalities among others.

If high assurance is a goal for the kernel, then the choice of implementation language is critical. Verifying formally that the kernel possesses certain properties (e.g., that it obeys an information security discipline) means proving mathematically that the kernel code itself possesses the desired properties. This, in turn, requires that the implementation language must possess a rigorous semantics. For higher assurance, the kernel language compiler and possibly other links in the toolchain must also be verified to preserve the desired properties.

Formal verification of code? This is frequently the point at which functional languages are mentioned, because several such languages (e.g., Haskell and ML) have rigorous semantic specifications [4], [5], [6], [7], [8] for large parts of the language.

The choice of a functional language for implementing the kernel is complicated by several unpleasant facts: "bread and butter" kernel behaviors (e.g., file handling, concurrency, exceptions, destructive update, etc.) are called the "Awkward Squad" in the functional programming community [9] precisely because they are difficult to handle in functional languages.

For the sake of discussion, let us consider the Haskell functional language [10] as an implementation language. Systems-level programming in Haskell inevitably involves the "IO monad". And what is the IO monad? In the memorably colorful words of Simon Peyton Jones (the principal architect of the Glasgow Haskell Compiler and a leading light in the Haskell community), the IO monad is a giant "sin-bin" [9] into which all real world impurities are swept. These impurities (e.g., file handling, concurrency, exceptions, destructive update, etc.) are called the "Awkward Squad" in the functional programming community precisely because they are difficult to handle in functional languages. Yet, the Awkward Squad is essential to kernel programming—after all, if a kernel isn't about handling concurrency, exceptions, etc., what is it about?

1.2 Making the Awkward Squad Less Awkward

HASK handles the Awkward Squad in two ways. HASK is a standalone domain-specific language (DSL) with its own compiler. HASK looks like Haskell [10], and, in fact, HASK programs can be executed by Haskell implementations with only some trivial syntactic changes. But, HASK's expressiveness is intentionally reduced to simplify its semantics, implementation and verification.

The other way that the Awkward Squad is made less awkward is with the "monadic programming" model of HASK. This distinguishes this research from that of others applying functional languages to system software [11], [12]. Historically, monads are used in two contexts: denotational semantics and functional programming. In denotational semantics, monads are algebraic structures representing computational effects, while within functional programming, monads are a programming abstraction for modularity akin to object orientation. Programming in HASK differs from ordinary functional programming practice in at least one key respect: monads are first-class language constructs.

Now is as good a time as any to explain a confusing misnomer in the Haskell world: the aforementioned IO monad in the Haskell language is a monad in name only. A monad [2], [13] is an algebraic structure is the same sense that a group, a ring or a vector space is an algebra; that is, it has operations defined on it of a certain type that obey certain defining equations. The Haskell IO monad has operations of the right type defined on it, but they are not guaranteed to obey the defining equations of a monad. Further muddying this already muddy water, Haskell also has a built type class called *Monad* and members of that type class, like the IO monad, are not necessarily monads in the algebraic sense. A Haskell programmer, having defined a *Monad*, would have to prove that his *Monad* is really a monad. In this paper, when we write "monad", we really mean monad in the rigorous, mathematical sense of the word. Monads are explained further in Section 3.

At this point, one might reasonably ask whether a whole new language is even necessary? Can't an existing language suffice—why a standalone language and not just a new library? High assurance is one of our main desiderata, and, if we are to have a chance of verifying our designs and tools, then we must program them in a language with a rigorous semantics. There are simply no general purpose languages with such a semantics. Such languages have so much "stuff" in them that establishing a humanly tractable language semantics would seem to be difficult, if not impossible.

HASK is a standalone language and requires its own compiler. While implementing HASK via existing Haskell compilers is possible, this style of implementation leaves much to be desired, both from the points of view of efficiency and high assurance. From an efficiency perspective, the kernel implementation produced by the GHC compiler is large and retains artifacts from the Haskell run-time system ill-suited to an operating system kernel (e.g., garbage-collection and closures). Neither GHC nor its runtime system were designed for formal verification.

2. What is a Science of Security?

There has been a lot of interest recently among various funding agencies in the US federal government in research seeking to uncover the "science of security." This interest stems, no doubt, from our increasing reliance on computational systems for everything from national defense to commerce to medicine. Computing is becoming ubiquitous and, in order to have any confidence that all this computational infrastructure is really worthy of trust, we really need to understand security from first principles. If we do not possess a science of security, then how do we distinguish trustworthy systems from untrustworthy ones?

The remainder of this section considers issues presented by a science of security and, in particular, what the term "science" might entail when combined with "security." The presentation is admittedly high level and conversational in tone. It is the authors' intention to ask questions rather than provide answers. In fact, the authors hasten to add that they have no pretensions to possessing the answers to these questions. This section contains several digressions into the history of mature scientific and engineering disciplines—chemistry and civil engineering—with the purpose of indicating what "maturity" entails for security. The authors are neither chemists, civil engineers, nor historians of science. Our purpose is to present an overview of the philosophy that motivates our research in high assurance computing.

2.1 Is a "Science" of Security really necessary?

Think about it this way. If you worked for a company that manufactured dynamite and none of the company's chemists had ever heard of Mendeleev's periodic table of elements, would you continue to work for that company? If the chemists merely worked from a recipe for dynamite, but did not understand the underlying principles of chemistry, they would not be able to distinguish safe synthesis of dynamite from unsafe synthesis. The probable results would be quite dramatic and unpleasant for all concerned.

Security as a discipline is now a collection of ideas and techniques. A popular textbook on security by Matt Bishop [14] contains thirty-five chapters on subjects ranging from access control, cryptography, security models, etc. What unites these seemingly disparate subjects? One frequently has the feeling when reading the literature of computer security that these areas are somehow connected to one another on some deep level, but the precise nature of those connections is not clear now, nor will they become so, until security is understood from first principles.

Security has been studied since the early 1970's [15], making it a decade or two younger than computing science as a whole (the umbrella term "computing science" includes all computational disciplines; e.g., computer science, computer engineering, etc.). That it has not yet evolved into a mature discipline should not be surprising when one bears in mind that older, more mature scientific and engineering disciplines (e.g., chemistry and civil engineering) became so over the course of centuries.

A historical digression provides a useful perspective and sense of proportion on the prospects for security science. Consider civil engineering, which can be said, with some justice, to be the most mature engineering discipline, having roots at least as far back as ancient Egypt. How quickly did scientific theory have an impact on civil engineering practice?

Consider the (admittedly anecdotal) example of the application of the differential and integral calculus to civil engineering. Both Newton and Leibniz published their respective versions of the calculus by 1675. Charles Augustin de Coulomb's paper *Essai sur une application des maximis règles et de minimis à quelques problèmes de statique relatifs à l'architecture*, first published in 1776, was the first publication to apply differential and integral calculus to civil engineering problems and, even then, calculus did not penetrate the civil engineering practice until the early to mid-19^{th} century [16]. Thomas Tredgold's classic treatise [17], published in 1820, formulates construction techniques in terms of trigonometry alone and does not mention erstwhile advanced ideas from physics and mathematics (e.g., statics and calculus).

Coulomb and Tredgold illustrate a parallel between the civil engineering of the early 19^{th} century and computing science as we know it today. Coulomb was a theorist who focused mainly on what we might now call mathematical physics. His approach was formal and grounded in mathematics. Tredgold was a self-taught engineer and his aforementioned treatise provides a collection of techniques for building structures (e.g., bridges, buildings, etc.) along with practical advice on the kinds of materials to be used. Tredgold wanted to know how to construct a thing and Coulomb wanted to understand why that thing's construction worked.

2.2 Foundations for the Science of Security

Tredgold and Coulomb are the perfect "poster children" for two respective and (alas) currently distinct camps within computing science: practical computer engineers and formal methods scientists. Formal methods is the application of mathematics to the design and construction of hardware and software systems. By combining design and development with rigorous mathematical analysis, formal methods seeks to construct systems that provably possess particular properties (e.g., are secure, fault-tolerant, safe, correct, etc.).

According to Parnas [18], formal methods was founded in 1967 by Robert Floyd with his paper *Assigning Meanings to Programs* [19]. Why assign meaning to a program? Because if you want to prove that a program has a particular property, you must first possess a rigorous—i.e., mathematical—definition of that program: no mathematical meaning, no mathematical proof. Similarly, if you want to know why a system S is secure, you must first possess a mathematical definition, in some form, of S. Here, "system" is interpreted very broadly; S may be a program, operating system, circuit, or, perhaps, some combination of heterogenous systems each possessing an individual mathematical definition. Having a controlled vocabulary for computational systems is a principal motivation for our use of monadic semantics [2].

Any science of security worth its salt must provide the rigorous definition of "secure" for the context of S, irrespective of the form S may take. To be worthy of the title "science", notions of security must be independent of particular, concrete instances of secure systems. This is not an exotic requirement in any sense. Arithmetic, for example, does not provide individual notions of natural numbers for each kind of object that one would count—there is one notion of natural number rather than “natural numbers for oranges” and “natural numbers for apples,” etc. Having a logical framework for specifying information flow between heterogenous subsystems motivates our interest in channel theory[3].

2.3 Challenges to a Science of Security

Formal methods has acquired a reputation in some quarters as a purely academic endeavor, where “purely academic” is generally taken to mean “unpractical.” This is not entirely fair as formal methods have made inroads into industrial practise [20], [21], [22], [23], [24], [25], but it must be recognized that formal methods are the exception and not the rule in industry [18].

David Parnas, one of the leaders in formal methods research since its inception, recently published a thought-provoking, critical assessment of the current state of the art in formal methods research [18]. While Parnas’ essay was concerned with formal methods for software, his observations are relevant to formal methods generally. Parnas identifies three negative developments in software engineering, two of which are especially relevant to the development of a science of security: the gaps between research and practice and between computing science and classical mathematics. Each gap presents challenges to be overcome in the pursuit of a science of security and below we discuss these challenges, as well as several others, in the context of a science of security.

2.3.1 Gap between Computing Science & Mathematics

Parnas observes that mathematics and theoretical computing science are separate to a surprising degree. Mathematics possesses a vast, deep body of concepts, structures and techniques that it has developed over the course of millenia. Theoretical computing science has only partaken of a tiny sampling of what mathematics has to offer. Formal methods is now at an early stage in its development where a variety of formalisms from mathematics are being investigated for their relevance to a science of computation. Formal methods may appear sometimes to produce “toys.” But the experimentation underlying formal methods research is the early part of an important intellectual exploration that must be made if computing science is to really become a science. Mathematics does not apply itself. It is hard work. Understanding Newton’s presentation of differential and integral calculus in terms of “fluxions” does not immediately suggest Coulomb’s application of calculus to calculating the strength of retaining walls. The mathematical raw material may be present in current mathematics, but fashioning that raw material into constructive engineering can take significant effort.

2.3.2 Gap between Computing Science Research & Practice

Parnas observes another gap between the Coulombs and Tredgolds in current computing science research. Formal methods research produces results that are not connected to engineering reality. Practical engineers sometimes rely on toolchains that are known to be deeply flawed, but continue to rely on them for lack of any better alternative.

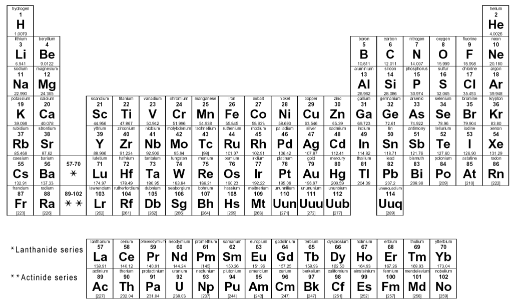
This is the problem of “legacy software” writ large. An investment in some form of intellectual capital is unlikely to be abandoned by the investor. A formal methods researcher may have invested significant time in the development of an idea or technique and will not readily abandon these cherished results even if they have not yet born fruit. A practical engineer wants to produce artifacts in the here and now and will continue to apply flawed technology to “get the job done.”

Neither the Coulombs nor the Tredgolds are being unreasonable. The formal methods researcher knows that, historically, every powerful and revolutionary idea started weak and had to be developed before its power could be recognized. Demands that the acorn become a might oak overnight are themselves unreasonable. Practical engineers cannot be expected to abandon proven, yet flawed, technology to start from scratch with each project. Nothing would ever get done.

Yet both the Coulombs and the Tredgolds must recognize that their ideas, tools and technologies may have a shorter lifespan than they imagine. And that, in the history of computing science, the short lifespan of particular ideas may ultimately be judged to be a good thing.

2.3.3 The Tower of Babel

One of the things that makes chemistry a mature science is that it has a controlled vocabulary. Chemists do not argue over how to express the molecule for water; it is simply H_2O. Neither do chemists argue over the meaning of “double bond,” “alkane” or “aromatic compound.” Chemistry’s maturity is made manifest in Mendeleev’s periodic table of elements:



The periodic table obviously does not capture all of chemical knowledge, but it does provide a foundation with which chemists can develop and communicate ideas to other chemists. Computing science in general and computer security in particular have no such organizing principles. There is a "Tower of Babel" problem in today's computing disciplines consisting a variety of ideas and techniques that, although they may seem somehow interdependent, the precise nature of their relationships is not clear. Indeed, exploring their connections does not appear to excite much interest.

The authors interest in channel theory (described below in Section 2.5) is motivated by its capability of expressing the relationships between subsystems characterized within different formalisms.

2.4 Security Theory in the Present

This section presents an example of a theory of security, Goguen and Meseguer's classic noninterference model [26], to illustrate the state of the art in security models. Noninterference may seem to be an odd choice to illustrate the state of the art since Goguen and Meseguer first published the model in 1982, but noninterference was and is enormously influential (e.g., as of this writing, it has 1145 distinct citations according to Google Scholar) and continues to inspire refinements and extensions. A manifestly incomplete list of noninterference descendants includes: Rushby [27], McCullough [28], McLean [29], Zakathinos & Lee [30]).

Noninterference is a formal model of security based on abstract state machines. It requires that, roughly speaking, low-security outputs are unaffected by high-security inputs to an abstract state representing the system. To understand noninterference, consider a system S with exactly two threads, A and B, and a trace t of a particular system execution. Trace t is a sequence of A and B operations reflecting the scheduling of the threads by S. There are three operations on S. For state σ, $\mathit{run}(t, \sigma)$ executes trace t and $\mathit{output}(\sigma, B)$ is the system output according to B. The trace $\mathit{purge}(t, A)$ is the subtrace of t with all of A's operations removed. For a given execution sequence and input state, t and σ, thread A does not interfere with B if, and only if,

$$\mathit{output}(\mathit{run}(t, \sigma), B) = \mathit{output}(\mathit{run}(\mathit{purge}(t, A), \sigma), B)$$

Most information flow security models are based either on the *noninterference* model of Goguen and Meseguer or on variants of it [31]. Such security models specify end-to-end system security policies in terms of the "views" of the system as a whole by groups of users/processes. Views are typically characterized by partitioning global system inputs and outputs and associating groups of users/processes with these partitions. These input and output partitions determine the view of its associated group. Noninterference-based security policies will require that, for example, changes in a high level security input partition will result in no change to a low level output partition.

2.5 Channel Theory and Information Security

It has long been held that information flow security models should be organized with respect to a theory of information,

$$\begin{tikzcd} & \mathsf{C} & \\ \mathsf{Hi} \arrow[ur, leftrightarrow, dotted, "h"] & & \mathsf{Lo} \arrow[ul, leftrightarrow, dotted, "l"'] \end{tikzcd}$$

Fig. 2: Channel Diagram relates component-level views (Lo and Hi) to a global view (C).
2

but typically they are not. What, for example, is the unit of information transferred in the noninterference model described above? The appeal of a information-theoretic foundation for information flow security seems natural, compelling and, indeed, almost tautological. Channel theory—a theory of information based in logic—can provide a foundation for security models based in noninterference. The first and last authors' recent publication [32] demonstrates how a McLean's hierarchy of information security policies [29] can be neatly captured in channel theory. This result suggests that channel theory is a useful organizing principle for information flow security.

The partitioning associated with noninterference security specifications implicitly divides a system into subsystems or components and defines permissible information flow in terms of the inputs and outputs resulting from component interaction. The main constructions of channel theory—classifications, infomorphisms, and channels—can be used to formulate an information-theoretic characterization of information flow security that is, in contrast to the traditional approach, explicitly component-based.

Figure 2 presents a channel diagram in which information flow between local components, Lo and Hi, is mediated by a global component, C. Individual components are called *classifications*. Classifications are local logical characterizations of components and their views and each view corresponds to an individual classification in channel-theoretic information security. In a channel diagram, information flow between component classifications occurs along *infomorphisms* (h and l) and is controlled by the channel *core* (C).

Mathematical logic, as most of us learn it [33], is not modular: there is no structural notion of logic "components" and "composition" akin to modules or classes in programming languages. The standard approach presents a logic as a monolith containing a single logical language, a single set of inference rules and a single class of structures or interpretations for the language. Channel theory [3] is a framework for formulating logical specifications in a modular (i.e., "distributed") manner.

The authors view channel theory as playing a similar role in mathematical logic as computational monads play in the semantics of programming languages with effects. Both are frameworks that support structuring specifications in a modular manner and neither is, strictly speaking, essential to their respective disciplines. That is, one can formulate language semantics without monads, just as one can formulate logical specifications without channel theory. The level of abstraction supported by monads have made monadic structuring one of the most important developments in language semantics since the late 1980's [2].

3. Monadic Programming for Secure Systems

This overview of monads and their security properties is, of necessity, extremely brief and high-level. It is presented with as few technical details as is possible so that the reader can quickly understand can quickly comprehend the gist of the approach. The risk of such a high-level presentation is that it may seem like we are describing nothing at all. The interested reader can find a thorough technical presentation in the references (especially, Harrison and Hook [34]).

3.1 Take Separation.

Goguen-Meseguer noninterference defines an abstract security model that does not include or imply any implementation strategy. We use an alternative to Goguen-Meseguer noninterference called *take separation* [34] that includes sufficient operational content to drive an implementation. Take separation is an algebraic security specification, formulated in terms of interactions of system operations or commands, and is provably equivalent to Goguen-Meseguer noninterference. Monadic information security possesses an advantage over traditional Goguen-Meseguer state machine security disciplines in that the approach relates to programs through language semantics, thus removing the need to rely upon external formalisms for program verification. One can always construct executable models obeying a take separation policy using monadic semantics [34].

Take separation is defined as follows for two processes, each of which are sequences of operations: $A = (a_0; a_1; \cdots)$ and $B = (b_0; b_1; \cdots)$. A system execution, ω, consists of any order-respecting interleaving of A and B's commands; a round-robin execution, for example, would be $\omega = (a_0; b_0; a_1; b_1; \cdots)$. To define $B\,|\!:A$, we assume the existence of function that purges B operations from a system execution, called $takeA$, and an B operation that masks out B's outputs, called $maskB$. The function $takeA$ purges B operations, leaving only A operations, and the operation $maskB$ scrubs all of B's outputs. Equations defining $takeA$ and $maskB$ are: $takeA\ n\ \omega = (a_1\ ;\ \cdots\ ;\ a_n)$, for all $n \geq 0$, and $(b\ ;\ maskB) = maskB$ for any operation b on B. Process B is *take separate* from A in ω if, and only if, $(a_1\ ;\ b_1\ ;\ \cdots\ ;\ a_n\ ;\ b_n)\ ;\ maskB = (takeA\ n\ \omega)\ ;\ maskB$ for all $n \in \{0, 1, \cdots\}$. This statement requires that A's outputs are invariant with respect to the actions of B in ω. That B's outputs are masked on both sides of the equation is important because B's outputs must be the same on both sides of the equation.

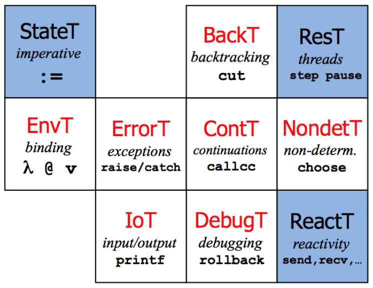


Fig. 3: The periodic table of programming languages. Each element in the table encapsulates a language constructor and molecules (i.e., individual languages) are combinations of elements. In the denotational semantics literature, "molecules" are also known as monads and elements are also known as "monad transformers" or "monad constructors."

3.2 Monadic Separation Kernels from 10,000 Feet

The best way to think of a monad is as a DSL. That is, it's just a small programming language. Monad DSLs are also modular. Bigger DSLs can be made by combining individual language building blocks together. Following MonadLab [35], an additive notation is used for composing building blocks; e.g., for blocks, b_1 and b_2, their composition is $b_1 \oplus b_2$. In the literature, DSL building blocks are usually called either monad transformers [36] or monad constructors [13]. Here's an important fact: any command from any programming language is expressible as a DSL constructor [13]. Just as Mendeleev's periodic table is a controlled vocabulary for chemistry, DSL constructors are a controlled vocabulary for defining programming languages (see Fig. 3).

The monad DSL for a Rushby-style separation kernel combines concurrency operations with one monad DSL for stateful effects per separated security domain. These "Rushby monad" DSLs come with properties by construction that are useful for verifying non-interference-based security properties. A monadic separation kernel with two domains, *Hi* and *Lo*, is defined in terms of five monad DSLs:

```
monad H  = ReactT Req Rsp ⊕ (StateT HA)
monad L  = ReactT Req Rsp ⊕ (StateT LA)
monad K  = (StateT HA) ⊕ (StateT LA)
monad R  = ResT ⊕ K
monad Re = ReactT Req Rsp ⊕ K
```

The building blocks, `(StateT HA)` and `(StateT LA)`, add commands for reading and writing from address spaces `HA` and `LA`, respectively. Note that this means that commands defined by these building blocks write to different address spaces by construction. The building block, `ReactT Req Rsp`, adds commands for interacting with the kernel. User processes running on either *Hi* and *Lo* can send a trap to the kernel of type `Req`. The kernel may then signal a response of type Rsp. The building block ResT allows threads to be separated into time slices and executed by the kernel.

The monad DSLs H and L define the languages of user processes on *Hi* and *Lo*. This means that *Hi* (*Lo*) code can access the HA (LA) address space as well as signal the kernel. By construction, *Hi* and *Lo* processes can only interact, if permitted, via the kernel, because their commands cannot access each other's address spaces. The monad DSL for the separation kernel is defined by the three monads DSLs K, Re and R. The kernel has access to all of HA and LA via K. The kernel is a tail-recursive function that takes a user program (i.e., from either an H or L), services any request it may have signaled, and dispatches a time slice of the use r process (an R object). Each monad DSL has a sequencing operation, », that corresponds to ";" in languages like C or Java.

By construction, we know a number of properties about programs in the monad DSLs H and L. Each has a special operator, zeroH and zeroL, that initializes the address spaces HA and LA, respectively. For any commands hi and lo, from (StateT HA) and (StateT LA), respectively, we know that a number of equations hold when executed by the kernel, including:

```
1. hi >> lo    = lo >> hi
2. lo >> zeroH = zeroH >> lo
3. hi >> zeroH = zeroH
```

The most important of these properties from the point of view of security is 1. (called *atomic noninterference* [34]). Because any such hi and lo commute, they are independent of one another. Property 2. says that lo and zeroL are also independent. But property 3. shows that zeroH cancels hi. We also know that » is associative.

A trace of a system execution is expressed in the R DSL, meaning that the signals have already been serviced. The result of the kernel servicing a process request is reflected by an operation on that process's address space (e.g., copying the content from a return register into a local address). A prefix of a trace will contain interleaved operations from each domain; e.g., a prefix might have the form: $\mathtt{lo}_0$ » $\mathtt{hi}_0$ » $\mathtt{lo}_1$. For such a trace prefix, cancel out all of the *Hi* operations with zeroH:

$$\begin{aligned}
&(\mathtt{lo}_0 \gg \mathtt{hi}_0 \gg \mathtt{lo}_1) \gg \mathtt{zeroH} \\
&\quad = \mathtt{lo}_0 \gg \mathtt{hi}_0 \gg \mathtt{lo}_1 \gg \mathtt{zeroH} && \{\ \text{assoc.}\ \} \\
&\quad = \mathtt{lo}_0 \gg \mathtt{hi}_0 \gg \mathtt{zeroH} \gg \mathtt{lo}_1 && \{\ 2.\ \} \\
&\quad = \mathtt{lo}_0 \gg \mathtt{zeroH} \gg \mathtt{lo}_1 && \{\ 3.\ \} \\
&\quad = \mathtt{lo}_0 \gg \mathtt{lo}_1 \gg \mathtt{zeroH} && \{\ 2.\ \} \\
&\quad = (\mathtt{lo}_0 \gg \mathtt{lo}_1) \gg \mathtt{zeroH} && \{\ \text{assoc.}\ \}
\end{aligned}$$

One will observe that this is precisely what is required by take separation. The zero operations defined by the StateT building blocks play the same role as *purge* does for the Goguen-Meseguer model.

The proof above shows that, as long as the kernel does not allow an illegal information flow, then the whole system has take separation. This follows immediately by the construction of the monadic DSLs above. The kernel must still be verified. Because it can communicates with each domain, a kernel could break the security policy. The good news is that the kernel description is short; for an example, see Fig. 5, which is explained below in Section 4.1.

3.3 The State Monad

This section presents the definition of a monad in Haskell for the sake of completeness. This section is not required to understand the rest of the paper.

Monads are typically represented in functional programming languages like Haskell [10]. The representation of S in a Haskell-like notation consists of the function and type declarations in Figure 4. The monadic type S a is a function type taking an input state of type Store to a product of outputs of type (a×Store) representing the output value and state respectively. Lambda notation is used to express functions in functional languages; e.g., $\lambda\mathtt{x}.\mathtt{x}+1$ is a function that takes an integer x as input and returns its increment as output. In x»y, the null bind operator (») threads an input state s through x, producing output state s′, that is then threaded through y.

The DSL for the state monad S is displayed in Figure 4 for some given some type Store. The command, (return 99) : S Int, is a trivial computation that returns the value 99. The command, get : S Store, reads and returns the current Store. Neither return nor get change the current Store state. Given a function f : Store→Store, (update f) transforms the current state by applying f to it; it produces a nil value of type (). The bind commands, »= and », sequence S-computations together analogously to ";" in C or Java; we will only use the so-called null bind (») in this paper.

```
return : a → S a
(>>=)  : (S a) → (a → S b) → S b
(>>)   : S a → S b → S b
update : (Store→Store) → S ()
get    : S Store
```

```
S a      = Store→(a×Store)
x >> y   = λs.let (d,s′) = x s in y s′
update f = λs.((),f s)
return v = λs.(v,s)
```

Fig. 4: The State Monad S (top) and its representation in a functional programming language (bottom).

```
-- DSL Construction:
type Hi   = Addr -> Int
type Lo   = Addr -> Int
data Req = Cont | Bcst Int | Rcv
data Rsp = Ack  | Rcvd Int
monad K   = (StateT Hi) ⊕ (StateT Lo)
monad R   = ResT ⊕ K
monad Re = (ReactT Req Rsp) ⊕ K

-- Kernel-level  Data
type Sys = (Q(Re()),Q Int,Q Int)
data HdlrOp a  = Halt | ServH a | ServL a
type Handshake = (Req,Rsp -> K (Re ()))

resched :: Sys -> Re () -> R ()
resched (ts,l,h) t  = kl (ts << t,l,h)
```

```
kl :: Sys -> R ()
kl (∅,_,_)    = return ()
kl (t◇ts,l,h) = disp (onsig t)
 where
   disp :: HdlrOp Handshake -> R ()
   disp Halt                = kl (ts,l,h)
   disp (ServH (Cont,k))    = stepH (k Ack) >>= resched (ts,l,h)
   disp (ServL (Cont,k))    = stepL (k Ack) >>= resched (ts,l,h)
   disp (ServH (Bcst m,k)) = kl (ts << rspdH k Ack,l,h << m)
   disp (ServL (Bcst m,k)) = kl (ts << rspdL k Ack,l<<m,h<<m)
   disp (ServH (Rcv,k))    = case h of
                ∅      -> kl (ts << t,l,∅)
                (m◇hs) -> kl (ts << rspdH k (Rcvd m),l,hs)
   disp (ServL (Rcv,k))    = case l of
                ∅      -> kl (ts << t,∅,h)
                (m◇ls) -> kl (ts << rspdL k (Rcvd m),ls,h)
```

Fig. 5: A separation kernel with two domains, H and L, written in HASK. Threads communicate via asynchronous broadcast & receive system calls. Note that messages broadcasted on L reach H but not vice versa. The unit and bind operations, return and »=, are overloaded for monads K, R, and Re. The thread operators (onsig, step, rspd) are discussed in Section 4.1.

4. HASK Language Design & Implementation

There were a number of issues we confronted and design decisions we made in the design of HASK. HASK is a pure functional language like the Haskell language, albeit with a number of strong constraints. The original experiments with monad-based security kernels were rendered in the Haskell functional programming language, but, for a number of reasons relating to its complicated semantics and implementation, Haskell was determined to be an unsuitable source language. HASK has been made essentially simpler than Haskell by introducing and enforcing a number of restrictions on its expressiveness.

HASK is a first-order language, meaning that functions are never proper values as they are in a higher-order language like Haskell. The rationale behind this somewhat unusual choice (unusual, but not unprecedented [37]) is that we did not want to include language constructs that would necessitate complex implementation strategies. As the resumption monadic paradigm developed in previous publications are all essentially first-order, HASK's economy of expressiveness is not a shortcoming and ultimately facilitates our implementation and verification efforts. Implementations of HASK programs are kernels and require the same basic runtime structures and characteristics; these are bounded usage of memory, loop-structured control flow and asynchronous interrupt handling. HASK disallows general recursion in favor of tail recursion.

Monads are first-class in HASK by which it is that there is a declaration form "**monad** $M = L_1 \oplus \cdots \oplus L_n$" in HASK for defining the monad DSL, M. The monad layer (transformer) expressions, L_i, determine which commands are present in M. Monad declarations of this form are an integral part of HASK syntax in that they define the operations of the Rushby monad in which a kernel may the be specified. This part of the language design is discussed in detail in a recent publication [35].

4.1 HASK by Example

This section gives an introduction to the HASK language via an a small kernel (see Fig. 5). The monad DSL for this kernel is constructed along the same lines as the monad DSLs from Section 3 in the upper left column. The address spaces, Hi and Lo, are maps from addresses to Int. The request type, Req, has requests for a broadcast message service as well as a request to receive a message. System responses, Rsp, has responses of the form (Rcvd i) which send the data i back to a process. The Cont and Ack request and response are NOPs. The languages of user processes (not shown) are defined as they are in Section 3.

HASK has a built-in queues captured as the type constructor, Q. Type of queues holding values of type a are written, Q a. The queue Q a has constructors for the empty queue and item insertion; these are ∅ :: Q a and («) :: Q a -> a -> Q a, resp. The head and tail of a queue q are accessed via pattern-matching within a **case** expression:

```
case q of
      ∅       -> e1
      (h◇hs)  -> e2
```

Expression e1 is only evaluated when q is empty. The head (h) and tail (hs) of q may be accessed within e2.

Using the queue abstraction, we define the kernel-level data, Sys (line 6 of Fig. 5). A Sys value is a tuple, (ts,l,h), consisting of a ready thread queue (ts), a Lo-domain message queue (l), and a Hi-domain message queue, (h). The handling of these message queues is critical to the kernel maintaining the domain separation policy.

The kernel itself is encapsulated by the mutually tail-recursive functions, `kl :: Sys -> R ()` and `disp :: HdlrOp HandShake -> R ()`. The `kl` picks a process `t` from the queue, if it exists, and runs it by passing it to `disp`. The dispatch function, `disp`, processes any request signal, executes the slice, and passes control back to `kl`. The security of the kernel is maintained by `disp`. If a high process broadcasts a message, it only affects the high security message queue. If a low security process requests a message, then it only receives those from the low security queue. The kernel is the only place where insecure flows could arise in this HASK kernel, so this kernel is secure. Verification of similar kernels is found in the references [34].

```
var channel : int := 0

handler 1 {
  if pid==0
    then switch(1)
    else switch(0)
  fi
}

handler trap {
  channel := R[5] ; switch(pid)
}
```

Fig. 6: A Simple ETIC Kernel

4.2 HASK Language Implementation

There are two implementation strategies for monadic models of take separation [38]. The first compiles models into three address code using a traditional compiler approach. The second automatically transforms models into abstract state machines via a program transformation known as defunctionalization [39].

The high-level architecture of a monad compiler is conventional, consisting of an intermediate code generation phase translating the source program (i.e., a program written in monadic style) to an appropriate intermediate representation (IR) and a phase translating the IR into an appropriate instruction set. Traditional language compilers also include considerable analysis and optimization in the compiler "back-end", which our monad compiler prototype currently does not perform.

Aside from the source language, the novelty in monadic compilers lies both in the process by which code generation is performed and in the form of the IR itself. This process of code generation involves the application of a classic semantics-preserving program transformation called *defunctionalization* [39] to ICG. This transformation provides a means by which to compile higher order functions to state machines which, in turn, are readily compiled to hardware. These abstract state machines may then be translated into VHDL using FSMLang [40].

4.3 Extended Typed Interrupt Calculus

The techniques described in the preceding sections achieve security by construction: the resumption-monad kernel model inherently excludes storage channels between processes, except when such channels are actively mediated by the kernel. Stock hardware typically does not provide such strong guarantees. While a typical CPU architecture provides *mechanisms* to enforce information flow policies (in the form of an MMU), these mechanisms do not provide inherent *guarantees* of security: the operating system still may grant low-security processes access to high-security data, in violation of the security policy. We address this discrepancy by introducing an intermediate language called the *Enhanced Typed Interrupt Calculus* (ETIC), patterned after the Typed Interrupt Calculus developed by Palsberg and Ma [41], [42]. ETIC may be viewed as a domain-specific language for OS kernels. It is a simple imperative language, enriched with primitives for interfacing with interrupts and hardware-enforced memory protection. It has a precise semantics, suitable for reasoning about security properties, yet may also be compiled to kernel-level code running on real hardware in a straightforward way.

4.3.1 Overview of ETIC

Figure 6 is a simple example of an ETIC kernel. Consider a machine consisting of one processor with a single interrupt request line (numbered 1), connected to a periodic timer. A pre-emptive multitasking kernel for such a machine may be implemented in ETIC simply by declaring a single interrupt handler for the timer, which invokes a scheduler terminating in a *switch* statement, and a trap handler for system calls. The kernel in Figure 6 assumes there are only two processes, numbered 0 and 1, and a single system call that allows a process to write a value (stored in register 5) to a single shared storage location.

A partial grammar for core ETIC is given in Figure 7. An ETIC kernel consists of a set of *handler* declarations, one for each interrupt request line and one for software traps. The language of commands consists of conditional statements, loops, assignment, sequencing, and three special primitives for manipulating the page table and executing a context switch to a user process.

4.3.2 Semantics

The semantics of ETIC is defined in a standard operational style in Figure 8; i.e. as a set of state transition rules. Figure 8 gives the most important rules. We assume that a semantics for single processes, whose state consists of registers R and memory M, is defined by a transition relation $\xrightarrow{U}$. On top of this we define a "machine-level" semantics, representing a machine with multiple user process and a kernel. Machine state is represented by a tuple consisting of an execution mode m (which is either U for user mode or K for kernel

$$
\begin{array}{rcl}
\textit{Prog} & ::= & \textit{Handler}^{*} \\
\textit{Handler} & ::= & \text{handler } \textit{IRQ} \ \{ \ \textit{Cmd} \ \} \\
\textit{IRQ} & ::= & 1 \mid 2 \mid \cdots \mid n \mid \text{trap} \\
\textit{Cmd} & ::= & \text{if } \textit{Expr} \text{ then } \textit{Cmd} \text{ else } \textit{Cmd} \\
 & \mid & \textit{Lhs} := \textit{Expr} \\
 & \mid & \textit{Cmd} \ ; \ \textit{Cmd} \\
 & \mid & \text{map}(\textit{Expr}, \textit{Expr}, \textit{Expr}) \\
 & \mid & \text{unmap}(\textit{Expr}, \textit{Expr}) \\
 & \mid & \text{switch}(\textit{Expr}) \\
\textit{Expr} & ::= & \textit{Literal} \mid \textit{Expr Binop Expr} \mid \textit{Variable} \mid R_i \\
\textit{Lhs} & ::= & \textit{Variable} \mid R_i
\end{array}
$$

Fig. 7: Grammar for Core ETIC

mode), the kernel command stream C, internal kernel state S, saved process registers R, user-process heap M, saved process identifier P, page table T, and finally the code for the interrupt handlers H (the interrupt handler for IRQ i is denoted by H_i).

Each rule in Figure 8 consists of a (possibly vacuous) assumption above the horizontal line, and a conclusion below it. We "lift" the single-process semantics into the machine semantics via rule LIFT: essentially, the single-process semantics determines what the system will do when it is in user mode. Assume that R is the user process registers, $\hat{M}$ is process P's view of the machine memory in page table T, and applying the single-process transition relation to $\langle R, \hat{M} \rangle$ results in registers R' and memory $\hat{M}'$. Then when process P is running the system may transition to a new state with registers R' and memory M', where M' results from "merging" the changes from $\hat{M}$ to $\hat{M}'$ into M according to process P's view of the memory. Rule INTR expresses the semantics for interrupts: whenever a user process is running, the system may transition to kernel mode and begin executing the code for an interrupt handler. Note that more than one transition rule may apply to any given machine state; in particular, both INTR and LIFT apply anytime a user process is running. This possibilistic nondeterminism reflects the assumption that interrupts may occur (or not occur!) at any time.

The transition rules also determine the behavior of the commands defined in Figure 7. The rule SWITCH says that executing the switch command overwrites the value in the system process ID register, and returns to user mode. The map command allows the kernel to update the page table by adding a mapping of PID-page pairs to physical memory addresses. Note that the rules for these commands only apply when the system is in kernel mode. The rules for the other kernel commands are similar and so are omitted here.

5. Related Work

A calculational approach to program construction [43] starts from a high-level specification and produces an implementation through a series of verifiable transformations. The motivation behind the calculational approach is that the formal connections between source specifications and their implementations support formal verification. The calculational approach to compiler construction [44], [45], [46], [47] follows this paradigm by transforming a high-level language semantics into compiler implementation.

Monadic semantics have played a role in other recent high assurance systems research as well. State-monadic semantics have been used recently for specifying ARM and x86 instruction sets [48], [49]. The design, construction and verification of a secure microkernel is described by Klein et al. [50], [51], [52]. This kernel—called seL4—is a version of the L4 microkernel [53] with security guarantees. Monads are used in the modeling phase and are implemented by hand translation into C. Monads have also play a role in recent information security models [54], [55]. Monads can be used as a scoping mechanism for side effects and this is central to the authors' approach to secure system construction. The security model advocated by the authors uses structural (i.e., "by construction") properties of monads as a central organizing principle of their approach, whereas the aforementioned models use monads more as a security level tag.

Monadic programs are typically implemented in higher order functional languages although, in certain cases, the notion of computation encapsulated by a monad could be more efficiently implemented directly. Monads of resumptions and state can be implemented efficiently via translation into simple imperative loop code [38]. The present approach relies on *monad compilation* to produce code rather than partial evaluation (as does pass separation). Monad compilation treats terms typed in a particular monad as a programming language unto itself. The syntactic monad-as-language treatment is not new: Hughes applied it in the development of a pretty-printing library [56] as did Hinze [57] in the derivation of backtracking monad transformers. Direct compilation of monads has also been explored by Danvy et al. [58] and Filinski [59].

Model-driven development (MDD) [60], [61] is a software engineering paradigm that has sparked considerable interest of late. Semantics-directed compilation is a form of MDD that has a long history within programming languages research. In the present work, the "models" are constructed with monad transformers. Such models have a dual nature: they are both denotational models (supporting formal reasoning) and operational models that may be executed.

Channel theory is closely related to Chu spaces, information systems and institutions. In the categorical view of channel theory, the objects are the same as those in Chu spaces [62], but the arrows are different. In Chu spaces, no

$$\frac{\langle R, \hat{M}\rangle \xrightarrow{U} \langle R', \hat{M}'\rangle \quad \hat{M} = \pi_{P,T}(M) \quad M' = \iota_{P,T}(M, \hat{M}, \hat{M}')}{\langle U, C, S, R, M, P, T, H\rangle \to \langle U, C, S, R', M', P, T, H\rangle} \ (\textsc{Lift})$$

$$\frac{}{\langle U, C, S, R, M, P, T, H\rangle \to \langle K, H_i, S, R, M, P, T, H\rangle} \ (\textsc{Intr})$$

$$\frac{}{\langle K, \text{switch}(e) :: C, S, R, M, P, T, H\rangle \to \langle U, C, S, R, M, [\![e]\!]_S, T, H\rangle} \ (\textsc{Switch})$$

$$\frac{}{\langle K, \text{map}(e_1, e_2, e_3) :: C, S, R, M, P, T, H\rangle \to \langle K, C, S, R, M, P, T[\langle [\![e_1]\!]_S, [\![e_2]\!]_S\rangle \mapsto [\![e_3]\!]_S], H\rangle} \ (\textsc{Map})$$

Fig. 8: Core ETIC Semantics

mention is made of the theory of a classification and most of the research appears to be directed at their categorical structure. By comparison, the categorical structure in channel theory, while useful, is not of primary importance. Scott's information systems [63] use similar structures specialized to computational domains. There is an extensive literature on institutions [64]. The aforementioned reference combines institutions with the work of Barwise and Seligman. An institution is a functor from an abstract category to the category of classifications.

6. Conclusion

To construct an embedded kernel using current technologies, how would you proceed? You would choose a target architecture for which one would presumably also have a C compiler and an assembler. Assume that the functionality that the kernel is to support is also known and has been provided to you. Let us say further that the kernel is to enforce some security policy like domain separation. With enough effort, you produce a kernel implementation in C with some additional assembly language routines. Now, the kernel is stress tested enough that you have become fairly confident that it works as intended. Now, you present this kernel to your boss, who demands that you justify your confidence that the kernel is indeed secure.

Your boss asks, "how do you know that high security information doesn't leak?" You answer, "Here are the lines of C in the kernel that enforce security." Puzzled and mildly irritated to have to look at code, your boss continues, "OK, all that gobble-de-gook code seems reasonable, but how do you know that the C compiler doesn't mess it all up somehow?" He never brought this up before, you think to yourself. The C compiler you used has over 5 million lines of code, so trying to validate that it doesn't somehow undermine your security mechanisms is not tractable. "Also," your boss continues, "there are all of these insidious code injection attacks with the architecture you chose, how does your kernel cope with them?" Come to think of it, precisely what did you need from the underlying hardware to build a secure system?

Like it or not, your boss's questions raise important issues. What sort of support do you need from hardware to build a secure system? Even if a software system is deemed secure, what are the underlying assumptions about the hardware that support this claim?

Security is an end-to-end concern. Even if we verify that a program obeys a security policy, there are still parts of the underlying system—e.g., the compiler, operating system and hardware—that could undermine our program's security when executed and this underlying system is almost certainly not built with your security policy in mind. Security must be taken into account through all phases of the system design and implementation—especially when the system security is to be formally verified. But, current toolchains are not designed with this end-to-end flow in mind, and no amount of pounding will drive the security square peg into the round hole of current methodologies. A true science of security will require radical intellectual retooling of how we design, construct and verify systems.

Functional languages are often heralded as a good foundation for high assurance computation, and we agree with this—as far as it goes. Languages like ML and Haskell do contain core languages that have been given rigorous semantics and this is critical to high assurance and formal verification. For certain applications and certain required levels of assurance, these languages may indeed provide sufficient frameworks for designing, verifying and construction of high assurance systems. However, there are applications—security microkernels in particular—where off-the-shelf general purpose functional languages do not suffice. For one thing, their semantically specified core languages do not tackle the Awkward Squad and this, in itself, makes high assurance development of system-level code problematic at best. For another, their compilers and run-time systems were not designed to be verified and, as a result, are every bit as difficult to verify as any other large compiler. Finally, their implementations have behaviors and functionalities that can also be problematic for system programming; for example, their executables may be too large for embedded systems and garbage collection can create space leaks and unpredictable timing behavior.

HASK retains only the parts of Haskell that we need while jettisoning the rest. Strong typing, pattern-matching, and functions are all beneficial. The Haskell type system can be extended or refined with security types [65] and this approach can serve to simplify the verification process. We have specified much of the HASK language and its infrastructure in the Coq theorem proving system [66] and are currently investigating machine-checked verification of HASK. Following the domain-specific language design approach with HASK is key to making HASK verification tractable.

What most distinguishes this research from that of others applying functional languages [11], [12] is our programming model. While the HASK language is, in fact, functional, the design of the language follows from a consideration of the features strictly necessary to express essential computational effects that are otherwise typically considered the domain of low-level imperative languages such as C. Because monads provide a precise semantic basis for the "Awkward Squad", we choose them as the basis of our approach.

Acknowledgment

This research was supported by NSF CAREER Award 00017806, US Naval Research Laboratory Contract 1302-08-015S, and the Gilliom Cyber Security Gift Fund.

References

[1] J. Rushby, "Design and verification of secure systems," in *Proceedings of the ACM Symposium on Operating System Principles*, vol. 15, 1981, pp. 12–21.

[2] E. Moggi, "Notions of computation and monads," *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, 1991.

[3] J. Barwise and J. Seligman, *Information Flow: The Logic of Distributed Systems*. Cambridge University Press, 1997, cambridge Tracts in Theoretical Computer Science 44.

[4] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (revised)*. Cambridge, MA: MIT Press, 1997.

[5] J. C. Mitchell and R. Harper, "The essence of ML," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '88, 1988, pp. 28–46.

[6] W. L. Harrison and R. B. Kieburtz, "The logic of demand in haskell," *Journal of Functional Programming*, vol. 15, no. 5, pp. 837–891, 2005.

[7] W. Harrison, "A simple semantics for polymorphic recursion," in *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS05)*, 2005, pp. 37–51.

[8] W. Harrison, T. Sheard, and J. Hook, "Fine control of demand in Haskell," in *6th International Conference on the Mathematics of Program Construction, Dagstuhl, Germany*, ser. Lecture Notes in Computer Science, vol. 2386. Springer-Verlag, 2002, pp. 68–93.

[9] S. Peyton Jones, "Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell," in *Engineering Theories of Software Construction*, ser. NATO Science Series. IOS Press, 2000, vol. III 180, pp. 47–96.

[10] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003.

[11] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach, "A principled approach to operating system construction in Haskell," in *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP05)*. New York, NY, USA: ACM Press, 2005, pp. 116–128.

[12] P. Li and S. Zdancewic, "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2007, pp. 189–199.

[13] E. Moggi, "An Abstract View of Programming Languages," Department of Computer Science, Edinburgh University, Tech. Rep. ECS-LFCS-90-113, 1990.

[14] M. Bishop, *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.

[15] ——. (2011, Apr.) Computer security archives project. [Online]. Available: http://seclab.cs.ucdavis.edu/projects/history/

[16] Karl-Eugen Kurrer, *The History of the Theory of Structures: From Arch Analysis to Computational Mechanics*. John Wiley & Sons, 2008.

[17] T. Tredgold, *Elementary Principles of Carpentry; A Treatise on the Pressure and Equilibrium of Timber Framing; The Resistance of Timber; and the Construction of Floors, Roofs, Centres, Bridges, etc.*, 1820.

[18] D. L. Parnas, "Really rethinking 'formal methods'," *IEEE Computer*, vol. 43, pp. 28–34, 2010.

[19] R. W. Floyd, "Assigning Meanings to Programs," in *Proceedings of a Symposium on Applied Mathematics*, ser. Mathematical Aspects of Computer Science, J. T. Schwartz, Ed., vol. 19. American Mathematical Society, 1967, pp. 19–31.

[20] *Software Assurance*. IEEE Computer, September 2010.

[21] A. Hall, "Seven myths of formal methods," *IEEE Software*, vol. 7, pp. 11–19, September 1990.

[22] W. Gibbs, "Software's chronic crisis," *Scientific American*, pp. 86–95, Sep. 1994.

[23] J. P. Bowen and M. G. Hinchey, "Seven more myths of formal methods," *IEEE Software*, vol. 12, pp. 34–41, July 1995.

[24] G. J. Holzmann, "Proving the value of formal methods," in *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, 1995, pp. 385–396.

[25] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria, "Software engineering and formal methods," *Communications of the ACM*, vol. 51, pp. 54–59, September 2008.

[26] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the 1982 IEEE Symposium on Security and Privacy*. IEEE Press, 1982, pp. 11–20.

[27] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Tech. Rep. CSL-92-02, December 1992.

[28] D. McCullough, "A hookup theorem for multilevel security," *IEEE Trans. Softw. Eng.*, vol. 16, no. 6, pp. 563–568, 1990.

[29] J. McLean, "A general theory of composition for a class of "possibilistic" properties," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 53–67, 1996.

[30] A. Zakinthinos and E. Lee, "A general theory of security properties," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1997, pp. 94–102.

[31] S. Zdancewic, "Challenges for information-flow security," in *Proceedings of the First International Workshop on Programming Language Interference and Dependence (PLID'04)*, 2004.

[32] G. Allwein and W. L. Harrison, "Partially-ordered modalities," in *Advances in Modal Logic*, vol. 8, 2010, pp. 1–21.

[33] E. Mendelson, *Introduction to Mathematical Logic*. Van Nostrand Reinhold Company, 1997.

[34] W. L. Harrison and J. Hook, "Achieving information flow security through monadic control of effects," *J. Comput. Secur.*, vol. 17, pp. 599–653, October 2009.

[35] P. Kariotis, A. Procter, and W. Harrison, "Making monads first-class with Template Haskell," in *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell (Haskell08)*, 2008, pp. 99–110.

[36] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1995, pp. 333–343.

[37] J. Goguen, "Higher-order functions considered unnecessary for higher-order programming," pp. 309–351, 1990.

[38] W. Harrison, A. Procter, J. Agron, G. Kimmel, and G. Allwein, "Model-driven engineering from modular monadic semantics: Implementation techniques targeting hardware and software," in *Proceedings of the IFIP Working Conference on Domain Specific Languages (DSL09)*. Berlin, Heidelberg: Springer-Verlag, July 2009, pp. 20–44.

[39] M. S. Ager, O. Danvy, and J. Midtgaard, "A functional correspondence between monadic evaluators and abstract machines for languages with computational effects." *Theor. Comput. Sci.*, vol. 342, no. 1, pp. 149–172, 2005.

[40] J. Agron, "Domain-specific language for hw/sw co-design for FPGAs," in *Proceedings of the IFIP Working Conference on Domain Specific Languages (DSL09)*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 262–284.

[41] J. Palsberg and D. Ma, "A typed interrupt calculus," in *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. London, UK: Springer-Verlag, 2002, pp. 291–310.

[42] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. A. Henzinger, and J. Palsberg, "Stack size analysis for interrupt-driven programs," *Inf. Comput.*, vol. 194, no. 2, pp. 144–174, 2004.

[43] R. Backhouse, *Program Construction: Calculating Implementations from Specifications*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[44] A. Igarashi and M. Iwaki, "Deriving compilers and virtual machines for a multi-level language," in *Proc. of the 5th Asian conference on Programming languages and Systems*, 2007, pp. 206–221.

[45] E. Meijer, "Calculating compilers," Ph.D. dissertation, University of Nijmegen, 1992.

[46] G. Hutton and J. Wright, "Calculating an exceptional machine," in *Trends in Functional Programming*, H.-W. Loidl, Ed., Feb. 2006, vol. 5.

[47] O. Danvy, "A journey from interpreters to compilers and virtual machines," in *Proc. of the 2nd Intl. Conf. on Gener. Prog. and Comp. Eng.*, 2003, pp. 117–117.

[48] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-cc multiprocessor machine code," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '09. New York, NY, USA: ACM, 2009, pp. 379–391.

[49] A. C. J. Fox and M. O. Myreen, "A trustworthy monadic formalization of the armv7 instruction set architecture," in *Interactive Theorem Proving (ITP)*, 2010, pp. 243–258.

[50] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, pp. 107–115, June 2010.

[51] D. Cock, G. Klein, and T. Sewell, "Secure microkernels, state monads and scalable refinement," in *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, ser. Lecture Notes in Computer Science, O. A. Mohamed, C. M. noz, and S. Tahar, Eds., vol. 5170. Springer-Verlag, 2008, pp. 167–182.

[52] K. Elphinstone, G. Klein, and R. Kolanski, "Formalising a high-performance microkernel," in *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06), Microsoft Research Technical Report MSR-TR2006-117*, 2006, pp. 1–7.

[53] J. Liedtke, "On micro-kernel construction," in *Symposium on Operating System Principles*. ACM, 1995.

[54] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A core calculus of dependency," in *Proceedings of the Twenty-sixth ACM Symposium on Principles of Programming Languages, San Antonio, Texas*, January 1999, pp. 147–160.

[55] K. Crary, A. Kliger, and F. Pfenning, "A monadic analysis of information flow security with mutable state," *Journal of Functional Programming*, vol. 15, no. 2, Mar. 2005.

[56] J. Hughes, "The Design of a Pretty-printing Library," in *Advanced Functional Programming*, ser. LNCS, vol. 925, 1995, pp. 53–96.

[57] R. Hinze, "Deriving backtracking monad transformers," in *International Conference on Functional Programming*, 2000, pp. 186–197. [Online]. Available: citeseer.nj.nec.com/hinze00deriving.html

[58] O. Danvy, J. Koslowski, and K. Malmkjaer, "Compiling monads," Kansas State University, Manhattan, Kansas, Tech. Rep. CIS-92-3, Dec. 91. [Online]. Available: ftp://ftp.diku.dk/pub/diku/semantics/papers/D-154.dvi.Z

[59] A. Filinski, "Representing layered monads," in *Proceedings of the 26st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM Press, 1999, pp. 175–188.

[60] D. Batory, "Program refactoring, program synthesis, and model-driven development." in *ETAPS Conference on Compiler Construction*, ser. LNCS, vol. 4420, April 2007, pp. 156–171.

[61] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, p. 25, 2006.

[62] M. Barr, "$*$-autonomous categories and linear logic," *Mathematical Structures Computer Science*, vol. 1, pp. 159–178, 1991.

[63] D. S. Scott, "Domains for denotational semantics," in *An extended version of the paper prepared for ICALP '82*. Springer–Verlag, 1982.

[64] J. Goguen, "Information integration in institutions," in *Thinking Logically: a Memorial Volume for Jon Barwise*, L. Moss, Ed. Indiana University Press, 200x.

[65] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, Jan. 2003.

[66] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. Springer Verlag, 2004.

SESSION

REGULAR SESSION - SECURITY: THREATS AND SOLUTIONS

Chair(s)

PROF. RYAN KASTNER

INVITED TALKS

Enforcing Information Flow Guarantees in Reconfigurable Systems with Mix-trusted IP

Ryan Kastner[‡], Jason Oberg[‡], Wei Hu[†,‡], Ali Irturk[‡]
[‡]Computer Science and Engineering, University of California, San Diego
[†]Automation, Northwestern Polytechnical University, Xi'an, China
{kastner,jkoberg,w3hu,airturk}@cs.ucsd.edu

***Abstract*—Trusted systems fundamentally rely on the ability to tightly control the flow of information both in-to and out-of the device. Due to their inherent programmability, reconfigurable systems are riddled with security holes (timing channels, undefined behaviors, storage channels, backdoors) which can be used as a foothold for attackers to strike. System designers are constantly forced to respond to these attacks, often only after significant damage has been inflicted. We propose to use the reconfigurable nature of the system to our advantage by taking a bottom-up, hardware based approach to security. Using an information flow secure hardware foundation, which can precisely verify all information flows from Boolean gates, security can be verified all the way up the system stack. This can be used to ensure private keys are never leaked (for secrecy), and that untrusted information will not be used in the making of critical decisions (for safety and fault tolerance).**

I. Introduction

Reconfigurable hardware frequently finds itself in charge of high-assurance applications such as flight control and medical systems. As these reconfigurable systems increase in design complexity, commercial off the shelf (COTS) intellectual property (IP) cores are becoming a commodity in these systems. Ensuring that the mix-trust in these systems does not violate the integrity or confidentiality of the system as a whole is required for its trusted operation. Such assurance often requires detailed verification including strict theorem proving and third party analysis [1]. This complicated process not only takes a tremendous amount of time estimated at 10 years [3] but also costs on the order of $10,000 per line of code [2]. Reducing this overhead is needed to keep these operation critical systems up-to-date with current technology at a reasonable cost.

An example of a system in which mix-trusted components interact can be found in a modern aircraft where a shared physical bus multiplexes between mix-trusted subsystems [8]. For example, if a bus arbitrates between user and flight control systems, unintended information flowing from the user to flight control system could have catastrophic consequences. It is absolutely required that untrusted information never corrupts trusted flight control components. If such connectivity is to be allowed, strict guarantees of information flow control are necessary to the correct and reliable operation of the aircraft.

To guarantee mix-trusted information flows only to where the system designer intends, information flow tracking (IFT) has been introduced. IFT works by monitoring the movement of data as it propagates through the system. Information flow tracking can be used to ensure that a secret key does not leak, in the context of Bell & LaPadula confidentiality [7] or to guarantee the integrity of trusted components, in the case of non-interference [6]. In general, there are two classes of information flows: *explicit* and *implicit*. Explicit information flows result from direct communication. For example, an explicit flow would occur between a host and device on a bus that were directly exchanging data or between processes over an inter-processes communication (IPC) mechanism. Implicit information flows are much more subtle and generally leak information through behavior. A common implicit information flow that occurs in hardware is a timing channel where information can be extracted from the latency of operations.

To account for these difficult to detect security holes, current methods are lacking in that they either perform physical isolation or "clock fuzzing" [14], [25]. Physical isolation works by separating trusted/untrusted or classified/unclassified subsystems from one another and allowing interconnect only at statically defined locations. This method is often effective although it tends to result in large area overheads since typical implementations require the replication of hardware. "Clock fuzzing" makes attempts to avoid physical isolation by presenting untrusted subsystems with a "fuzzed" clock that produces artificial errors in timing information. This tries to reduce the ability to gain information from timing channels but in reality only decreases the bandwidth of the channel. A more robust solution is to eliminate all such information flows and verify their absence; this can currently be done using a technique called gate level information flow tracking.

Gate Level Information Flow Tracking (GLIFT) [26] provides a solution for tracking information flows in hardware. Since GLIFT tracks information at the granularity of Boolean gates, it can be used on any digital hardware. Furthermore, it is capable of detecting logical flows including those through timing channels because all information becomes explicit in hardware. This does, however, focus only on logical flows and excludes physical phenomena such as power channels and EM radiation. This bottom-up approach to security can be seen in Figure 1 where a secure hardware foundation is created to ensure information flow security at the lowest abstraction level. With this secure hardware base, microarchitectural designs can be implemented with strict information flow guarantees.

This paper discusses the concerns with using mix-trusted IP in reconfigurable high-assurance systems and an overview of current techniques for information flow control in them. It addresses how mix-trusted components can be interconnected with confidence that they will adhere to the defined information flow policy. It specifically focuses on this bottom-up approach to security by providing a secure computing base to build up from. It also discusses several areas of future research in the security in reconfigurable systems. This includes providing better use of untrusted IP and how to provide the greatest reconfigurability while maintaining strict security.

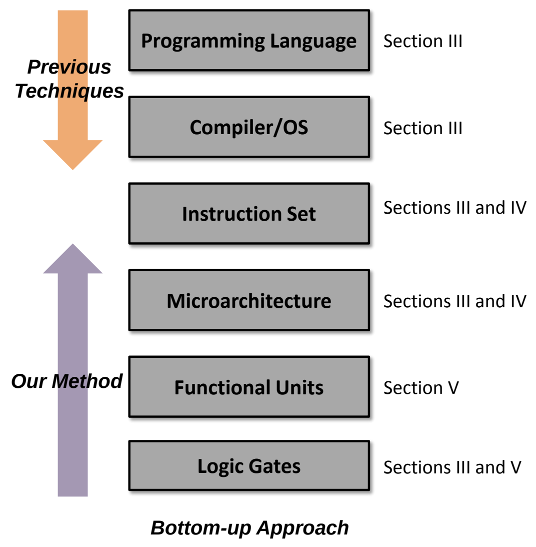


Fig. 1. Our bottom-up approach to security in high-assurance systems. Building a secure information flow secure hardware skeleton allows for strict control of all information flows all the way up the system hierarchy.

The remainder of this paper is organized as follows. In Section II we motivate the need for information flow security in mix-trusted reconfigurable systems. Section III presents the previous work in hardware information flow tracking techniques, specifically gate level information flow tracking (GLIFT). It also illustrates techniques applied to all layers in the system design stack as shown in Figure 1. Section IV discusses how a secure reconfigurable system can be designed based on findings in our previous work. It elaborates on a prototype system we developed using the mentioned bottom-up design methodology. Section V provides some fundamentals of gate level information flow tracking. This section focuses on how the Logic Gate level of the system stack can be developed with confidence. Section VI concludes the paper and presents some future research directions.

II. Reconfigurable Security and Motivation

Modern reconfigurable systems are typically designed with a number of mature building blocks known as *soft-cores* which generally come from vendors of varying trust. Some examples of these cores include AES encryption units, digital signal processors (DSP), memories, and Xilinx's MicroBlaze Processor [20]. The information flowing between these cores needs to occur only where the designer intends. For instance, trust needs to be upheld in a network interface core to ensure proper routing of information and data encryption cores must be proven to not leak the secret key through security holes including those from timing. From the designer's perspective, building a system with existing building blocks is tremendously easier than designing the system from scratch. As a result, these different cores should be placed into a system with confidence that they will not have any malicious effects on its operation.

Some potential threats that come up when using mix-trusted IP cores stem from the fact that their overall behavior is completely untrusted. Untrusted cores might be filled with malicious inclusions such as hardware trojans [4], [5] which can spawn unknown and harmful behavior that can cripple the integrity of other trusted portions of the system. Further, third-party cores can potentially learn secret information about classified components in the system if they all share a common bus. Providing strict information flow control in these reconfigurable systems is required if confidentiality and integrity of the systems is to be upheld.

Currently, it is difficult to ensure that an untrusted core is in fact behaving as expected. Some methods have focused on physically isolating cores and providing methods for routing over only statically defined channels. Our previous work [9] addresses this notion by using "moats" to isolate cores while allowing interconnect between cores only through pre-defined "drawbridges". In doing so, cores are expected not to leak secret keys (in the case of confidentiality) or be contaminated by untrusted data (in the case of integrity). However, even though cores are only understood to explicitly communicate through predefined channels, there is no guarantee about whether or not policy-violating information flows occur through implicit channels.

Previous work has shown that timing side channel attacks can be used to extract secret encryption keys from the latencies of caches [10] and branch predictors [13]. Cache timing attacks can obtain the secret key by observing the time for hit and miss penalties of the cache. Branch predictor timing channels are exploited in a similar manner where information is leaked through the latency of predicted and mis-predicted branches. Another exploit can be seen in a common bus where devices communicate implicitly through traffic (or lack of it) on the bus [14]. In order to have complete confidence that information is only flowing through the statically defined channels, strict information flow control needs to be guaranteed. The next section discusses some common techniques, specifically Gate Level Information Flow Tracking (GLIFT) which can be used to monitor the movement of all information in a system even those through timing channels.

III. Information Flow Tracking and GLIFT

A large amount of previous work has been done in the area of information flow security for complete systems. Numerous works have been done on information flow tracking specifically in hardware because monitoring information flows at this level allows for unintended flows to be identified without significantly affecting system performance. This section discusses the previous research in information flow security for complete sytems and specifically focuses on GLIFT since it is the primary technique we used in previous and continuing analyses.

Information flow security focuses on monitoring the movement of data among different trust levels. Traditionally information flow security is guaranteed by ensuring that a particular information flow policy, such as integrity or confidentiality, is upheld. These policies can be modeled using a lattice (L, $\sqsubseteq$) [19], where L is the set of security labels and $\sqsubseteq$ is a partial order between these security labels that specifies the permittable information flows. For example, consider the example lattices shown in Figure 2. Figure 2 (a) and (b) show two common binary lattices. For binary security lattices, the term *taint* is often used for the higher label on the lattice. For integrity, untrusted information is considered *tainted* in order to monitor if this taint violates a trusted (untainted) location. For confidentiality, taint is defined differently. The policy taints secret information to verify whether this leaks to an unclassified domain. Figure 2 (c) shows a confidentiality security lattice with multiple trust levels. Here confidentiality specifies that information may flow upward on this lattice but not downward. For simplicity, our analysis focuses on the binary lattices `trusted` $\sqsubset$ `untrusted` (a) for integrity and `unclassified` $\sqsubset$ `secret` (b) for confidentiality. For the two policies, this relation shows that information is allowed to flow from trusted to untrusted or from unclassified to secret respectively.

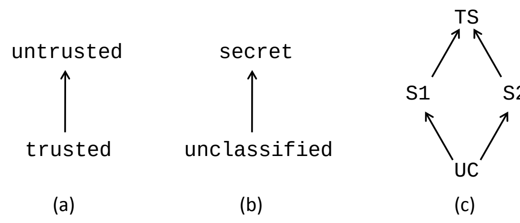


Fig. 2. Examples of different security lattices. (a) is a typical binary security lattice for integrity. (b) is a binary security lattice for confidentiality. (c) is a more complicated confidentiality lattice with multiple labels. Here TS is top-secret, S1 and S2 are two security levels which are less secure than TS but more private than UC (unclassified).

To be concrete, consider the code snippet shown in Figure 3. First, to understand explicit information flows and how they can be tracked, consider the assignment shown `y = z`. For integrity, using the lattice `trusted` $\sqsubset$ `untrusted`, this code is secure if $L(\mathtt{z}) \sqsubseteq L(\mathtt{y})$. In other words, this code is information flow secure only if the label assigned to `z` allows for information to flow into `y` without violating the integrity of `y`. Integrity in this situation is not violated as long as both `y` and `z` have the same label or `y` is untrusted and `z` is trusted. Confidentiality follows a similar procedure but using the lattice `unclassified` $\sqsubset$ `secret`. Using this lattice, the assignment is information flow secure if $L(\mathtt{z}) \sqsubseteq L(\mathtt{y})$. Meaning that information can only flow into `y` from `z` if `y` is at higher or equal security level to that of `z`. Implicit flows in this example follow a similar strategy except that they flow information indirectly to the variable. This particular code shows an implicit channel in the form of a branch. Here `x` leaks information to `y` because, depending on `x`, `y` will be assigned the value of `z`. For both confidentiality and integrity, implicit flows need to also be eliminated. In this particular example, to enforce integrity or confidentiality, the labels must adhere to $L(\mathtt{x}) \sqsubseteq L(\mathtt{y})$ in a similar manner as the explicit flow. Note that if integrity or confidentiality is to hold for the entire code, both the information flow constraints for the explicit *and* implicit flows must be enforced. Using this common model of information flow security, many implementations have been made to enforce this at all layers of the system design.

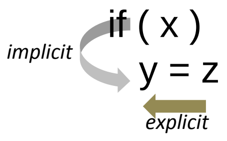


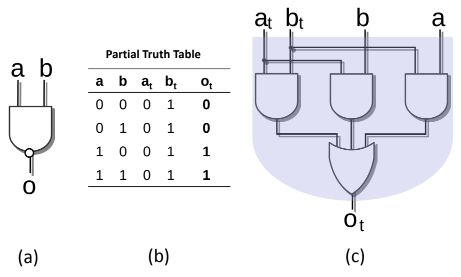
Fig. 3. Simple code snippet showing explicit information flow from `z` to `y` and implicit information flow from `x` to `y`.

The most common techniques for information flow security are implemented in programming languages using type based systems and in operating systems. Sabelfield and Myers [16] present a survey on the different programming language based techniques. Most work has been done in static compile based techniques which build off of the typing system of a language in order to enforce information flow security. These methods have shown to be effective and can even eliminate implicit channels due to conditional branches in execution. Jif [15] is a good example of such a type based system. Flume [17] has been shown to enforce information flow security using abstractions for operating system primitives such as processes, pipes, and the file system. Using theorem proving techniques, sel4 [18] has been shown to be fully verified for correctness. These schemes are often effective but are forced to abstract away the potential implicit flows that occur in hardware. Further, these schemes force the designer to comply with a new typing system (in programming language techniques) or reduce the overall system performance (in operating system abstractions).

To maintain system performance, information flow tracking has been proposed in hardware. The most common hardware information flow tracking strategies focus on the Instruction Set Architecture (ISA) and microarchitecture. One such technique called Dynamic information flow tracking (DIFT), proposed by Suh et al. [22], tags information from untrusted channels such as network interfaces and tracks it throughout a processor. They label certain inputs to the processor as "spurious" (tainted) and check whether or not this input causes a branch to potentially untrusted code. This technique has been shown to successfully prevent buffer-overflow [35] and format string attacks [36]. Raksha [23] is a DIFT style processor that allows the security policies to be reconfigured. Minos [24] uses information flow tracking to dynamically monitor the propagation of integrity bits to ensure that potentially harmful branches in execution are prevented in a manner similar to [22].

These previous techniques are effective at ensuring that potentially harmful branches in control flow are prevented or guaranteeing the integrity of critical memory regions. However, these methods target a higher level of abstraction (from the microarchitecture up as shown in Figure 1) and cannot be used to monitor the information flows in general digital hardware. For this reason, these methods also fail to detect hardware specific side channels in the form of timing. GLIFT provides a solution for tracking information flows, including those through timing channels, in general digital hardware. GLIFT works by tracking each individual bit in a system as they propagate through Boolean gates. This is done using an additional tag bit commonly referred to as *taint* and tracking logic which specifies how taint propagates. Information is said to flow through a logic gate if particular *tainted* inputs have a chance to *affect* the output.

Taint is a label associated with each data bit in the system which indicates whether or not this particular data bit should be tracked. If integrity is a concern, untrusted information is tainted to ensure that this tainted information does not flow to a trusted location. In the case of confidentiality, secret information is tainted to monitor whether it leaks to a public domain. Taint is propagated whenever a particular tainted data bit can affect the output. In other words, if the output of a function is dependent on changes to tainted inputs, then the output is marked as tainted.



Partial Truth Table

a	b	a_t	b_t	o_t
0	0	0	1	**0**
0	1	0	1	**0**
1	0	0	1	**1**
1	1	0	1	**1**

Fig. 4. (a) A two-input NAND gate. (b) Truth table of two-input NAND gate with taint information (not all the combinations are shown). (c) The corresponding tracking logic of two-input NAND gate is $ab_t + ba_t + a_tb_t$. Every change at the input of the gate is precisely tracked at the output.

For example, consider a simple 2-input NAND gate as seen in Figure 4 (a) and its corresponding tracking logic as shown in Figure 4 (c). For a NAND gate, only particular input changes will result in a change at the output. Specifically, consider the case in which $a = 0$ and $b = 1$. Here changing the value of b will cause no change at o since $a = 0$, meaning that there is no information flowing from b to o. If b were to be tainted ($b_t = 1$) and a untainted ($a_t = 0$) in this case, o would be untainted ($o_t = 0$) since the tainted input does not affect the output. A subset of all such combinations can be seen in Figure 4 (b). Using the full truth table, a function can be derived for all similar input combinations into a tracking logic function as shown in Figure 4 (c). Since NAND is functionally complete, the tracking logic for any digital circuit can be derived by constructively generating the tracking logic for each gate. In other words, given a circuit represented as NAND gates, the circuit can have complete information flow tracking by interconnecting the tracking logic for each individual NAND gate. This results in a design that precisely tracks the information flow of each individual bit. As mentioned, GLIFT is a useful tool for analyzing any digital hardware because it exposes all information flows explicitly. The next section will discuss how we used GLIFT and related techniques to develop a prototype system that was verified to be information flow secure.

IV. Information Flow Secure System Design

As reconfigurable systems become more complex, it is very common to have multiple mix-trusted IP cores existing in the same design. Previous work makes efforts to ensure physical isolation between different IP cores and have interconnect only through known channels [9]. However, since information can often flow through difficult to detect timing channels, deeper analysis is required in order to guarantee complete information flow isolation between mix-trusted IP cores in these systems. This section discusses how a secure reconfigurable system can be designed from the bottom-up (Figure 1) using GLIFT and related techniques.

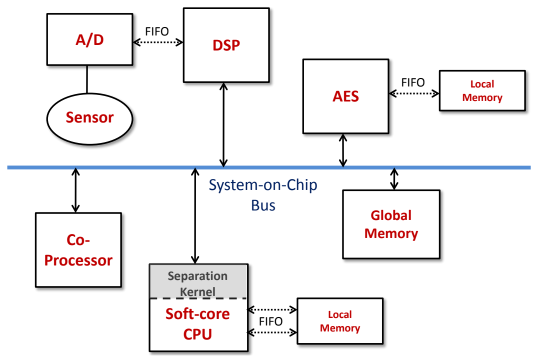


Fig. 5. A common reconfigurable system which consists of a soft-core processor, a global system-on-chip bus, and various peripherals. Interaction between each of these components must be regulated to ensure that information is flowing only to where the designer intends it.

Typical embedded reconfigurable systems consist of a soft-processor core running a microkernel or more robust operating system. Interacting with this processor is typically a wide range of peripherals from digital signal processors (DSP), AES encryption cores, memories, and even other processor cores. A common reconfigurable based system can be found in Figure 5. Here a soft-core processor exists with several different peripherals all interacting over a common bus. There are also components interacting with their own memories (AES core) or with external sensors (DSP core). Common bus protocols in such systems include ARM's AMBA, Inter-integrated circuit protocol (I^2C), and the universal serial bus (USB). The soft-core processor is typically the "heart" of the system and manages most of the interaction and runs application software. Typical embedded systems consist of a small microkernel which manages the execution of mix-trusted applications. As a result, it is essential to provide information flow guarantees in the processor core, its interaction with various peripherals over commodity buses, and in the microkernel itself since it arbitrates between mix-trusted executions. The following subsections introduce these different components and discuss our general testing flow.

A. GLIFT Test Method

The typical GLIFT based testing method can be seen in Figure 6. Here a design is modeled in a hardware description language (HDL) such as Verilog or VHDL. This testing flow is general enough to target any model in Verilog or VHDL, however finite state machine (FSM) representations are much easier to analyze for implicit timing channels since state transitions commonly occur every cycle.

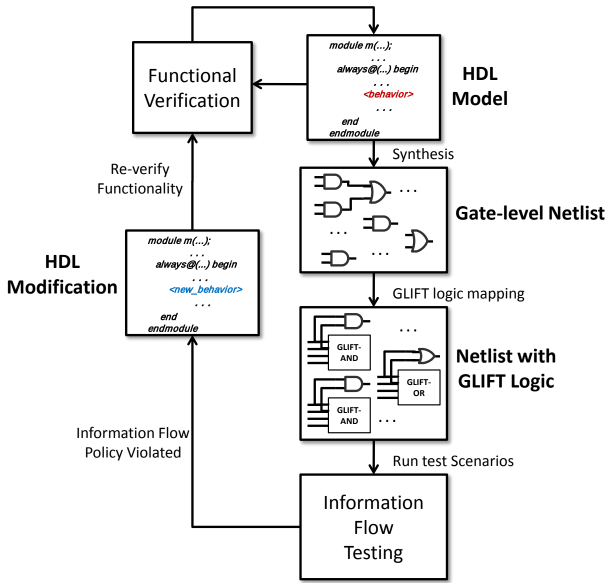


Fig. 6. Information flow test method for general digital hardware. Designs are represented as a finite state machine in Verilog or VHDL and processed through the testing flow.

As Figure 6 shows, the design enters the analysis flow as Verilog or VHDL. At this stage, the hardware model undergoes typical functional verification to ensure that the design conforms to the correctness of its specification. Note that at this stage, information flow guarantees are not yet being tested.

After synthesis, the design is equipped with GLIFT logic, which allows all of its information flows to be analyzed. There are many methods for generating this tracking logic for a given circuit, but this particular analysis uses a constructive approach. In other words, every gate in the system has logic attached to it individually in a linear fashion. This design equipped with GLIFT logic is now capable of being analyzed for unintended information flows. At this stage, test vectors are run on the hardware to see if it violates the information flow policy. If the policy is violated, the description of the hardware is modified and once again undergoes functional verification and proceeds through the testing flow again.

Making modifications to the hardware model is not obvious and generally requires a form of time-multiplexing to prove the absence of timing channels from the design. This has shown to be effective when analyzing processor cores and bus controllers as subsequent sections discuss in more detail.

B. Processor Core

The processor core is required to handle majority of the application execution in the system. Since the processor core acts as the central unit of the system, it requires very strong information flow guarantees. It needs to be able to support mix-trusted program execution and ensure that its state is recovered or purged such that it does not leak any information.

An execution lease architecture, as proposed by Tiwari et al. [30], provides an effective method for guaranteeing the integrity and confidentiality of the system following mix-trusted context switches. As shown in Figure 7, this architecture works by *leasing* out the hardware for an untrusted application to execute for a fixed amount of time and with restricted memory bounds. In other words, the processor restricts lower trust programs to a space-time sandbox. This allows the leasee to execute any code it desires during this fixed time slot. Once the time slot expires, the leaser restores the program counter back to a known value and the system state is restored.

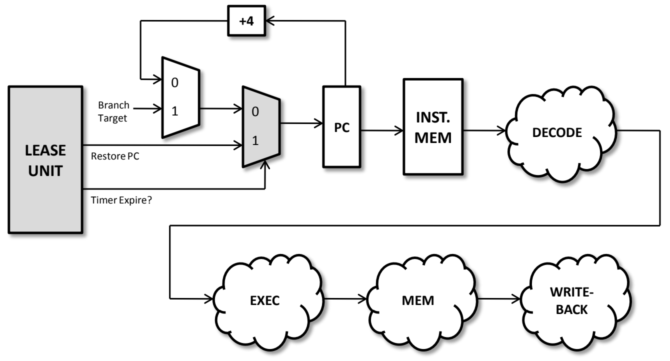


Fig. 7. The execution lease architecture fits directly into the typical 5-stage pipeline. Here the lease-unit restores the program counter upon the expiration of a lease. During a lease, the processor operates as normal. Restoring the system after a lease allows the architecture to execute untrusted code without leaking information.

Additionally, the execution lease architecture uses a stack of nested leases to allow for a larger granularity of trust. This stack of leases is managed in the execution lease unit. This unit keeps track of the current memory and time bounds. If the currently executing program wishes to execute untrusted code, it stores its current program counter in the lease unit, sets the memory and time bounds in the lease unit, then jumps to the untrusted code. Since there is a stack of nested leases, multiple programs can lease out the architecture to other less trusted ones. It should be noted that untrusted code cannot lease out to trusted code since the untrusted application can potentially observe the state once they lease expires, thus violating the information flow policies.

Specifically, this architecture implements these leases using two instructions known as `set_timer` and `set_membounds` which set the amount of time on the lease and the memory bounds respectively. The architecture first begins in a trusted state and this trusted state can be leased out to execution contexts which are at a lower trust level (in the case of integrity). Using the stack of leases, each subsequent lease can be further leased out to lower trust levels using the mentioned instructions.

The execution lease processor provides information flow guarantees by dynamically tracking information flows using GLIFT but lacks much of the performance optimizations of a modern processor including caches and pipelining. Our Star-CPU [31] builds off of the execution lease processor with a few additional features including caches and pipelining. Caches, pipelines, branch predictors and other stateful elements in a processor open the doors for potential timing attacks. Such attacks on caches have been shown to leak secret keys [10], [11] through the latency of caching operations. The attack works by an untrusted process first filling the contents of the cache. Subsequently a trusted process runs and upon a context switch back to the untrusted process, the untrusted process is able to observe which of its cache lines were evicted and extract the secret key as a result. The issue results from the mix-trust sharing of data in the cache and some current methods have made attempts at solving this problem [11] but some have found these new designs to still be partially vulnerable [12]. Our Star-CPU solves this issue by requiring that contents of the cache be strictly controlled by a trusted entity (separation kernel) and that strict partition in the cache is enforced. By having a trusted kernel controlling the contents of the cache, clearing and strict partitioning is enforced between context switches. This ensures that mix-trusted processes never learn information about one another through caching behavior.

Our Star-CPU provides further optimization over the execution lease CPU by providing pipelining. In a secure system, pipelines can leak information through timing channels due to non-deterministic behavior caused by branches and memory stalls. To ensure that such information does not leak between two processes of different trust, our CPU ensures that the pipeline is appropriately flushed between context switches. This ensures that such dynamic behavior does not cause information flows between programs of varying trust through implicit timing channels.

With any processor, it is desirable to communicate with various peripherals to gather data from sensors, read memories, or interact with DSPs. These peripherals all come from different trust levels and ensuring that data flows only to where the designer intends is required. The next subsection discusses these details further by analyzing the information flow characteristics of I^2C and USB.

C. Secure Input-Output for System Interaction

As mentioned, it is very common for reconfigurable systems to offer the flexibility for system designer to use many IP cores from many different vendors all with varying trust. Aside from the central processor core, many of these mix-trusted cores integrate directly into the system bus architecture and are frequently swapped out for either updated or completely different cores. For high-assurance systems, it is required that the communication between these various cores is information flow compliant. This section discusses our test methodology when analyzing information flow properties of bus protocols using GLIFT and briefly discusses our analysis of two common protocols: I^2C and USB.

1) Information Flows in I^2C: The inter-integrated circuit (I^2C) protocol was first developed by Philips. The protocol consists of a simple two wire bus with a serial data line (SDL) and serial clock line (SCL). In I^2C, devices all sit on a global bus with the master so explicit information clearly flows between all devices, since they can explicitly snoop the communication on the bus. However, as some of our previous work has shown [28], information can leak through difficult to detect timing channels.

To observe these timing channels, we processed our I^2C controller FSM through the testing flow shown in Figure 6. Our test consisted of a scenario in which master performs a write transaction with an *untrusted* slave and subsequently a write transaction with an *trusted* slave. This particular testing scenario is concerned with non-interference [6] (data integrity), but the analysis certainly works for confidentiality. As mentioned, I^2C operates on a global bus, so information clearly flows explicitly between devices on the bus since they are capable of openly snooping. However, even if devices are assumed to not snoop the bus, the state at which the master is left in leaks information from the untrusted slave to the trusted slave. This implicit information leak is in the form of a timing channel.

Following the communication with the untrusted slave, the master changes state and ultimately ends up in an untrusted state due to its interaction with an untrusted device. Subsequent communication with a trusted devices causes this untrusted information to flow into the trusted device. Since we are concerned about guaranteeing non-interference, we need to ensure that no information flows from the untrusted device to the trusted one through any channel. To eliminate all channels, we need to first bound the amount of time in which devices can communicate with the master. Secondly, we need to enforce that the master is returned back to a “known” state prior to communication with other devices. This solution can be seen in Figure 8 where an adapter is placed in between each device and the global bus. This adapter allows for a device to be connected to the bus only in a specific time slot. Once this time slot expires, the execution lease unit shown restores the master back to a known state to eliminate any possibility of an implicit flow. It should be noted that such an adapter can serve two purposes: 1) for information flow control and 2) for fault tolerance via a fault control unit (FCU) as discussed by [33]. Such dual usage justifies the hardware overhead and extends the confidence in the bus system to not only be fault tolerant but also information flow secure.

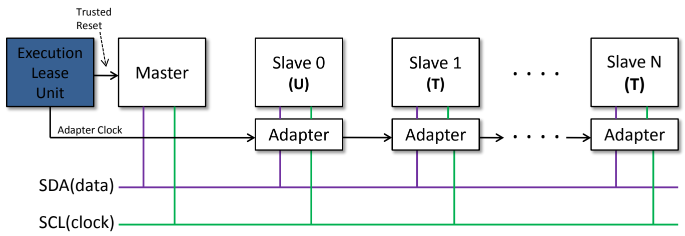


Fig. 8. I^{2}C bus system with an adapter which restricts information flows between devices. (U) represents an untrusted device and (T) a trusted one. After a time slot expires, the master is restored back to a known state prior to communicating with other devices.

2) Information Flows in USB: We chose to also analyze USB because of its different bus topology. USB operates as a star-tiered topology and does not operate on a global bus like I^{2}C. Since the bus is not global, there are no explicit information flows between devices because the architecture does not allow them to communicate with one another. However, as we will discuss, similar implicit information flows leak information between devices as seen in I^{2}C.

To identify the timing flows in USB, we modeled a USB system in Verilog and processed it using the flow shown in Figure 6. Our testing scenario consists of a USB host and two USB devices; one untrusted and the other trusted. The host performs a write transaction to an untrusted device and then subsequently a write to the trusted device. As in I^{2}C, this particular scenario focuses on non-interference to monitor the integrity of the trusted device. Since the devices on a USB bus have no way to interact with one another, there are no explicit flows. However, in a similar manner as I^{2}C, information implicitly flows between devices in the form of timing.

In this scenario, since the host first communicates with an untrusted device, the state in which it is left in unavoidably becomes untrusted. Subsequent communication with a trusted device causes this untrusted information to flow into the trusted device. Although this timing channel occurs in a nearly identical way to that of I^{2}C, to guarantee the absence of information flows requires a slightly different approach since USB is a much different bus architecture. In this case, the host operates using two different states, one which is untrusted and the other trusted. If the master is doing trusted communication it uses the trusted state and the untrusted state for untrusted communication. Swapping between the states is done in a similar manner as a context switch where the state is replaced yet none of the hardware is required to be replicated.

Having strict information flow isolation on a bus is only as good as the software which manages it. The next subsection discusses how a separation kernel can be used to allow for secure software executions and have secure management of both the processor and I/O.

D. Separation Kernel

Separation kernels are essential when managing many multi-process environments of varying trust. Separation kernels were first introduced by J. Rushby [29] as a way to show provable isolation between "partitions" and allowing communication through only predefined channels. Such a model is required when managing mix-trusted applications in a highly reconfigurable system. Reconfigurable systems will often continuously evolve their application source code, so it is essential that a managing mechanism can ensure information flow security between these applications.

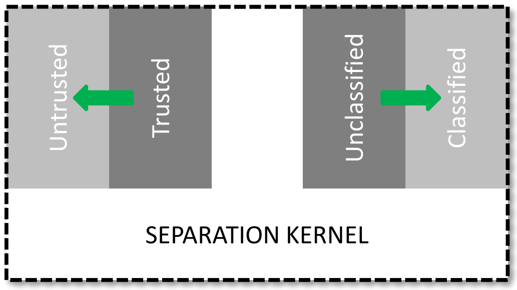


Fig. 9. A common separation kernel structure. A separation kernel manages the execution of different partitions of varying trust. Partitions are allowed to communicate over only predefined channels. For example, a trusted partition can write to an untrusted one and a classified partition can read from an unclassified one.

A high level overview of the organization of a separation kernel can be seen in Figure 9. As shown, there are different partitions which have varying trust. The separation kernel by definition should allow explicit communication only through defined communication channels. For example, communication between two processes should only exist through an IPC mechanism defined a priori. The separation kernel is required to manage all *leasing* parameters as enforced by the previously mentioned information flow secure CPU in Section IV-B.

Our implementation of such a microkernel in our information flow secure prototype system [31] requires that information flow security is preserved between context switches. Our separation kernel operates at the highest level of trust and is in charge of initially leasing out the architecture to less trusted execution contexts using the `set_timer` and `set_membounds` instructions. Since the kernel is at the highest level of trust, it is essential that while managing context switches it does not leak information between partitions. To enforce this, our separation kernel must not only have complete control over the microarchitectural state, but must also avoid pipeline stalls and cache misses since such non-determinism can leak information between partitions of varying trust upon a context switch.

Upon a context switch from an expired lease, the separation kernel follows a deterministic procedure. After a delay to flush the pipeline of the CPU, it commits the `set_partitionID` instruction for itself which allows it to access the memory of all partitions to store their state. This instruction activates the portions of the cache that are reserved for the currently executing partition. Once activated, the kernel obtains the current PC of the expired partition and stores it in the memory location reserved for this specific partition. Once the previous partitions state is saved, the kernel sets up the new context using the `set_partitionID`, `set_timer`, and `set_membounds` which set up the cache bound, lease duration, and new partition memory bounds respectively. Once the context for this new partition is set, the kernel loads the partitions PC and jumps to it.

Since the separation kernel operates in its own reserved cache partition, it never misses the cache. It also contains no branches that introduce non-deterministic pipeline delays. Such deterministic operation allows the kernel to perform a context switch between any two partitions without leaking any information about the preempted partition. Further, the behavior of the kernel must be strictly independent from any partition since it operates at the highest level of trust. To understand the importance of having deterministic context switches, suppose that the kernel did not operate in its own cache partition. If this was the case, the time in which a context switch occurred is directly dependent on what occurred in the old partition. Not only does information leak from the old partition to the new partition through this timing channel, but information also leaks from the less trusted partition into the kernel itself since it indirectly influenced its behavior. Since we are interested in adhering to information flow security through *all* channels, such timing channels need to be eliminated.

The kernel operates at the highest level of trust in the system and builds directly off of our secure hardware skeleton, thus its information flow security is also statically verified. All levels of our system are verified to be information flow secure using our $*$-logic technique, which is discussed in the next section.

E. Static Verification of Information Flow Security in Hardware

To statically verify that our complete system adheres to our information flow policy. We also developed a tool, known as $*$-logic (star-logic) [31], which is used to statically verify our policy.

This tool works by first building an abstract representation of the original design which gives the flexibility to specify any part of the system as unknown or $*$. This is required because in most reconfigurable systems the specific applications or operations of the system are not necessary known a priori. Once in this abstracted state, information flow tracking labels (such as trusted and untrusted) are assigned to the individual bits of the system. This abstracted system equipped with information flow tracking logic is simulated for all possible combinations. Since the only known part of the system at this stage is the separation kernel itself, all states can be exhaustively enumerated to ensure complete information flow security. As a result of this, this abstraction allows for the processor and system to be statically verified even if much of it is left unspecified at design time. Using our tool, we showed that this complete system is verifiably information flow secure given that the software base (separation kernel) was written by a trusted source.

Another static technique that works well for this verification is our tool Caisson [32]. Caisson is an HDL that uses static type checking to ensure information flow security. This static type checking is very similar to the previous IFT techniques in programming languages [16] as discussed in Section III. The key difference is that this language is intended for designing hardware, so the static type checking translates to secure hardware upon compilation. In other words, once hardware is designed in Caisson and passes the security type checker, our compiler generates information flow secure and synthesizable Verilog HDL. In our previous work [32], we have shown that a complete information flow secure processor can be designed using Caisson's static guarantees.

In both cases, this static verification requires some assumptions. First, the designer needs understand what resulted in an information flow or caused an error in the type checker and have the ability to make modifications to the system so that it is information flow compliant. When analyzing flows in hardware, using GLIFT or $*$-logic, designers should also be confident that the observed information flows are in fact not false positives. The next section discusses these issues to give the hardware designers better confidence in the information flow characteristics of the hardware.

F. Future Research in Information Flow Secure Systems

There is much room for improvement for the development and testing of information flow secure systems at every level of the design phase. In these systems, it is often desirable to re-use existing processor cores such as MicroBlaze [20] or Nios II [21] rather than designing one for security. In fact, most reconfigurable systems use these soft-core processors for applications which do not require a large software base and are mostly dataflow intensive. In this case, information flow control through the processor and between peripherals should be guaranteed. Future research directions should work towards monitoring the movement of information both intra- and inter-IP cores even if the RTL is not released. A question we are working to answer is: How do we leverage both software (for monitoring the movement of information through the embedded code) and hardware (for monitoring the movement of information between peripherals) techniques to build secure systems with exisisting procesors?

More research should also be taken in the security of the bus system. The bus system is the central location for all communication between the various cores involved in the system. Since many reconfigurable systems frequently interconect different components in the bus architecture generally from varying trust levels, having an information flow secure bus architecture is required if the overall system is concluded to be secure. Our previous work has shown a couple of promising methods for designing a secure bus architecture with timing enforcement in adapters for I^2C and root hubs in the host for USB. However, more research needs to be taken into using existing system-on-chip protocols with only minimal modification. Future research should be tagetted at the complicated and robust bus architectures used in modern systems, such as ARM's AMBA. Some common questions are: What sort of performance trade off do our methods have? How hard is it to prove the information flow properties for complex bus systems? What knowledge do we need to design provably information flow secure protocols? It should also be mentioned that our previous analyses of I^{2}C and USB involved specific testing scenarios. Meaning, we were only able to show information flow security for specific bus transactions. Future research directions are targeted at using our most recent technique, $*$-logic as discussed in Section IV-E, to statically verify the absence of information flow for any bus transaction.

The separation kernel also has future research for information flow security. In a reconfigurable environment it would be desirable to have the flexibility to modify or configure the properties associated with partitions or even the number of partitions in the kernel. Currently, making these changes requires that the system be completely re-verified for information flow security. Future research should be put into providing abstractions such that more reconfigurability is allowed in the system kernel to avoid re-verification. Is it possible to allow dynamic reconfigurability of portions of the kernel? How restrictive do we need to be without violating its integrity?

V. GLIFT Generation in Reconfigurable Systems

Up to this point, we have discussed how an information flow secure reconfigurable system can be designed with the help of GLIFT. However, much of the details of GLIFT and the details of its functionality have been left out. This section first discusses two GLIFT logic generation methods, namely the *brute force* and *constructive* methods. It also focuses on a preciseness problem that arises when using the constructive method.

A. The Brute Force Method

The brute force method for generating GLIFT logic follows straight from the definition of information flows and, as the name suggests, is a "brute force" approach with its complexity being exponential. This method works by changing an input to a function and observing whether or not this caused a changed at the output. If a change occurred, information is said to flow from the input to the output by definition of an information flow. For each such case we add a logic term to the GLIFT logic function. Once all input combinations are checked, we will have generated complete GLIFT logic which precisely tracks information flows through the original function.

The complexity of this algorithm is $O(2^{2n})$ where n is the number of inputs in the original function under test. The interested reader should refer to some of our previous work [34] to find more details of this proof and analysis. Although the brute force method does in fact generate a precise GLIFT function, (i.e. it indicates the presence of an information flow *iff* one actually occurred), its computational complexity makes it impractical for any reasonably large designs. Another method, known as the constructive method, operates in linear time but with a slight trade-off in precision.

B. The Constructive Method

The constructive method is a much less complex approach than the brute force method. It works by generating GLIFT logic for each primitive in the system compositionally. This is done by creating a library for each primitive in the design and mapping this primitive to its corresponding GLIFT logic in a similar manner as technology mapping. For example, GLIFT logic primitives for *AND*, *OR*, and *NOT* can be built into a library. As the circuit is being processed, each gate has its corresponding GLIFT logic replaced using the GLIFT primitives from the aforementioned library.

Figure 10 shows the constructive method when used on a 2-input Multiplexer. First, the logic equation represented as a network of *NOT*, *AND* and *OR* gates. Then, these logic primitives are replaced using the GLIFT primitives in the library as previously mentioned. Finally, proper connections are made to complete the shadow logic circuit.

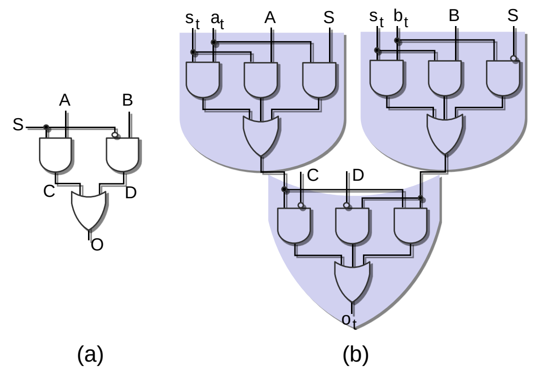


Fig. 10. (a) A 2-to-1 multiplexer. (b) GLIFT logic of 2-to-1 multiplexer generated using the constructive method.

However, generating GLIFT logic using the constructive method is not always guaranteed to be precise. Meaning, a GLIFT logic function generated with this method will often indicate that a flow of tainted information propagated from the input to the output when it in fact no such flow occurred. In other words, the GLIFT logic will have false positives indicating the false presence of information flows.

For example, Table I shows the number of "1's" (minterms) in the GLIFT logic truth tables for a 4-bit adder generated by the two discussed methods. We can see that the number of minterms for the brute force method is less than or equal to that of the constructive method. This means that the constructive method more frequently indicates that information flowed from the input to the output of the logic function. Since the brute force method generates precise GLIFT logic by its definition, the constructive method is actually overly conservative because it contains false positives.

Such additional minterms are false positives that indicate that a flow of information has occurred when in fact it has not. Taint can quickly propagate throughout the system, e.g.,

TABLE I
MINTERM COUNTS OF SHADOW LOGIC FUNCTIONS OF A 4-BIT ADDER GENERATED BY THE BRUTE FORCE AND CONSTRUCTIVE METHODS.

Method	sum[0]	sum[1]	sum[2]	sum[3]	cout
Brute Force	229376	241664	246272	248000	208160
Constructive	229376	245760	251648	250656	227864

a tainted state machine can taint the whole design in just a few clock cycles or a tainted PC (Program Counter) will quickly cause every bit of information in the processor to become tainted. When a conservative shadow logic function is used for taint propagation, the entire system can get into a tainted state when in fact it is not tainted. At this point, a declassification such as what is presented in [30], from a separation kernel with the highest security level is required to recover the system to a usable state. Generally speaking, being conservative is safe but frequent declassification will make a system unusable since an overbearing number of false-positives will continuously indicate that the system is untrusted (non-interference) or leaking confidential information (confidentiality). Section V-C discusses the imprecision problem and overviews solutions to it.

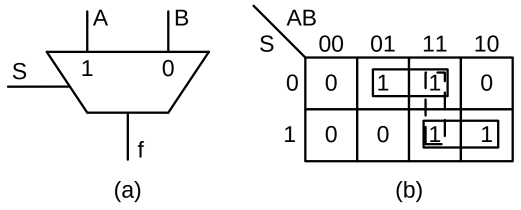
C. Imprecision Problem

As mentioned, the constructive method tends to report false positives in information flows for certain functions. The precision problem is important to understand especially when information flows are to be understood with high confidence. In a reconfigurable system, if the information flow tracking logic used frequently indicates that the system is in an untrusted state, then proving safe interaction between mix-trusted subsystems will essentially become impossible. Understanding how to reduce the amount of false positives in the analysis logic is essential to building an information flow secure system.

The constructive method in general produces more false positives than the brute force method making it overly conservative. A good example of these false positives can be seen in a 2-input multiplexer if GLIFT logic for this circuit is generated using the constructive method as shown in Figure 10.

The overly conservative result occurs when the select line of the multiplexer is tainted. In this case, the GLIFT logic indicates that the result is tainted regardless of what the inputs are. This is certainly the case if the inputs to the multiplexer are different since changes at the tainted select line will cause a change at the output of the function. However, if both inputs are the same, then the select line does not actually affect the output since the output will remain the same regardless of what the select line's value is.

To build a better intuition, consider the Karnugh Map in Figure 11 when S is tainted with A and B both untainted and logical 1. If the GLIFT logic is generated constructively for this circuit the two terms shown in black boxes will have GLIFT logic generated in addition to the OR gate that takes the disjunction of the two terms. Independently, the GLIFT logic for each of these terms is required to assume that the output of the function changed whenever the value of one of these terms changes in order to ensure "correctness". However, it can be seen that if the select line (S) changes and the values of A and B are both 1, the output value does not change (i.e. f remains 1), so there is in fact no tainted information flowing from the select line to the output.



S \ AB	00	01	11	10
0	0	1	1	0
1	0	0	1	1

Fig. 11. Karnaugh map of a 2-input multiplexer. The initial function $f = SA + \overline{S}B$, when shadowed constructively, is not precise. The dotted box indicates the additional term AB that must be added to the original logic function to insure its constructively derived shadow function is precise.

The root of this problem stems back to logic hazards in digital circuits and requires that the circuit have all *prime implicants* before having its GLIFT logic generated using the constructive method. The extra prime implicant in this example is indicated by the checkered box in the Karnaugh Map. The interested reader should refer to our previous work on this topic [34] for a formal proof of how this imprecision problem can be solved and how it requires a trade off between area and delay. The key detail from this precision problem is that some circuits (e.g. a 2-input multiplexer) require a careful information flow analysis in order to create a system which will be information flow secure with the best flexiblity. Since multiplexers are essential building blocks for any digital system, poor management of this precision problem easily results in an explosion of false positives which makes creates tremendous burdens during information flow verification when trying to understand the cause of the information flows.

D. Future Research in GLIFT Generation

Continuing research in GLIFT logic generation should focus on methods for better optimization for area, delay, and simulation time. It should also concentrate on methods for better understanding the precision problem. Optimizing for area, delay, and simulation time can be done using different encodings techniques for the GLIFT logic. This will produce state reductions resulting in smaller and more efficient tracking logic. How can we reduce the overhead of the tracking logic so that it is suitable to be deployed dynamically? Can we overload the logic such that it serves more than one purpose?

Another research area should focus on the precision problem in more detail. Being completely conservative is ineffective since the solution ends up being similar to physical isolation. Full precision is likely excessive since it results in tremendously large overheads in area and delay. Specifically, how much precision does a given application need to adequately track its information flows? Can this amount be determined and possibly quantified for a given system? Answers to these questions will allow more effecient use of GLIFT logic and will help designers better understand the security holes in their designs.

VI. Conclusions and Future Research

As the number of reconfigurable systems increases, the designs they implement are becoming even more complex. As these systems continue to benefit from COTS IP cores from vendors of varying trust bases, the need to ensure safe interaction between different components is critical. Information flow control is essential in these systems to guarantee the integrity of trusted components and confidentiality of secret data. GLIFT is a useful technique for monitoring information flows even those through implicit timing channels. This bottom-up approach to secure reconfigurable system design allows for strong information flow guarantees through all channels including timing. We have shown how this bottom-up approach can be used to build a verifiably information flow secure system with the help of gate level information flow tracking and related techniques.

Future research should focus on allowing more flexibility in the reconfigurability of the system. We have shown that current methods allow for a good starting point, where a secure hardware foundation can be multiplexed with mix-trusted cores and software and still be information flow secure. Strong information flow guarantees should be allowed for reconfigurable systems which use existing IP cores. New research needs to leverage both IFT in software and hardware to monitor the information inter- and intra-processor to allow the reuse of existing processor cores. In addition, our current methods have shown that mix-trusted cores can exist on the same bus without violating information flow security. However, more intricate bus protocols and architectures should be tested to show that this scales to modern systems. Lastly, kernels in such systems should have the flexibility to be reconfigured without complete system re-verification. Further abstractions are necessary to allow enough flexibility in the kernel design to accommodate more configurable and robust applications.

References

[1] *Common criteria for information technology security evaluation*. http://www.commoncriteriaportal.org/cc/

[2] *What does cc eal6+ mean?* http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean/

[3] *The integrity real-time operating system*. http://www.ghs.com/products/rtos/integrity.html

[4] Y. Jin and Y. Makris. *Hardware Trojan detection using path delay fingerprint*. IEEE International Workshop on Hardware-Oriented Security and Trust, Anaheim CA, 2008.

[5] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey. *Hardware Trojan horse detection using gate-level characterization*. Proceedings of the Design Automation Conference, 2009. vol., no., pp.688-693, 26-31 July 2009

[6] J. A. Goguen, J. Meseguer, *Security Policies and Security Models*. pp.11, IEEE Symposium on Security and Privacy, 1982

[7] D. Bell and L. LaPadula. *Secure computer systems: Mathematical foundations*. Technical report, Technical Report MTR-2547, 1973

[8] D. Federal Aviation Administration (FAA). *Boeing model 787-8 airplane; systems and data networks securityisolation or protection from unauthorized passenger domain systems access*. http://cryptome.info/faa010208.htm

[9] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine. *Moats and Drawbridges: An Isolation Primitive for Recongurable Hardware Based Systems*. In Proceedings of the Symposium on Research in Security and Privacy, Oakland, May 2007

[10] D. J. Bernstein. *Cache-timing attacks on AES*. Technical Report, 2005.

[11] Z. Wang and R. Lee. *New cache designs for thwarting cache-based side channel attacks*. In Proceedings of the 34th International Symposium on Computer Architecture, San Diego, CA, June 2007

[12] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. *Deconstructing new cache designs for thwarting software cache-based side channel attacks*. In Proceedings of the 2nd ACM workshop on Computer security architectures, CSAW 08, pages 2534, New York, NY, USA, 2008. ACM

[13] O. Accigmez, J. pierre Seifert, and C. K. Koc. *Predicting Secret Keys via Branch Prediction*. In Cryptology, The Cryptographers Track at RSA, pages 225-242. Springer-Verlag, 2007.

[14] W. M. Hu. *Reducing Timing Channels by Fuzzy Time*. In Proceedings of the Symposium on Research in Security and Privacy, Oakland, May 1991.

[15] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. *Jif: Java information flow*. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[16] A. Sabelfeld and A. C. Myers. *Language-based information-ow security*. IEEE Journal on Selected Areas in Communications, 21:2003, 2003

[17] M. Krohn and E. Tromer. *Noninterference for a practical difc-based operating system*. In Proceedings of the 2009 IEEE Symposium on Security and Privacy, 200

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verication of an os kernel. In SOSP 09: 22nd Symposium on Operating Systems Principles, pages 207220, NY, USA, 200

[19] D.E. Denning. *A lattice model of secure information flow*. Comm. ACM 19, 5 (May 1976), 236-243.

[20] *Xilinx Microblaze Soft-core Processor*. Available Online: http://www.xilinx.com/tools/microblaze.htm

[21] *Nios II Embedded Processor*. Available Online: http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[22] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. *Secure Program Execution via Dynamic Information Flow Tracking*. In ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, pages 85-96, New York, NY, USA, 2004. ACM Press.

[23] M. Dalton, H. Kannan, and C. Kozyrakis. *Raksha: A Flexible Information Flow Architecture for Software Security*. In 34th Intl. Symposium on Computer Architecture (ISCA), June 2007.

[24] J. R. Crandall and F. T. Chong. *Minos: Control Data Attack Prevention Orthogonal to Memory Model*. In Proceedings of the International

Symposium on Microarchitecture (MICRO), 2004

[25] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. *A Retrospective on the VAX VMM Security Kernel*. IEEE Transactions on Software Engineering, 17(11):11471165, 1991.

[26] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood, *Complete information flow tracking from the gates up*. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2009.

[27] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood and R. Kastner, *Theoretical Analysis of Gate Level Information Flow Tracking*, In proceedings of the 47th Design Automation Conference (DAC'10), June 2010.

[28] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, *Provable Information Flow Isolation in I^2C and USB*, In proceedings of the 48th Design Automation Conference (DAC'11), June 2011.

[29] J. Rushby, *Proof of Separability*, In Proceedings of the 5th International Symposium on Programming, Springer Verlag LNCS Vol. 137, pp. 352–367, Turin, Italy, 1982

[30] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood. *Execution Leases: A Hardware-Supported Mechanism for Enforcing Strong Non-Interference*, Proceedings of the International Symposium on Microarchitecture (Micro), December 2009. New York, NY

[31] M. Tiwari, J. Oberg, X. Li, J. K. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. *Crafting a Usable Microkernel, Processor and I/O System with Strict and Provable Information Flow Security*, Proceedings of the International Symposium of Computer Architecture. (ISCA) June 2011. San Jose, California

[32] X. Li, M. Tiwari, J. Oberg, F. T. Chong, T. Sherwood, and B. Hardekopf. *Caisson: A Hardware Description Language for Secure Information Flow*. In Proceedings of Programming Language Design and Implementation (PLDI 2011)

[33] J. Rushby, *Bus Architectures For Safety-Critical Embedded Systems*, Proceedings of the First Workshop on Embedded Software (EMSOFT), 2001, Lake Tahoe, CA.

[34] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. *Theoretical Fundamentals of Gate Level Information Flow Tracking*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).

[35] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, *StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks*. Proceedings of the 7th conference on USENIX Security Symposium, p.5-5, January 26-29, 1998, San Antonio, Texas

[36] T. Newsham. *Format String Attacks*. Guardent, Inc. September 2000. http://hackerproof.org/technotes/format/formatstring.pdf

Verifying the Authorship of Embedded IP Cores: Watermarking and Core Identification Techniques

Jürgen Teich and Daniel Ziener
Hardware/Software Co-Design, Department of Computer Science
University of Erlangen-Nuremberg, Germany
email: *{juergen.teich, daniel.ziener}@cs.fau.de*

Abstract—*In this paper, we present an overview of existing watermarking techniques for FPGA and ASIC designs as well as our new watermarking and identification techniques for FPGA IP cores. Unlike most existing watermarking techniques, the focus of our new techniques lies on ease of verification, even if the protected cores are embedded into a product. Moreover, we have concentrated on higher abstraction levels for embedding the watermark, particularly at the logic level, where IP cores are distributed as netlist cores. With the presented watermarking methods, it is possible to watermark IP cores at the logic level and identify them with a high likelihood and in a reproducible way in a purchased product from a company that is suspected to have committed IP fraud. The investigated techniques establish the authorship by verification of either an FPGA bitfile or the power consumption of a given FPGA.*

1. Introduction

The ongoing miniaturization of on-chip structures allows us to implement very complex designs which require very careful engineering and an enormous effort for debugging and verification. Indeed, complexity has risen to such enormous measures that it is no longer possible to keep up with productivity demands if all parts of a design must be developed from scratch. A popular solution to close this so called *productivity gap* is to reuse design components that are available in-house or that have been acquired from other companies. The constantly growing demand for ready to use design components, also known as IP cores, has created a very lucrative and flourishing market which is very likely to continue its current path not only into the near future. Today, there is a huge market and repertoire of IP cores which can be seen in special aggregation web sites, for example [1] and [2], which administrate IP core catalogs.

One problem of IP cores is the lack of protection mechanisms against *unlicensed usage*. A possible solution is to hide a unique *signature* (*watermark*) inside the core. However, there also exist techniques where an IP core can be identified without an additional signature. *Identification* methods are based on the extraction of unique characteristics of the IP core, e.g., lookup table contents for FPGA IP cores. With these techniques, the author of the core can be identified and an unlicensed usage can be proven. In this paper, watermarking as well as identification techniques for IP cores will be presented.

Our vision is that unlicensed IP cores, embedded in a complete SoC design which could be further embedded into a product, can be detected solely by using the given product and information from the IP core developer. Information of the accused SoC developer or product manufacturer should not be necessary and no extra information should be required from the accused company. Obviously, such concepts need advanced verification techniques to detect a signature or certain IP core characteristics, present in one of many IP cores inside a system. Furthermore, the embedded author identification should be preserved even when the IP cores pass through different design flow steps. It is of utmost importance that a watermark is transparent towards design and synthesis tools, that is, the embedded identification must be preserved in all possible scenarios. Whilst on the one hand, we must deal with the problem that automated design tools might remove an embedded signature all by themselves, a totally different aspect is that embedded signatures must also be protected against the removal by illegitimate parties whose intention is to keep the IP core from being identifiable. The latter is not to be taken lightly because if a sufficiently funded company decides to use unlicensed cores to, for example, lower design costs, there are usually very high skilled employees assigned with the task to remove or bypass the embedded watermark.

In Figure 1, a possible watermarking flow is depicted. An IP core developer embeds a signature inside his core using a *watermark embedder* and sells the protected IP core. A third-party company may obtain an unlicensed copy of the protected IP core and use it in one of their products. If the IP core developer becomes suspicious that his core might have been used in a certain product without proper licensing, he can simply acquire the product and check for the presence of his signature. If this attempt is successful and his signature presents a strong enough proof of authorship, the original core developer may decide to accuse the product manufacturer of IP fraud and press legal charges.

IP cores exist for all design flow levels, from plain text HDL cores on the register-transfer level (RTL) to bitfile cores for FPGAs or layout cores for ASIC designs on the device level. In the future, IP core companies will concentrate more and more on the versatile HDL and netlist cores due to

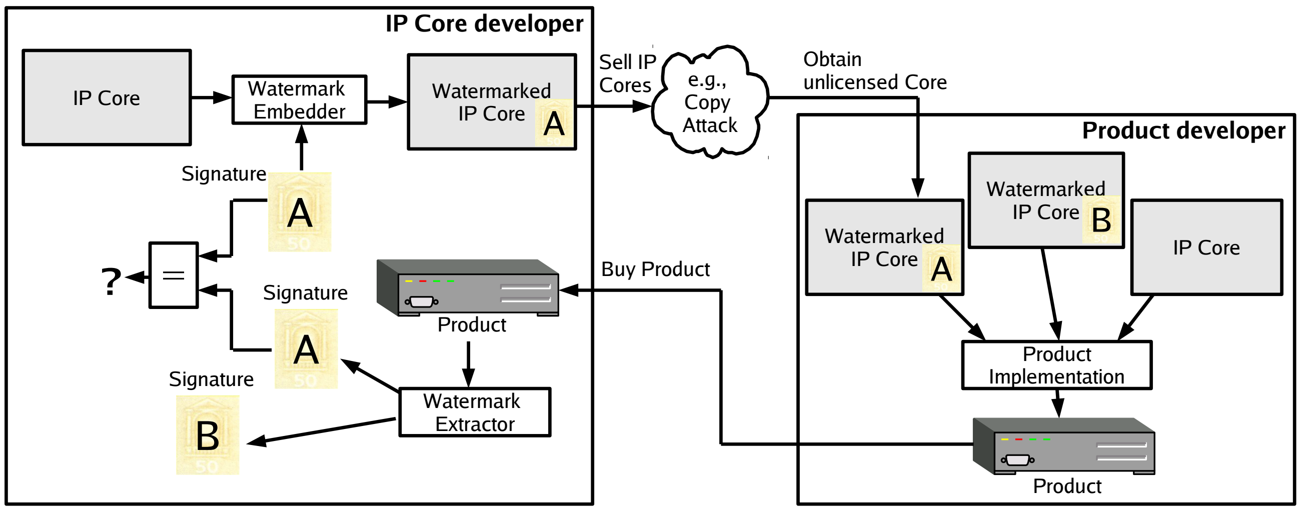


Fig. 1: This figure shows a typical watermarking flow: An IP core developer embeds a watermark A inside his core. If a product developer obtains an unlicensed core and embeds this core in his product, the IP core developer can buy this product and extract the watermarks of all used IP cores. Now, he is able to compare his signature with the extracted signatures.

their flexibility. One reason for this development is that these cores can be easily adapted to new technologies and different FPGA devices. This work focuses on watermarking methods for IP cores implemented for FPGAs. These have a huge market segment and the inhibition threshold for using unlicensed cores is lower than in the ASIC market where products are produced in high volumes and vast amounts of funds are spent for mask production. Moreover, we concentrate on flexible IP cores which are delivered on the logic level in a netlist format. The advantage of this form of distribution is that these cores can be used for different families FPGA devices and can be combined with other cores to obtain a complete SoC solution. Our methods differ from most other existing watermarking techniques, which do not cover the area of netlist cores, or are not able to easily extract an embedded watermark from a heterogeneous SoC implemented in a given product.

The remaining work is organized as follows: In Section 2, an overview of related work for IP watermarking is provided. Section 3 deals with different strategies to extract a watermark from an FPGA embedded into a product. We proceed by describing two ways for extracting a watermark. The first way explains the identification of an IP core from an FPGA bitfile in Section 4. Analyzing the power consumption of the FPGA in order to verify the presence of a watermark is the second method and will be discussed in Section 5. In conclusion, the contributions will be summarized.

2. Related Work

In general, hiding a signature into data, such as a multimedia file, some text, program code, or even an IP core by steganographic methods is called watermarking. For multimedia data, it is possible to exploit the imperfection of human eyes or ears to enforce variations on the data that represent a certain signature, but for which the difference between the original and the watermarked work cannot be recognized. Images, for example, can be watermarked by changing the least significant bit positions of the pixel tonal values to match the bit sequence of the original authors signature. For music, it is a common practice to watermark the data by altering certain frequencies, the ear cannot perceive and thus not interfering with the quality of the work [3].

In contrast, watermarking IP cores is entirely different from multimedia watermarking, because the user data, which represents the circuit, must not be altered since functional correctness must be preserved. A *fingerprint* denotes a watermark which is varied for individual copies of a core. This technique can be used to identify individual authorized users. In case of an unauthorized copy, the user, the copied source belongs to, can be detected and the copyright infringement may be reconstructed. Watermarking procedures can be categorized into two groups of methods: *additive methods* and *constraint-based methods*.

A survey and analysis of watermarking techniques in the context of IP cores is provided by Abdel-Hamid and others [4]. Further, we refer to our own survey of watermarking techniques for FPGA designs [5]. Moreover, a general survey of security topics for FPGAs is given by Drimer [6].

2.1 Additive Watermarking of IP Cores

Additive methods are watermarking procedures, where a signature is added to the core. This means that the watermark is not embedded into the function of the core. Nevertheless, the watermark can be masked, so it appears to be part of the functional part. Additive watermarks can be embedded into HDL, bitfile or layout cores.

2.1.1 HDL Cores

Additive watermarking for HDL cores seems to be very complicated, because of the human-readable structure of the HDL code. Hiding a watermark there is very difficult, because on the one hand, an attacker may easily detect the watermark, and on the other hand, subsequently used design tools might remove the watermark during circuit optimization. However, it is not impossible to include an additive HDL component into the core, which may not removed by the design tools.

Castillo et al. hide a signature into unused space of dedicated lookup table based memory [7]. To extract the signature, an additional logic monitors the input stream for a special *signature extraction sequence*. If this sequence is detected, the signature is sent to the outputs of the core. This approach was later generalized for other memory structures in [8].

Oliveira presents a general method for watermarking *finite state machines* (FSMs) in a way that on occurrence of a certain input sequence, a specific property exhibits [9]. The certain input sequence corresponds to the signature which is previously processed by cryptographic functions. A similar approach is presented by Torunoglu and others in [10] which explores unused transitions.

The disadvantage of these approaches is the usage of ports for signature verification. This works only if the ports are reachable. If the core is embedded into other cores, the ports of the watermarked core can be altered which falsifies or prevents the detection of the signature in the output stream. This applies also to the signature extraction sequence in the input stream.

2.1.2 Bitfile Cores

The approach of Lach and others watermarks bitfile cores by encoding the signature into *unused lookup tables* [11]. At first, the signature will be hashed and coded with an error correction code (ECC) to be able to reconstruct the signature even if some lookup tables are lost, e.g., during tampering. After the initial place and route pass, the number of unused lookup tables will be determined. The signature is split into the size of the lookup tables and additional LUTs are added to the design. Then, the place and route process will be started again with the watermarked design. Later, the approach was improved by using many small watermarks instead of a single large one [12]. The size of the watermarks should be limited by the size of a lookup table. The advantage is that small watermarks are easier to search for, and for verification, only a part of all of watermark positions must be published. With the knowledge of the published position, the watermark can be easily removed by an attacker. At the verification process, only a few positions of the watermark need to be used to establish the ownership. A second improvement is that a *fingerprinting technology* is added to the approach that enables the owner to see which customer has given the core away [13]. The fingerprinting technology is achieved by dividing the FPGA into tiles. In each tile, one lookup table is reserved for the watermark. The position of the mark in the tile encodes the fingerprint. For verification, it is possible to read out the content of the lookup table from a bitfile. So, these methods are easy to verify. It's more difficult to determine the position of the watermark in a tile, but it's still generally possible. However, if an attacker knows the position of the watermark, it is easy to overwrite it.

Saha and others present a watermarking strategy for FPGA bitfiles by subdividing the lookup table locations into sets of 2×2 tiles [14]. The number of used lookup tables in a set is used as signature. From an initial level, additional lookup tables are added to achieve the fill level according to the signature. The input and output are connected to the *don't care inputs* of the neighboring cells. Kahng and others show in [15] that the configuration of the multiplexer of unused CLB outputs in FPGA bitfiles can carry a signature. The signature is embedded after the bitfile creation and by knowing the encoding of the bitfile. These configuration bits can be later extracted to verify the signature.

Van Le and Desmedt show that these additional watermark schemes for bitfile cores can be easily attacked by reverse engineering, watermark localization, and subsequent watermark removal [16]. A simple algorithm is introduced which identifies lookup tables or multiplexers whose outputs are not connected to any output pins. However, these attacks are only successful if reverse engineering of the bitfile is possible and the costs of reverse engineering are not too high.

Finally, Kean and others present a watermarking strategy where a signature is embedded into an FPGA bitfile core or design [17]. The read out of the signature is done by measuring the temperature of the FPGA. This approach is commercially available as the product *DesignTag* from *Algotronix*.

2.2 Constraint-Based Watermarking of IP Cores

All optimization problems have constraints which must be satisfied to achieve a valid solution. Solutions which satisfy this constraints are the *solution space*. Constraint-based watermarking techniques represent a signature as a set of *additional constraints* which are applied to the hardware optimization and synthesis problem. These additional constraints reduce the solution space since the chosen solution must also satisfy the additional constraints [18], [19].

Qu proposes a methodology to make a part of the watermark – for constraint-based watermarking, some additional constraints – public which should deter attackers [20]. The other parts, called *private watermark*, are only known by the core author and are used to verify the authorship in case that the public watermark was attacked. A similar approach is used by Qu and others to generate different fingerprints by dividing the additional constraints into two parts [21]: The first part is a set of relaxed constraints which denote the watermark. By applying distinct constraints to the second part, different independent solutions can be generated which may be used as diverse fingerprinted designs.

Charbon proposed a technique to embed watermarks on different abstraction levels which he called *hierarchical watermarking* [22]. The idea is, if an attacker is able to remove a watermark, for example, embedded into the layout of a circuit, the watermarks added at higher abstraction levels are still present. However, Charbon focused more on *layout, nets*, and *latch watermarking techniques* which are only applicable for ASIC layout cores.

The verification of a constraint-based watermark is usually done with the watermarked core as it is. This means the watermarked core can be purchased or published and from the distributed cores the watermark can be verified. However, if the core is combined with other cores and traverses further design steps, the watermark information is usually lost or it cannot be extracted.

Van Le and Desmedt [16] present an ambiguous attack for constraint-based watermarking techniques. The authors add further constraints to the watermarked solution by allowing only a minimal increase of the overhead. The result is a slightly degenerated solution which satisfies many additional constraints. This means that in this solution, a lot of different signatures can be found which destroys the unique identification of the core developer. They choose, for example, the constraint-based watermarking approach for *graph coloring*. Further, this attack might be applicable to other constraint-based watermarking techniques.

As it was the case with additive watermarking strategies, constraint-based watermarking strategies are applicable for HDL, netlist, and bitfile cores.

2.2.1 HDL Cores

HDL code is usually produced by human developers or high-level synthesis tools. Both can set additional constraints to watermark a design. One approach is to use a watermarked *scan chain* [23]. Scan chains are usually used in ASIC designs to access the internal registers for debugging purposes. The use of scan chains in FPGA designs is rather unusual, but might be helpful in some cases. Depending on the signature, we have a variation on the scan chains which can be used to detect the watermark. This approach is easy to verify, if the scan chains can be accessed from outside of the chip. Problems occur, if the scan chain is only used internally or is not connected to any device. In such a case, there is no verification possibility.

Some work was done for watermarking *digital signal processing* (DSP) functions [24], [25]. This kind of watermarking has more in common with media watermarking instead if IP watermarking. Both approaches alter the function of the core slightly by embedding a watermark. In [24], the coefficients of *finite impulse response* (FIR) filters are slightly varied according to the watermark. Additionally, the authors use different structures to build the FIR filter which also corresponds to the signature. In [25], these ideas are extended and proven correct by mathematical analysis.

2.2.2 Netlist Cores

An approach to watermark netlist cores is to preserve certain nets during *synthesis* and *mapping* [19]. Synthesis tools merge signals or nets together and produce new nets. Only a few nets from the synthesis input will be visible in the synthesis result. The technology mapping tool also eliminates nets by assembling gates together in a lookup table. Kirovski's approach enumerates and sorts all nets in a design. The first nets of the input are chosen by the synthesis tools according to a signature. These nets will be prevented from elimination by the design tools by connecting these nets to a temporary output of the core. The new outputs from additional constraints for the synthesis tool, and the corresponding result is related to the watermark. A disadvantage is that it is easy to remove the additional logic. If the content of the lookup table is synthesized again, the watermark will be removed.

Meguerdichan and others presented a similar approach for netlist cores where additional constraints are added during the *technology mapping step* of the synthesis process [26]. In this approach, critical signals are not altered which preserves the timing and the performance of the core. The signature is encoded into the number of allowed inputs of a certain primitive cell, e.g., a gate or a lookup table. The primitive cells which are not in the critical path are enumerated, and according to the signature, the number of usable inputs are constrained.

Khan and others watermark netlist cores by doing a *rewiring* after synthesis [27]. Rewiring means that redundant connections between primitive cells are added in the netlist which makes other original connections redundant. These new redundant connections are removed.

Bai and others introduce a method for watermarking transistor netlists for *full custom designs* [28]. The transistors are enumerated and sorted into a list like in the approach above. Corresponding to the pseudo random stream generated from the signature, the width of the transistor gate is altered. If the transistor is assigned a '1' from the random stream, the transistor width is increased by a constant value.

2.2.3 Bitfile and Layout Cores

Additional placement, routing, or timing constraints can be added to watermark bitfile cores. To embed a watermark with placement constraints, Kahng and others place the *configurable logic blocks* (CLBs) in even or odd rows depending on the signature [29]. In this approach, the signature is transformed into even/odd row placement constraints. The placed core will be tested on preserving the constraints and, if necessary, CLBs are swapped. The problem of verification is to extract the CLB placement information. Only if knowing how the CLBs correspond to the signature, the watermark can be verified. A strategy to achieve this is to uniquely enumerate the CLBs in an FPGA from the top left corner.

Kahng and others [29] propose a second approach by adding constraints to the router. The constraints achieve that a net selected by the signature is routed with some additional, unusual routing resources. These unusual resources can be, for example, *wrong way* segments. A wrong way segment is a segment in which the net goes to the wrong direction and then back in the right direction to form a backstrap. The authors claim that this is unlikely for a normal router, and so such a net can be verified as a watermarked net.

Saha and others present a watermarking scheme by altering the size of the repeaters according to the signature [14]. In high performance ASIC designs, *repeaters* (a buffer for amplification of the signal) are inserted into critical nets to decrease the delay.

3. Watermark Verification Strategies for Embedded FPGAs

The problem of applying watermarking techniques to FPGA designs is not the coding and insertion of a watermark, rather it is the verification with an FPGA embedded in a system that poses the real challenge. Hence, our methods concentrate in particular on the verification of watermarks. When considering finished products, there are five potential sources of information that can be used for extracting a watermark: The configuration bitfile, the ports, the power consumption, electromagnetic (EM) radiation, and the temperature.

If the developer of an FPGA design has disabled the possibility to simply read back the bitfile from the chip, it can be extracted by wire tapping the communication between the PROM and the FPGA. Some FPGA manufactures provide an option to encrypt the bitstream which will be decrypted only during configuration inside the FPGA. Monitoring the communication between PROM and FPGA in this case is useless, because only the encrypted file will be transmitted. Configuration bitfiles mostly use a proprietary format which is not documented by the FPGA manufacturers. However, it seems to be possible to read out some parts of the bitfile, such as information stored in RAMs or lookup tables. In Section 4, we introduce netlist IP core identification and watermarking methods where the verification is done by using the extracted configuration bitstream.

Another popular approach for retrieving a signature from an FPGA is to employ unused ports. Although this method is applicable to top-level designs, it is impractical for IP cores, since these are mostly used as components that will be combined with other resources and embedded into a design so that the ports will not be directly accessible any more. Due to these restrictions, we do not discuss the extraction of watermarks over output ports.

Furthermore, it is possible to force patterns on the power consumption of an FPGA, which can be used as a covert channel to transmit data to the outside of the FPGA. We have shown in [30] and [31] that the clock frequency and toggling logic can be used to control such a power spectrum covert channel. The basic idea to use these techniques for watermarking is to force a signature dependent toggle pattern and extract the resulting change in power consumption as a signature from the FPGA's power spectrum. We refer to this method as “Power Watermarking” in Section 5

With almost the same strategy it is also possible to extract signatures from the electro magnetic (EM) radiation of an FPGA. A further advantage of this technique is that a raster scan of an FPGA surface with an EM sensor can also use the location information to extract and verify the watermark. Unfortunately, more and more FPGAs are delivered in a metal chip package which absorbs the EM radiation. Nevertheless, this is an interesting alternative technique for extracting watermarks and invites for future research.

Finally, a watermark might be read out by monitoring the temperature radiation. The concept is similar to the power and EM-field watermarking approaches, however, the transmission speed is drastically reduced. Interestingly, this is the only watermarking approach which is commercially available [17]. Here, reading the watermark from an FPGA may take up to 10 minutes. More about the different verification strategies can be found in [32].

4. Watermark Verification using the FPGA Bitfile

This section gives an overview of methods where the verification is done by *extracting an FPGA bitfile*. The bitfile can be analyzed to detect structures that can carry a watermark or that can be used to identify an IP core. Here, *lookup table contents* are used which are excellently suitable for watermarking and IP core identification. We start out by discussing how the contents of the lookup tables may be extracted from the FPGA bitfile. Following, methods for netlist and IP core identification are proposed (see also [33]). Following, watermarking methods for bitfile and netlist cores are discussed (see also [34]). The focus of these watermarking methods lies on the usage of *functional lookup tables* in order to increase the robustness against removal attacks. The term *functional lookup table* refers to lookup tables which are already used in a given (non-watermarked) IP core and represent a part of the functional logic of the core which may not be removed by an attacker in order to retain the correctness of the core.

4.1 Lookup Table Content Extraction

For FPGA designs, the functional lookup tables are an ideally suited component for carrying watermarks or using it for IP core identification. From a finished product, it is possible to obtain the configuration bitstream of the FPGA. The extraction of the lookup table contents from the configuration bitfile depends on the FPGA device and the FPGA vendor. To read out the LUT content directly from the bitfile, it must be known at which position in the bitfile the lookup table content is stored and how these values must be interpreted. In [33], for example, a standard black-box reverse engineering procedure is applied to interpret Xilinx Virtex-II and Virtex-II Pro bitfiles.

4.2 Identification of Netlist Cores by Analysis of LUT Contents

In this approach, we do not add any signature or watermark. The core itself remains unchanged, so the functional correctness is given and no additional resources are used. We compare the content of the used lookup tables from the registered core with the used lookup tables in an FPGA design from the product of the accused company. If a high percentage of identical content is detected, the probability that the registered core is used is very high.

The synthesis tool maps the combinatorial logic of an FPGA core to lookup tables and writes these values into a netlist. After the synthesis step, the content of the lookup tables of a core is known, so we can protect netlist cores which are delivered at the logic level. The protection of bitfile cores at the device level is also possible.

After the core is purchased, the customer can combine this core with other cores. In the following CLB mapping step, it is possible that lookup tables are merged across the core boundaries or are removed by an optimizing transformation. This happens when different cores share logic or when outputs of the core are not used. These lookup tables cannot be found in the FPGA bitfile, but experimental results in [33] show that the percentage of these lookup tables compared to the number of all lookup tables in the core is typically low for the used mapping tool (Xilinx *map*).

After the extraction of the content of lookup tables from a bitfile, we can compare the obtained values with the information in the netlist. Unfortunately, the mapping tools do not necessarily adopt these values. The mapping tool may merge lookup tables from different cores together, convert one, two or three input lookup tables to four input lookup tables and permute the inputs to achieve a better routing.

All lookup tables of an FPGA have n inputs. On most FPGA architectures, lookup tables have $n = 4$ or $n = 6$ inputs. In a core netlist, also lookup tables with less than n inputs may exist. These lookup tables must be mapped onto n input lookup tables. If one input is unused, only half of the memory is needed to store the function and the remaining space must be filled. In the case that a function uses less inputs than the underlying technology of the FPGA provides, it is desirable to turn the unused inputs into don't cares. Intuitively, this can be achieved rather easily by replicating the function table as it is demonstrated in Figure 2.

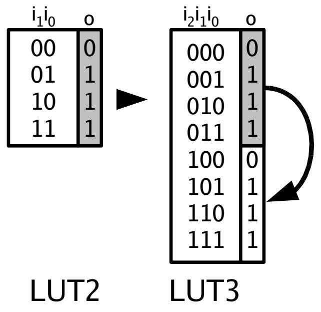


Fig. 2: Converting a two input lookup table into a three input lookup table with unused input i_2.

The mapping tool can permute the inputs of the lookup tables, for example, to achieve a better routing. In most FPGA architectures, the routing resources for lookup table inputs are not equal, and so a permutation of the lookup table inputs can lower the amount of used routing resources. Permutation of the inputs significantly alters the content of a lookup table. For n inputs, $n!$ permutations exist and thus up to $n!$ different lookup table values for one so-called *unique function*. To compare the contents of the lookup table from the netlist and the bitfile, it must be checked if one of these possible different lookup table values for one unique function is equal to the value of the lookup table in the bitfile. This is done by creating a table with all possible values of lookup tables for all unique functions (see Figure 3).

More about this method as well as experimental results

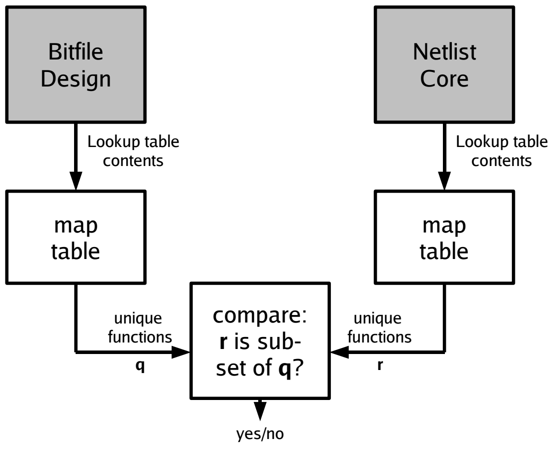


Fig. 3: Before the lookup table contents of the bitfile and the netlist are compared, they are mapped into unique functions.

and a robustness analysis can be found in [33] and [32]. The experimental results show that it is possible to identify a core in a design with a high probability.

4.3 Identification of HDL Cores by Analysis of LUT Contents

In the last section we have shown that is possible to identify an IP core, distributed as a netlist, in an FPGA design by analyzing the LUT contents of the configuration bitfile. However, many IP cores are published at the RTL abstraction level as HDL core.

To identify HDL cores, the lookup table contents can be used as well. However, the lookup table content is generated by the synthesis step, which is executed after the publication of the HDL cores. Therefore, a pirate who can obtain an unlicensed HDL core, controls the complete design flow from the RTL to the device level. It is up to the pirate to decide which synthesis tool is used to synthesize the core and therefore, create the lookup table contents. Different synthesis tools might create different lookup table contents. To prove or disprove this assumption, we analyzed common synthesis tools with respect to the generation of lookup table contents.

The first step is to analyze different netlist cores to find out whether they were generated from the same HDL core. The goal is to find different netlist cores which can be assigned to a corresponding HDL source even if they were synthesized with different tools and different synthesis parameters.

The comparision is based on the lookup table contents. Therefore, to compare two netlist cores, the first step is to extract the lookup table contents from the netlist cores and map these to unique functions. The probability that both cores were generated from the same source is high if a high percentage of lookup tables which can be found in both cores implement in both cores the same unique functions. A detailed description of this method as well as experimental results are presented in [32].

This method is the first steps towards an identification of HDL cores in bitfiles. Here, we concentrated on the synthesis step between the RTL and logic abstraction level. However, to build the complete chain for identification of HDL cores from bitfiles some links are missing. Identifying HDL cores in netlists has an inherent uncertainness comparable to the identification of netlist cores in bitfiles. By combining both techniques the uncertainness can be too high to give a trustful result. Nevertheless, this is an interesting topic for future research.

4.4 Watermarks in LUTs for Bitfile Cores

In this section, we introduce our first watermarking technique for IP cores. The easiest way to watermark an FPGA design is to place the watermarks into the bitfiles. Bitfiles are very inflexible because they were specifically generated for a certain FPGA device type, however, it makes sense to sell bitfile IP cores for common development platforms which carry the same FPGA type. Usually, a bitfile core is a whole design which is completely placed and routed and therefore ready to use. There also exist partial bitfiles which carry only one core. These partial bitfile cores can be combined into one FPGA which increases the flexibility of these cores and therefore may increase the trade possibilities.

In this approach, we hide our signature inside unused lookup tables. It is very unlikely that a design or bitfile core uses all available lookup tables in an FPGA. Before a design reaches this limit, the routing resources are exhausted and the timing degenerates rapidly. Therefore, many unused lookup tables exist in usual designs. On the other hand, lookup table content extraction is not difficult. Using lookup tables for hiding a watermark which are far away from the used ones, makes it easier for an attacker to identify and remove them. Even if an attacker is able to extract all lookup tables from a bitfile core, the lookup tables which carry the watermark should not be suspicious.

In Xilinx devices, lookup tables are grouped together with *flip-flops* into slices. A slice usually consists more than one lookup table, e.g., the Virtex-II and Virtex-II Pro devices have two lookup tables in one slice. It is not unusual that only one lookup table of a slice is used and the other remains unused. Hiding a watermark in the unused lookup table of a used slice is less obvious than using lookup tables in unused slices. Even if the attacker is able to extract the lookup table content and coordinates, the watermarks are hard to detect.

The extraction and verification of the watermark is rather easy. First of all, the content and the coordinates of all used lookup table of the core are extracted. For the verification there exist two approaches: a *blind approach* and a *non-blind approach*. In the blind approach, the watermarks are searched in all extracted lookup table contents, whereas in the non-blind approach the location of the watermarks are known. Having the right coordinates, the watermarked lookup table content can be directly compared to the watermarks of the core developer. The locations of the watermarks delivered from the core developer, however, should be kept secret, because otherwise it is very easy for an attacker to remove the marks.

More about this method as well as experimental results and a robustness analysis can be found in [32].

4.5 Watermarks in Functional LUTs for Netlist Cores

Since we want to keep the IP core as versatile as possible, we watermark the design in the form of a netlist representation, which, although technology dependent to a certain degree, can still be used for a large number of different devices. Netlist designs will almost certainly undergo the typical design flow for silicon implementations. This also includes very sophisticated optimization algorithms, which will eliminate any redundancy that can be found in the design in order to make improvements. As a consequence it is necessary to embed the watermarks in the netlist in such a way, that the optimization tools will not remove the watermarks from the design.

In Xilinx FPGAs, for example, lookup tables are essentially RAM cells, with the inputs specifying which of the stored bits to deliver to the output of the RAM. Naturally, these cells can therefore also be used as storage, but also as shift-register cells (see Figure 4). Interesting, however, is the fact that if the cell is configured as a lookup table, Xilinx optimization tools will try to optimize the contained logic function. If the cell is in contrast configured as a shift-register or distributed RAM, the optimization tools will leave the contents alone, but the logic function is still carried out. This means, that if we want to add redundancy to a netlist, that is not removed by automized tools, all we have to do is to take the corresponding cells out of the scope of the tools.

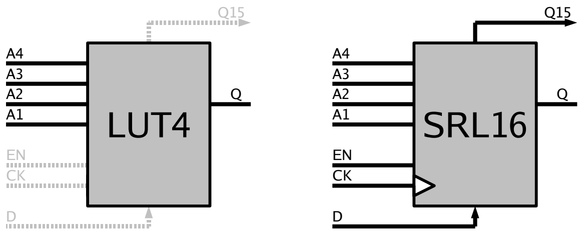


Fig. 4: In the Xilinx Virtex architecture, the same standard cell is used as a lookup table (LUT4) and also as a 16-bit shift-register lookup table (SRL16).

FPGAs usually consist of the same type of lookup tables with respect to the number of inputs. For example, the Xilinx Virtex-II uses lookup tables with four inputs whereas the Virtex-5 has lookup tables with six inputs. However, in common netlist cores many logical lookup tables exist, which have less inputs than the type used on the FPGA.

These lookup tables are mapped to the physical lookup tables of the FPGA during synthesis. If the logical lookup table of the netlist core has fewer inputs than the physical representation, the memory space which was not present in the logical representation remains unused. Using the unused memory space of functional lookup tables for watermarking without converting the lookup table either to a shift register or distributed memory turns out to be not applicable, because design flow tools identify the watermark as redundant and remove the content due to optimization. Converting the watermarked functional lookup table into a shift register or a memory cell prevents the watermark from deletion due to optimization.

If a product developer is accused of using an unlicensed core, the product can be purchased and the bitfile can be read out, e.g., by wire tapping. The lookup table content and the content of the shift registers can be extracted from the bitfile. Now, the extracted lookup table or shift register content can be used for a watermark detector which can decide if the watermark is embedded in the work or not.

A detailed description of this method as well as the experimental verification results and the overhead analysis are described in [34] and [32].

5. Power Watermarking

This section describes watermarking techniques introduced in [30] and [31], where a signature is verified over the *power consumption pattern* of an FPGA. For power watermarking methods, the term *signature* refers to the part of the watermark which can be extracted and is needed for the detection and verification of the watermark. The signature is usually a bit sequence which is derived from the unique key for author and core identification.

There is no way to measure the relative power consumption of an FPGA directly. Only by measuring the relative supply voltage or current the actual power consumtion can be inferred. We have decided to measure the voltage of the core as close as possible to the voltage supply pins such that the smoothing from the plane and block capacities are minimal and no shunt is required. Most FPGAs have *ball grid array* (BGA) packages and the majority of them have vias to the back of the PCB for the supply voltage pins. So, the voltage can be measured on the rear side of the PCB using an oscilloscope. The voltage can be sampled using a standard oscilloscope, and analyzed and decoded using a program developed to run on a PC. The decoded signature can be compared with the original signature and thus, the watermark can be verified. This method has the advantage of being non-destructive and requires no further information or aids than the given product (see Figure 5).

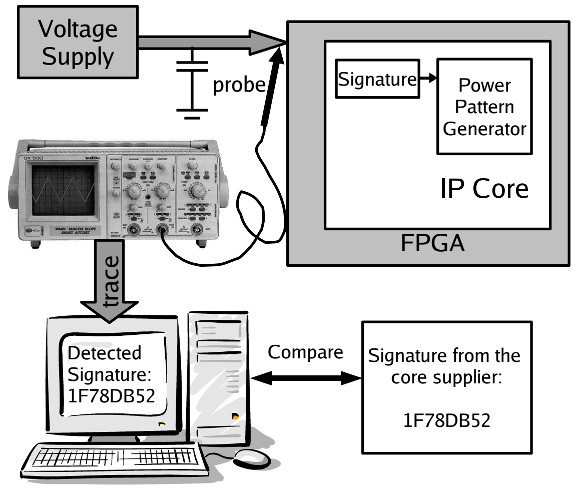


Fig. 5: Watermark verification using power signature analysis: From a signature (watermark), a power pattern inside the core will be generated that can be probed at the voltage supply pins of the FPGA. From the trace, a detection algorithm verifies the existence of the watermark.

In the power watermarking approach described in [35] an [30], the amplitude of the interferences in the core voltage i altered. The basic idea is to add a power pattern generato (e.g., a set of shift registers) and clock it either with th operational clock or an integer division thereof. This powe pattern generator is controlled according to the encoding o the signature sequence which should be sent.

The mapping of a signature sequence $s = \{0,1\}^n$ ont a sequence of symbols $\{\sigma_0, \sigma_1\}^n$ [31] is called encoding $\{0,1\}^n \rightarrow \mathcal{Z}^n, n \geq 0$ with the alphabet $\mathcal{Z} = \{\sigma_0, \sigma_1\}$ Here, each signature bit $\{0,1\}$ is assigned to a symbol. Eac symbol σ_i is a triple $(e_i, \delta_i, \omega_i)$, with the *event* $e_i \in \{\gamma, \bar{\gamma}\}$ the *period length* $\delta_i > 0$, and the *number of repetition* $\omega_i > 0$. The event γ is *power consumption through a shi operation* and the inverse event $\bar{\gamma}$ is *no power consumption* The period length is given in terms of number of cloc cycles. For example, the encoding through 32 shifts wit the period length 1 (one shift operation per cycle) if th data bit '1' should be sent, and 32 cycles without a shift operation for the data bit '0' is defined by the alphabet $\mathcal{Z} = \{(\gamma, 1, 32), (\bar{\gamma}, 1, 32)\}$.

Different power watermarking encoding schemes were introduced and analyzed. The basic method with encoding scheme: $\mathcal{Z} = \{(\gamma, 1, 1), (\bar{\gamma}, 1, 1)\}$, the enhanced robustness encoding: $\mathcal{Z} = \{(\gamma, 1, 32), (\bar{\gamma}, 1, 32)\}$, and the BPSK approach: $\mathcal{Z} = \{(\gamma, 1, \omega), (\bar{\gamma}, 1, \omega)\}$ are explained in detail in [30]. The correlation method with encoding $\mathcal{Z} = \{(\gamma, 25, 1), (\bar{\gamma}, 25, 1)\}$ can be reviewed in [31]. To avoid interference from the operational logic in the measured voltage, the signature is only generated during the reset phase of the core.

The power pattern generator consists of several shift registers, causing a recognizable signature- and encoding-dependent power consumption pattern. For example, the typical swing of an FPGA core voltage signal which results from a shift of a huge shift register is shown in Figure 6.

As mentioned before in Section 4.5, a shift register can also be used as a lookup table and vice versa in many FPGA architectures (see Figure 4 in Section 4.5). A conversion of functional lookup tables into shift registers does not affect the functionality if the new inputs are set correctly. This allows us to use functional logic for implementing the power pattern generator. The core operates in two modes, the *functional mode* and the *reset mode*. In the functional mode, the shift is disabled and the shift register operates as a normal lookup table. In the reset mode, the content is shifted according to the signature bits and consumes power which can be measured outside of the FPGA. To prevent the loss of the content of the lookup table, the output of the shift register is fed back to the input, such that the content is shifted circularly. When the core changes to the functional mode, the content have to be shifted to the proper position to get a functional lookup table for the core.

To increase the robustness against removal and ambiguity attacks, the content of the power consumption shift register

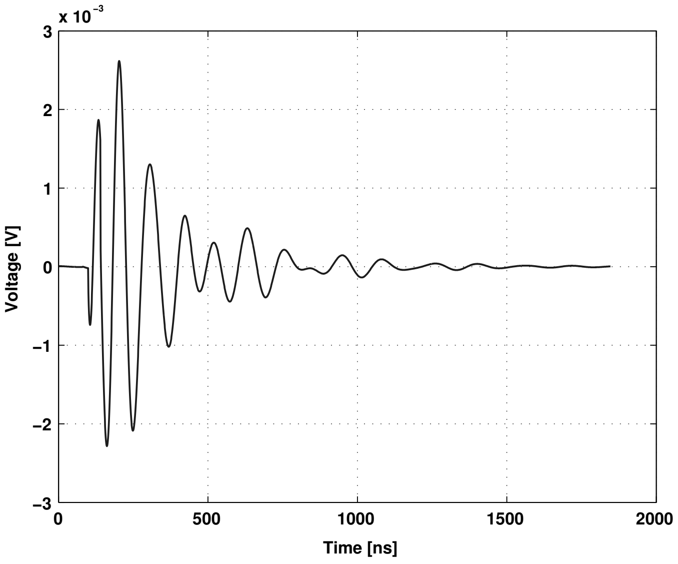


Fig. 6: This measurement is obtained by a shift of a huge shift register, implemented using 128 SRL16 primitive cells in the Spartan-3 FPGA on the Digilent Spartan-3 starter board [36]. Note that the DC component of the core voltage signal is removed by an AC filter.

which is also part of the functional logic can be initialized shifted. Only during the reset state, when the signature is transmitted, the content of the functional lookup table can be positioned correctly. So, normal core operation cannot start before the signature was transmitted completely. The advantage is that the core is only able to work after sending the signature. Furthermore, to avoid a too short reset time in which the watermark cannot be detected exactly, the right functionality will only be established if the reset state is longer than a predefined time. This prevents the user from leaving out or shorten the reset state with the result that the signature cannot be detected properly.

The signature itself can be implemented as a part of the functional logic in the same way. Some lookup tables are connected together and the content, the function of the LUTs, represents the signature. Furthermore, techniques described in Section 4.5 can be used to combine an additional watermark and the functional part in a single lookup table if not all lookup table inputs are used for the function. For example, LUT2 primitives in Xilinx Virtex-II devices can be used to carry an additional 12-bit watermark by restricting the reachability of the functional lookup table through clamping certain signals to constant values. Therefore, the final sending sequence consists of the functional part and the additional watermark. This principle makes it almost impossible for an attacker to change the content of the signature shift register. Altering the signature would also affect the functional core and thus result in a corrupt core.

The advantages of using the functional logic of the core as a shift register are the reduced resource overhead for watermarking and the robustness of this method. It is hard, if not impossible, to remove shift registers without destroying the functional core, because they are embedded in the functional design.

The watermark embedder consists of two steps. First, the core must be embedded in a wrapper which contains the control logic for emitting the signature. This step is done at the register-transfer level before synthesis. The second step is at the logic level after the synthesis. A program converts suitable lookup tables (for example LUT4 for Virtex-II FPGAs) into shift registers for the generation of the power pattern and attaches the corresponding control signal from the control logic in the wrapper (see Figure 7).

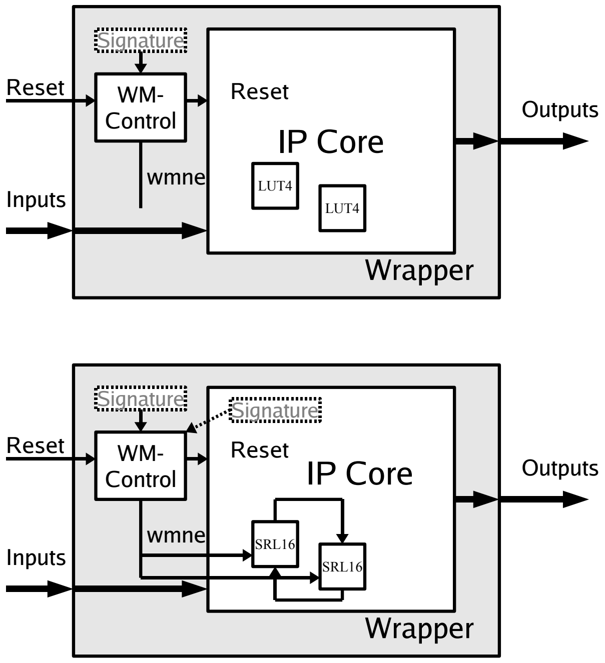


Fig. 7: The core and the wrapper before (above) and after (below) the netlist alternation step. The signal “wmne" is an enable signal for shifting the power pattern generator shift register.

The wrapper contains the control logic for emitting the watermark and a register that contains the signature. The ports of the wrapper are identical to the core, so we can easily integrate this wrapper into the hierarchy. The control logic enables the signature register while the core is in reset state. Also, the power pattern shift registers are shifted in correspondence to the current signature bit. If the reset input of the wrapper is deasserted, the core function cannot start immediately, but only as soon as the content in the shift registers has been shifted back to the correct position. Then the control logic deasserts the internal reset signal to enter normal function mode. The translation of four input lookup tables (LUT4) of the functional logic into 16 Bit shift registers (SRL16) is done at the netlist level.

The embedding procedure for Virtex-II netlist cores is done by a program which parses an EDIF netlist and writes back the modified EDIF netlist. First, the program reads all LUT4 instances. Then, the instances are converted to a shift register (SRL16), if required, initialized with the shifted value and connected to the clock and the watermark enable (wmne) signal according to Figure 7. Always two shift registers are connected together to rotate their contents. Finally, the modified netlist is created. The watermarked core is now ready for purchase or publication.

A company may obtain an unlicensed version of the core and embeds this core in a product. If the core developer has a suspicious fact, he can buy the product and verify that his signature is inside the core using a detection function. The detecting function depends on the encoding scheme. In [30] and [31], the detecting functions of all introduced encoding schemes are described in detail.

The advantage of power watermarking is that the signature can easily be read out from a given device. Only the core voltage of the FPGA must be measured and recorded. No bitfile is required which needs to be reverse-engineered. Also, these methods work for encrypted bitfiles where methods extracting the signature from the bitfile fail. Moreover, we are able to sign netlist cores, because our watermarking algorithm does not need any placement information. However, many watermarked netlist cores can be integrated into one design. The results are superpositions and interferences which complicate or even prohibit the correct decoding of the signatures. To achieve the correct decoding of all signatures, we proposed *multiplexing* methods in [37]. In this paper we show that the most promising techniques for use on an FPGA are time (TDM) and code (CDM) multiplexing.

6. Summary

In this paper, we have presented an overview of existing and new approaches for identication and watermarking of IP cores. Our methods follow the strategy of an easy verification of the watermark or the identification of the core in a bought product from an accused company without any further information. Netlist cores, which have a high trade potential for embedded systems developers, are in the focus of our analysis. To establish the authorship in a bought product by watermarking or core identification, we have discovered different new techniques, how information can be transmitted from the embedded core to the outer world. In this paper, we concentrated on methods using the *FPGA bitfile* which can be extracted from the product and on methods where the signature is transmitted over the *power pins* of the FPGA. All methods mentioned in this overview paper are described in detail with experimental results in [32] and in the corresponding referenced papers.

References

[1] Design & Reuse, "Catalyst of Collaborative IP Based SoC Design," *URL: http://www.design-reuse.com/.*

[2] Chip Estimate, "ChipEstimate.com," *URL: http://www.chipestimate.com/.*

[3] L. Boney, A. H. Tewfik, and K. N. Hamdy, "Digital Watermarks for Audio Signals," in *International Conference on Multimedia Computing and Systems*, 1996, pp. 473–480. [Online]. Available: citeseer.ist.psu.edu/boney96digital.html

[4] A. T. Abdel-Hamid, S. Tahar, and E. M. Aboulhamid, "A Survey on IP Watermarking Techniques," *Design Automation for Embedded Systems*, vol. 9, no. 3, pp. 211–227, 2004.

[5] D. Ziener and J. Teich, "Evaluation of Watermarking Methods for FPGA-Based IP-cores," University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, D-91058 Erlangen, Germany, Tech. Rep. 01-2005, Mar. 2005.

[6] S. Drimer, "Security for Volatile FPGAs," Nov. 2009.

[7] E. Castillo, L. Parrilla, A. Garcia, A. Loris, and U. Meyer-Baese, "IPP Watermarking Technique for IP Core Protection on FPL Devices," in *International Conference on Field Programmable Logic and Applications, 2006. FPL'06*, 2006, pp. 487–492.

[8] E. Castillo, L. Parrilla, A. Garcia, U. Meyer-Baese, G. Botella, and A. Lloris, "Automated Signature Insertion in Combinational Logic Patterns for HDL IP Core Protection," in *4th Southern Conference on Programmable Logic, 2008*, 2008, pp. 183–186.

[9] A. L. Oliveira, "Techniques for the Creation of Digital Watermarks in Sequential Circuit Designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1101–1117, 2001.

[10] I. Torunoglu and E. Charbon, "Watermarking-based Copyright Protection of Sequential Functions," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 3, pp. 434–440, 2000.

[11] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Signature Hiding Techniques for FPGA Intellectual Property Protection," in *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM, 1998, pp. 186–189.

[12] J. Lach, W. H. Mangione-Smith, and Potkonjak, "Robust FPGA Intellectual Property Protection through Multiple Small Watermarks," in *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. New York, NY, USA: ACM, 1999, pp. 831–836.

[13] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Fingerprinting Techniques for Field-Programmable Gate Array Intellectual Property Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. volume 20, 2001.

[14] D. Saha and S. Sur-Kolay, "Fast Robust Intellectual Property Protection for VLSI Physical Design," in *ICIT '07: Proceedings of the 10th International Conference on Information Technology*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–6.

[15] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. M. Potkonjak, P. A. Tucker, H. Wang, and G. Wolfe, "Watermarking Techniques for Intellectual Property Protection," in *DAC '98: Proceedings of the 35th annual Design Automation Conference*. New York, NY, USA: ACM, 1998, pp. 776–781.

[16] T. V. Le and Y. Desmedt, "Cryptanalysis of UCLA Watermarking Schemes for Intellectual Property Protection," in *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*. London, UK: Springer-Verlag, 2003, pp. 213–225.

[17] T. Kean, D. McLaren, and C. Marsh, "Verifying the Authenticity of Chip Designs with the DesignTag System," in *HOST '08: Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 59–64.

[18] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. M. Potkonjak, P. A. Tucker, H. Wang, and G. Wolfe, "Constraint-Based Watermarking Techniques for Design IP Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 10, pp. 1236–1252, 2001.

[19] D. Kirovski, Y.-Y. Hwang, M. Potkonjak, and J. Cong, "Intellectual Property Protection by Watermarking Combinational Logic Synthesis Solutions," in *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM, 1998, pp. 194–198.

[20] G. Qu, "Publicly Detectable Watermarking for Intellectual Property Authentication in VLSI Design," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, pp. 1363–1367, 2002.

[21] G. Qu and M. Potkonjak, "Fingerprinting Intellectual Property using Constraint-addition," in *DAC '00: Proceedings of the 37th Annual Design Automation Conference*. New York, NY, USA: ACM, 2000, pp. 587–592.

[22] E. Charbon, "Hierarchical Watermarking in IC Design," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1998, pp. 295–298.

[23] D. Kirovski and M. Potkonjak, "Intellectual Property Protection Using Watermarking Partial Scan Chains For Sequential Logic Test Generation," in *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998. [Online]. Available: citeseer.ist.psu.edu/218548.html

[24] A. Rashid, J. Asher, W. H. Mangione-Smith, and M. Potkonj, "Hierarchical Watermarking for Protection of DSP Filter Cores," in *Proceedings of the Custom Integrated Circuits Conference. Piscataway, NJ*. IEEE Press, 1999, pp. 39–45.

[25] R. Chapman and T. S. Durrani, "IP Protection of DSP Algorithms for System on Chip Implementation," *IEEE Transactions on Signal Processing*, vol. 48, no. 3, pp. 854–861, 2000.

[26] S. Meguerdichian and M. Potkonjak, "Watermarking while Preserving the Critical Path," in *DAC '00: Proceedings of the 37th Annual Design Automation Conference*. New York, NY, USA: ACM, 2000, pp. 108–111.

[27] M. M. Khan and S. Tragoudas, "Rewiring for Watermarking Digital Circuit Netlists," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 1132–1137, 2005.

[28] F. Bai, Z. Gao, Y. Xu, and X. Cai, "A Watermarking Technique for Hard IP Protection in Full-custom IC Design," in *International Conference on Communications, Circuits and Systems (ICCCAS 2007)*, 2007, pp. 1177–1180.

[29] A. B. Kahng, S. Mantik, I. L. Markov, M. M. Potkonjak, P. A. Tucker, H. Wang, and G. Wolfe, "Robust IP Watermarking Methodologies for Physical Design," in *DAC '98: Proceedings of the 35th annual Design Automation Conference*. New York, NY, USA: ACM, 1998, pp. 782–787.

[30] D. Ziener and J. Teich, "Power Signature Watermarking of IP Cores for FPGAs," *Journal of Signal Processing Systems*, vol. 51, no. 1, pp. 123–136, April 2008.

[31] D. Ziener, F. Baueregger, and J. Teich, "Using the Power Side Channel of FPGAs for Communication," in *Proceedings of the 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010)*, May 2010, pp. 237–244.

[32] D. Ziener, "Techniques for Increasing Security and Reliability of IP Cores Embedded in FPGA and ASIC Designs," Dissertation, University of Erlangen-Nuremberg, Germany, July 2010, verlag Dr. Hut, Munich, Germany.

[33] D. Ziener, S. Aßmus, and J. Teich, "Identifying FPGA IP-Cores based on Lookup Table Content Analysis," in *Proceedings of 16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, Madrid, Spain, Aug. 2006, pp. 481–486.

[34] M. Schmid, D. Ziener, and J. Teich, "Netlist-Level IP Protection by Watermarking for LUT-Based FPGAs," in *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT 2008)*, Taipei, Taiwan, Dec. 2008, pp. 209–216.

[35] D. Ziener and J. Teich, "FPGA Core Watermarking Based on Power Signature Analysis," in *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT 2006)*, Bangkok, Thailand, Dec. 2006, pp. 205–212.

[36] Digilent Inc., "Spartan-3 Starter Board," *URL: http://www.digilentinc.com/.*

[37] D. Ziener, F. Baueregger, and J. Teich, "Multiplexing Methods for Power Watermarking," in *Proceedings of the IEEE Int. Symposium on Hardware-Oriented Security and Trust (HOST 2010), Anaheim, USA*, June 2010.

Establishing Dedicated Functions on FPGA Devices for High-Performance Cryptography

Tim Güneysu
Horst Görtz Institute for IT-Security, Ruhr-Universität Bochum, Germany
tim.gueneysu@rub.de

Abstract— *This work presents a unique design approach to implement standardized symmetric and asymmetric cryptosystems on modern FPGA devices. While most other FPGA implementations optimize cryptosystems on an algorithmic level for being optimally placed in the generic logic, our primary goal is to shift as many cryptographic operations as possible into specific hard cores that have become available on modern reconfigurable devices. Such dedicated functions provide, for example, large blocks of memory or accelerated arithmetic functions for digital signal processing applications. Using these dedicated function, we present specific design approaches that enable a performance for the symmetric AES block cipher (FIPS 197) of up to 55 GBit/s and a throughput of more than 30.000 scalar multiplications per second for asymmetric Elliptic Curve Cryptography over NIST's P-224 prime (FIPS 186-3).*

1. Introduction

Due to their growing popularity, many cryptographic implementations for FPGAs were developed in the last years, following many different design goals, algorithmic improvements and optimization strategies. However, nearly all of the available proposals elaborate on how to implement an optimized cryptosystem in the generic fabric of an FPGA (i.e., solely by using the configurable logic elements). However, most of the modern FPGA classes provide more than just a sea of generic logic elements – for example, FPGA devices also include dedicated hard cores for clock (re-)generation, I/O transfer, memory and Digital Signal Processing (DSP) functions. In this article, we investigate on alternative ways to implement standardized cryptosystems with FPGAs by making extensively use of these special hard core components. In particular, we will employ memory and DSP components including their special arrangement on-chip to build high-performance cryptographic systems. The following two sections explain why this approach is appealing for the design of powerful cryptosystems.

1.1 Achieving High-Performance

It is evident that the feature of flexibility of Programmable Logic Devices (PLD) comes at the cost of additional gate complexity. More precisely, the FPGA manufacturer needs to take an overhead of a factor between 20 to 40 for a reconfigurable gate into account, compared to its static counterpart on an ASIC. Although this seems to be primarily an issue for the hardware manufacturer in the first place, it also comes at the cost of reduced performance due to increased signal propagation delays. This is the main reason that many dynamic SRAM-based FPGAs only operate at clock frequencies up to 600 MHz while static integrated circuits (i.e., ASICs) easily run at frequencies of 2-3 GHz. As a compensation for this issue, the hardware manufacturers integrated dedicated hard cores for frequently used functions. These hard cores do not provide any reconfigurability but offer a significantly higher performance at reduced resource costs. Thus, in general an FPGA designer should preferably employ dedicated hard cores (if available) instead of implementing the same function with configurable logic. Although the use of device-specific components might slightly reduce the ease of portability between different FPGA devices, it will certainly lead to considerably higher system performance.

1.2 Improving Resource Utilization

FPGAs are sold as an prefabricated circuit that provide a specific amount of logical elements and function hard cores for each device. As already mentioned, most applications implement circuits using merely the configurable logic elements of an FPGA. Many dedicated hard cores of the device, however, remain unused – although they are also part of the FPGA package. More precisely, when a hardware designer selects a specific FPGA device for a design, he is usually limited in his choice to the available device configurations dictated by FPGA manufacturer. So he usually chooses a device which fits best the needs of the main application. However, since FPGAs are generically designed, the chosen device still provides many features and logical components which are actually not required by the design. In particular for applications, in which security functions are actually meant to play only a subsidiary role, it can be beneficial to have a design option for *supplementary* cryptosystems at hand that primarily makes use of the yet unused elements. This allows for a better overall resource utilization of the available logic functions of an FPGA and finally also reduces the costs of the project.

1.3 Scope of this Contribution

In this article we will present implementations for the two most commonly used symmetric and asymmetric cryptosystems in embedded systems. In particular, we laid special emphasis on the implementation of established and standardized cryptosystems that can be directly employed in products without any (known) security issues.

First, we propose an alternative design for the Advanced Encryption Standard which is considered the most popular block cipher nowadays due to established the NIST FIPS 197 standard [28]. The target platform for our special design strategy will be a Xilinx Virtex-5 FPGA [37]. For this device, we will implement AES based on the 32-bit T-Table method [8, Section 4.2] by taking advantage of large dual-ported memories and Digital Signal Processing (DSP) hard cores. Unlike conventional AES design approaches for these FPGAs (e.g., [4]), our design is therefore especially suitable for applications where configurable logic elements are the limiting resource, but yet not all embedded memory and DSP blocks are used.
Note in this context that other authors already proposed the use the T-Table method for AES also on FPGAs [6], [26], [15], [5]. However in contrast to these designs, our approach maps the *complete* AES data path onto embedded elements contained in Virtex-5 FPGAs. This strategy provides most savings in logic and routing resources and results in the highest data throughput on FPGAs reported in open literature.

Second, we discuss the efficient implementation of asymmetric Elliptic Curve Cryptography (ECC) over standardized NIST primes (according to NIST FIPS 186-3 [29]) based on the hard cores located in common Virtex-4 FPGAs. Since asymmetric cryptographic algorithms are known to be extremely arithmetic intensive, the exclusive use of DSP blocks for the arithmetic computations offers a significant acceleration compared to conventional designs. In general, to achieve real high-performance ECC (i.e., to reach less than 500μs per point multiplication) on (affordable) embedded computing platforms was an open challenge for quite a long time. This holds especially for ECC over prime fields, which are often preferred over binary fields due to standards. Due to the enormous design complexity, high-performance implementations for ECC over prime fields often require a huge number of logic resources and were only feasible on the largest available FPGA devices. Some implementations have already attempted to address this problem by using available arithmetic functions in the reconfigurable device for a few parts of the computations, like built-in 18×18 multipliers [24]. But other components of the circuitry for field addition, subtraction and inversion have been still implemented in the generic logic of an FPGA. This is different for the design approach described in this article: we will discuss the complete relocation of the arithmetic intensive operations for ECC into dedicated hard cores of the FPGA.

1.4 Outline

This article is structured as follows: we first review the relevant and existing literature in Section 2, before we briefly introduce the properties and functionality of hard cores in modern FPGAs in Section 3. Then, in Section 4 we discuss our alternative approach to implement the most common symmetric cryptosystem based on these special FPGA functions, namely the AES block cipher. Next, we present in Section 5 how to use these functions to build an efficient design for asymmetric Elliptic Curve Cryptosystems (ECC), before we conclude in Section 6.

2. Related Work

Since FPGA technology has become mature to provide a suitable platform for cryptographic systems, many designs for AES and ECC have been proposed in the open literature. At this point, our goal is not to review all available implementations but rather to briefly introduce general concepts that highlight the differences in design with respect to this article. For a more thorough discussion on cryptographic system implementation, please refer to, for example, the discussion on AES designs in [21], or the survey on elliptic curve cryptosystems in [10].

2.1 Strategies to Design AES

Most AES designs are usually straightforward implementations of a single AES round or loop-unrolled, pipelined architectures for FPGAs utilizing a vast amount of user logic elements [14], [22], [20]. Particularly, the required 8×8 S-Boxes of the AES are mostly implemented in the Lookup Tables (LUT) of the user logic usually requiring large portions of the reconfigurable logic. For example, the authors of [35] report 144 LUTs (4-input LUTs) to implement a single AES S-Box what accumulates to 2304 LUTs for a single AES round. More advanced approaches [25], [35], [6], [5] used the on-chip memory components of FPGAs, implementing the S-Box tables in separate RAM sections on the device. Since RAM capacities were limited in previous generations of FPGAs, the majority of implementations only mapped the 8×8 S-Box into the memory (when not directly computed on-the-fly by extension field arithmetic) while all other AES operations like ShiftRows, MixColumns and the AddRoundKey are realized using traditional user logic what proved costly in terms of flip-flops and LUTs.
We will now discuss and categorize published AES implementations according to their performance and resource consumption (and implicitly, if a small 8-bit, medium 32-bit or wide 128-bit data-path is used).

To our knowledge, only few implementations ([15], [32], [5]) transferred the software architecture based on the T-table to FPGAs. Due to the large tables and the restricted memory capacities on those devices, certain functionality must be still encoded in user logic up to now (e.g., the multiplication elimination required in the last AES round). The new features of Virtex-5 devices provide wider memories and more advanced logic resources. The concept presented here and in our original work [11] is thus the first T-table-based AES-implementation that efficiently uses mostly device-specific features and minimizes the need for generic logic elements. We will provide three individual solutions that address each of the design categories mentioned above – minimal resource usage, area-time efficiency and high -throughput.

2.2 Strategies to Design ECC

In the case of high speed architectures for ECC, most implementation primarily address elliptic curves over binary fields $GF(2^m)$ since the arithmetic is more hardware-friendly due to carry-free computations [30], [12]. However, this work focuses solely on the prime field $\mathbb{F}_p$ and enables fast modular arithmetic by use of Mersenne-like primes. First implementations for ECC over prime fields $\mathbb{F}_p$ have been proposed by [31], [33] demonstrating ECC processors built completely in reconfigurable logic. The contribution by [24] proposes a high-speed ECC crypto core for arbitrary moduli with up to 256-bit length designed on a large number of built-in multiplier blocks of FPGA devices providing a significant speedup for modular multiplications. However, other field operations have been implemented in the FPGA fabric, resulting in a very large design (15,755 slices and 256 multiplier blocks) on a large Xilinx XC2VP125 device. The architecture presented in [9] was designed to achieve a better trade-off between performance and resource consumption. According to the contribution, an area consumption of only 1,854 slices and a maximum clock speed of 40 MHz can be achieved on a Xilinx Virtex-2 XC2V2000 FPGA for a parameter bit length of 160 bit.

Our approach to implementing an FPGA-based ECC engines was to shift *all* field operations into the integrated DSP building blocks available on modern FPGAs. This strategy frees most configurable logic elements on the FPGA for other applications and requires less power compared to a conventional design. In addition to that, this architecture offers a very high performance for ECC computations over prime fields with up to 256-bit security in reconfigurable logic (i.e., a single scalar multiplication takes 495 μs).

The approach presented in our original work [17] and this article has been picked up by Hamilton and Marnane in [18] using the GLV method for point arithmetic and a modular Hiasat multiplier for (non-standardized) Mersenne primes. In this work, the authors present a design that requires 1439 slices and 56 DSPs on an Xilinx Virtex-5 FPGA that computes a point multiplication over a 256-bit elliptic curve in only 188 μs.

3. Embedded Hard Cores in Modern FPGAs

In this section, we will introduce the functionalities of embedded and dedicated hard core functions that are provided by many modern FPGAs. Since their invention in 1985, FPGAs merely consist of a large sea of generic, reconfigurable logic implementing simple gate functions. Although devices became more complex over time, there are still designs which were preferably placed externally in separate peripheral devices since it was too inefficient to implement them with this generic gate logic. Examples of these functions blocks are large memory blocks, hard microprocessors, and fast serial transceivers. Thus, FPGA manufacturers integrate more and more of these dedicated function blocks into modern devices

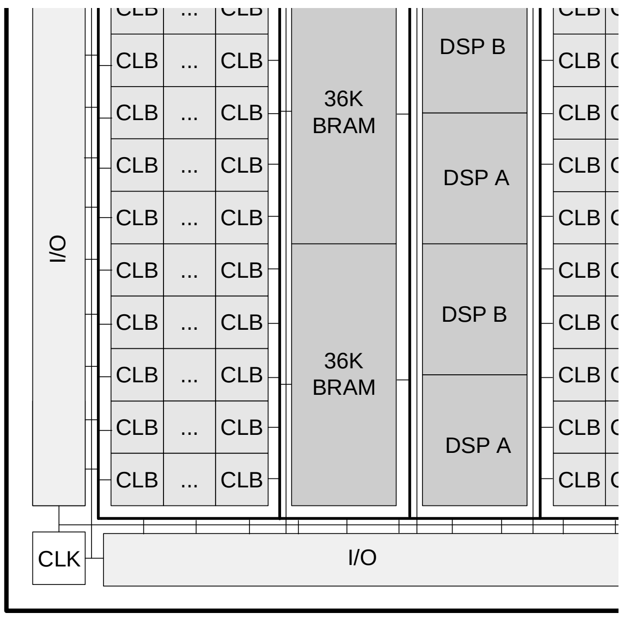


Fig. 1: Simplified structure of Xilinx Virtex-5 FPGAs.

to avoid the necessity of chip extensions on the board. Figure 1 depicts the simplified structure of recent Xilinx Virtex-5 FPGAs including separate columns of additional function hard cores for memory (BRAM) and arithmetic DSP operations. Note that other FPGA classes, like Spartan-3 or Virtex-4 have a similar architecture despite variations in dimensions and features of the embedded elements. In Virtex-4 and Virtex-5 devices, the DSP blocks are grouped in pairs that span the height of four or five configurable logic blocks (CLB), respectively. The dual-ported BRAM matches the height of the pair of DSP blocks and supports a fast data path between memory and the DSP elements.

In particular interest is the use of these memory elements and DSP blocks for efficient boolean and integer arithmetic operations with low signal propagation time. Large devices of Xilinx's Virtex-4 and Virtex-5 class are equipped with up to thousand individual function blocks of these dedicated memory and arithmetic units. Originally, the integrated DSP blocks – as indicated by their name – were designed to accelerate Digital Signal Processing (DSP) applications, e.g., Finite Impulse Response (FIR) filters, etc. However, these arithmetic units can be programmed to perform universal arithmetic functions not limited to the scope of DSP filter applications; they support generic multiplication, addition and subtraction of (un)signed integers. Depending on the FPGA class, common DSP component comprises an l_M-bit signed integer multiplier coupled with an l_A-bit signed adder where the adder supports a larger data path to allow accumulation of multiple subsequent products (i.e., $l_M < l_A$). More precisely, Xilinx Virtex-4 FPGAs support 18-bit unsigned integer multiplication (yielding 36-bit products) and three-input addition, subtraction

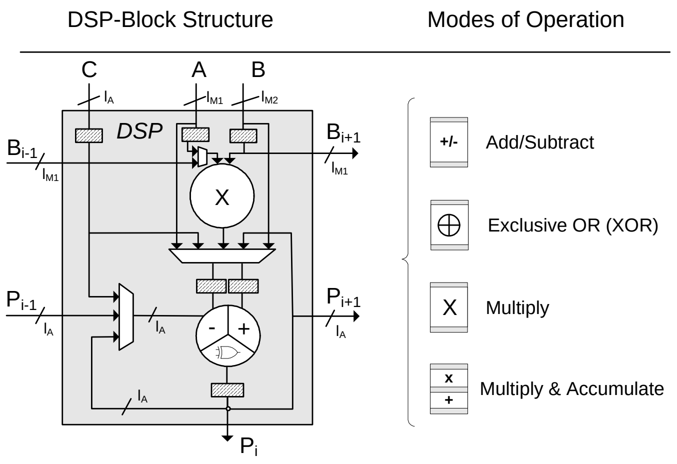


Fig. 2: Generic and simplified structure of DSP-blocks of advanced FPGA devices.

or accumulation of unsigned 48-bit integers. Virtex-5 devices offer support for even wider 25×18-bit multiplications. Since DSP blocks are designed as an embedded element in FPGAs, there are several design constraints which need to be obeyed for maximum performance with the remaining logic, e.g., the multiplier and adder block should be surrounded by pipeline registers to reduce signal propagation delays between components. Furthermore, since they support different input paths, DSP blocks can operate either on external inputs A, B, C or on internal feedback values from accumulation or the result P_{j-1} from a neighboring DSP block. Figure 2 shows the generic DSP-block and a small selection of possible modes of operations available in recent Xilinx Virtex-4/5 FPGA devices.

When using DSP blocks to develop high-performance cryptographic designs, there are several criteria which should be obeyed to exploit their full performance for complex arithmetic. Note that the following aspects have been designed to target the requirements of Xilinx Virtex FPGAs:

1) *Build DSP cascades:* Neighboring DSP blocks can be cascaded to widen or extend their atomic operand width (e.g., from 18-bit to 256-bit).
2) *Use DSP routing paths:* DSPs have been provided with inner routing paths connecting two adjacent blocks. It is advantageous in terms of performance to use these paths as frequently as possible instead of using FPGA's general switching matrix for connecting logic blocks.
3) *Consider DSP columns:* Within a Xilinx FPGA, DSPs are aligned in columns, i.e., routing paths between DSPs within the same column are efficient while a switch in columns can lead to degraded performance. Hence, DSP cascades should not exceed the column width (typically 32/48/64 DSPs per column, depending on the device).
4) *Use DSP pipeline registers:* DSP blocks feature pipeline stages which should be used to achieve the maximum clock frequency supported by the device (up to 550 MHz).
5) *Use different clock domains:* Optimally, DSP blocks can be operated at maximum device frequency. This is not necessarily true for the remainder of the design so that separate clock domains should be introduced (e.g. by halving the clock frequency for control signals) to address the critical paths in each domain individually.

4. Implementing the AES Block Cipher

The following AES cipher implementation is almost exclusively based on embedded memory and arithmetic units embedded of Xilinx Virtex-5 FPGAs. It is designed to match specifically the features of this modern FPGA class – yielding one of the smallest and fastest FPGA-based AES implementation reported up to now – with minimal requirements on the (generic) configurable logic of the device.

4.1 Mathematical Background

Although AES is a well-established algorithm, we will briefly review the relevant operations of the AES block cipher to keep this contribution self-contained. AES was designed as a Substitution-Permutation Network (SPN) and uses between 10, 12 or 14 rounds (depending on the key length with 128, 192 and 256 bits, respectively) for encryption and decryption of one 128-bit block. In a single round, the AES operates on all 128 input bits. Fundamental operations of the AES are performed based on byte-level field arithmetic over the Galois Field $GF(2^8)$ so that operands can be represented in 8-bit vectors. Processing these 8-bit vectors serially allows implementations on very small processing units, while 128-bit data paths allow for maximum throughput. The output of such a round, or state, can be represented as a 4×4 matrix A of bytes $a_{i,j}$ where i and j denotes the corresponding row and column, respectively. Four basic operations process the AES state A in each round:

1) *SubBytes*: all input bytes of A are substituted with values from a non-linear 8×8-bit S-Box.
2) *ShiftRows*: the bytes of rows R_i are cyclically shifted to the left by 0, 1, 2 or 3 positions.
3) *MixColumns*: columns $C_j = (a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})$ are matrix-vector-multiplied by a matrix of constants in $GF(2^8)$.
4) *AddRoundKey*: a round key K_i is added to the input using $GF(2^8)$ arithmetic.

The sequence of these four operations defines an AES round, and they are iteratively applied for a full encryption or decryption of a single 128-bit input block. Since some of the operations above rely on $GF(2^8)$ arithmetic it is possible to combine them into a single complex operation. In addition to the Advanced Encryption Standard, an alternative representation of the AES operation for software implementations on 32-bit processors was proposed in [8, Section 4.2] based on the use of large lookup tables. This approach requires four lookup tables with 8-bit input and 32-bit output for the four round transformations, each the size of 8 Kbit. According to [8], these transformation tables T_i with $i = 0..3$ can be computed as follows:

$$T_0[x] = \begin{bmatrix} S[x] \times 02 \\ S[x] \\ S[x] \\ S[x] \times 03 \end{bmatrix} \quad T_1[x] = \begin{bmatrix} S[x] \times 03 \\ S[x] \times 02 \\ S[x] \\ S[x] \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} S[x] \\ S[x] \times 03 \\ S[x] \times 02 \\ S[x] \end{bmatrix} \quad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \times 03 \\ S[x] \times 02 \end{bmatrix}$$

In this notation, $S[x]$ denotes a table lookup in the original 8×8-bit AES S-Box (for a more detailed description of this AES optimization see NIST's FIPS-197 [28]). The last round, however, is unique since it omits the MixColumns operation, so we need to give it special consideration. There are two ways for computing the last round, either by "reversing" the MixColumns operation from the output of a regular round by another multiplication in $GF(2^8)$, or creating dedicated T-tables for the last round. The latter approach will allow us to maintain the same data path for all rounds, so – since Virtex-5 devices provide larger memory blocks than former devices – we chose this method and denote these T-tables as $T_{[j]'}$. With all T-tables at hand, all transformation steps of a single AES round can be redefined as

$$\begin{aligned} E_j \quad = \quad & K_{r[j]} \oplus T_0[a_{0,j}] \oplus T_1[a_{1,(j+1 \bmod 4)}] \oplus \\ & T_2[a_{2,(j+2 \bmod 4)}] \oplus T_3[a_{3,(j+3 \bmod 4)}] \end{aligned} \qquad (1)$$

where $K_{r[j]}$ is a corresponding 32-bit subkey and E_j denotes one of four encrypted output *columns* of a full round. From this formula it is obvious that based on only four T-table lookups and four XOR operations, a 32-bit output E_j of the AES round can be computed. To obtain the result of a full round, Equation (1) must be performed four times on all 16 input bytes.

Input data to an AES encryption can be defined as four 32-bit column vectors $C_j = (a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})$ with the output similarly formatted in column vectors. According to Equation (1), these input column vectors need to be split into individual bytes since all bytes are required for the computation steps for different E_j. For example, for column $C_0 = (a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0})$ the first byte $a_{0,0}$ is part of the computation of E_0, the second byte $a_{1,0}$ is used in E_3, etc. Since fixed (and thus simple) data paths are preferable in hardware implementations, we have rearranged the operands of the equation to align the bytes according to the input columns C_j when feeding them to the T-table lookup. In this way, we can implement a unified data path for computing all four E_j for a full AES round. Thus, Equation (1) transforms into

$$\begin{aligned} E_0 &= K_{r[0]} \oplus T_0(a_{0,0}) \oplus T_1(a_{1,1}) \oplus T_2(a_{2,2}) \oplus T_3(a_{3,3}) \\ &= (a'_{0,0}, a'_{1,0}, a'_{2,0}, a'_{3,0}) \\ E_1 &= K_{r[1]} \oplus T_3(a_{3,0}) \oplus T_0(a_{0,1}) \oplus T_1(a_{1,2}) \oplus T_2(a_{2,3}) \\ &= (a'_{0,1}, a'_{1,1}, a'_{2,1}, a'_{3,1}) \\ E_2 &= K_{r[2]} \oplus T_2(a_{2,0}) \oplus T_3(a_{3,1}) \oplus T_0(a_{0,2}) \oplus T_1(a_{1,3}) \\ &= (a'_{0,2}, a'_{1,2}, a'_{2,2}, a'_{3,2}) \\ E_3 &= K_{r[3]} \oplus T_1(a_{1,0}) \oplus T_2(a_{2,1}) \oplus T_3(a_{3,2}) \oplus T_0(a_{0,3}) \\ &= (a'_{0,3}, a'_{1,3}, a'_{2,3}, a'_{3,3}) \end{aligned}$$

where $a_{i,j}$ denotes an input byte, and $a'_{i,j}$ the corresponding output byte after the round transformation. However, the unified input data path still requires a look-up to all of the four T-tables for the second operand of each XOR operation. For example, the XOR component at the first position of the sequential operations E_0 to E_3 and thus requires the lookups $T_0(a_{0,0})$, $T_3(a_{3,0})$, $T_2(a_{2,0})$ and $T_1(a_{1,0})$ (in this order) and the corresponding round key $K_{r[j]}$. Though operations are aligned for the same input column now, it becomes apparent that the bytes of the input column are not processed in canonical order, i.e., bytes need to be swapped for each column $C_j = (a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})$ first before being fed as input to the next AES round. Note that the given transpositions are static so that they can be efficiently hardwired in our implementation.

Finally, we need to consider the XOR operation of the input key and the input 128-bit block which is done prior to the round processing. Initially, we will omit this operation when reporting our results for the round function. However, adding the XOR to the data path is simple, either by modifying the AES module to perform a sole XOR operation in a preceding cycle, or – more efficiently – by just adding an appropriate 32-bit XOR which processes the input columns prior being fed to the round function.

We now delve into the implementation details. For the design described in the following section, we exploit some of these features like growing capacities of integrated Block RAMs as well as logical units of DSP hard cores.

4.2 Implementation

In Section 4.1, we have introduced the T-table method for implementing the AES round most suitable for 32-bit microprocessors. Now, we will demonstrate how to adapt this technique into modern reconfigurable hardware devices in order to achieve high throughput for modest amounts of resources. Our architecture relies on dual ported 36 Kbit Block RAMs (BRAM) (with independent address and data buses for the same stored content) and DSP blocks. The fundamental idea of this work is that the 8 to 32-bit lookup followed by a 32-bit XOR AES operation perfectly matched this architectural alignment of Virtex-5 FPGAs. Based on these primitives, we developed a basic AES module that performs a quarter (one column) of an AES round transformation given by Equation (1).

4.2.1 Basic Module

Figure 3 shows the architecture of our basic module. The most complex part is the alignment of the inputs: here, four bytes $a_{i,j}$ are selected from the current state A at a time and passed to the BRAMs for the T-table lookup. Since the order of bytes $a_{i,j}$ vary for each column computation E_j, this requires a careful design of the input logic since it need to support selection from all four possible byte positions of each 32-bit column input. Hence, instead of implementing a complex input logic, we modified the order of operations according to Equations (2) exploiting that addition in $GF(2^m)$(i.e., XOR) is a commutative operation. When changing the order of operations dynamically for each computation of E_j, this requires that all four T-table lookups with their last-round T-table counterparts are stored in each BRAM. However, that would require to fit a total of eight 8 Kbit T-tables in a single 36 Kbit dual-port RAM. As discussed in Section 4.1, for performance and resource efficiency reasons we opted against adding out the MixColumn operations from the stored T-tables and preferred a solution so that all BRAM can provide all eight required tables. Utilizing the fact that all T-tables are byte-wise transpositions of each other, we can produce the output of T_1, T_2 and T_3 by cyclically byte-shifting of the BRAM's output for T-table T_0. Using this observation, we only store T_0 and T_2 and their last-round counterparts $T_{0'}$ and $T_{2'}$ in a single BRAM. Using a single byte circular right rotation $(a, b, c, d) \rightarrow (d, a, b, c)$, T_0 becomes T_1, and T_2 becomes T_3. The same holds also for the last round's T-tables. In hardware, this only requires a 32-bit 2:1 multiplexer at the output of each BRAM with a select signal from the control logic. For the last round, a control bit is connected to a high order address bit of the BRAM to switch from the regular T-table to the last round's T-table. A dual-port 32 Kbit BRAM with three control bits, and a 2:1 32-bit mux allows us to output all T-table combinations. Using two such BRAMs with identical content, we get the necessary lookups for four columns, each capable of performing all four T-table lookups in parallel.

Note that both the BRAMs and DSP blocks provide internal input and output registers for pipelining along the data path so that we include these registers without occupation of any flip-flops in the fabric. At this point, we already had six pipeline stages that could not have been easily removed if our goal was high throughput. Instead of trying to reduce pipeline stages for lower latency, we opted to add two more so that we are able to process two input blocks at the same time, doubling the throughput for separate input streams. One of these added stages is the 32-bit register after the 2:1 multiplexer that shifts the T-tables at the output of the BRAM. A full AES operation is implemented by operating the basic construct with an added feedback scheduling in the data path.

The first column output E_0 becomes available after the eighth clock cycle and is fed back as input for the second round. For the second round, the control logic switches the 2:1 input multiplexer for the feedback path rather than the external input. In the eight pipeline stages we can process two separate AES blocks, since we only need 4 stages to process the 128-bit of one block. This allows us to feed two consecutive 128-bit blocks one after another, in effect doubling our throughout without any additional complexity.

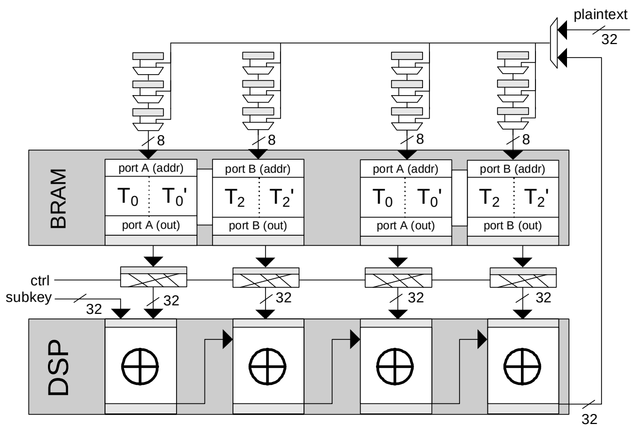


Fig. 3: The complete basic AES module consisting of 4 DSP slices and 2 dual-ported Block Memories. Tables T_1 and T_3 are constructed on-the-fly using byte shifting from tables T_0 and T_2 in the block memory, respectively.

Up to now we focused on the encryption process, though decryption is quite simply achieved with minor modifications to the circuit. As the T-tables are different for encryption and decryption, storing them all would require double the amount of storage what is not desirable. Recall, however, that any T_i can be converted into T_j simply by shifting the appropriate amount of bytes. The most straightforward modification to the design is to replace the 32-bit 2:1 mux at the output of the BRAM with a 4:1 mux such that all byte transpositions can be created. Then, we load the BRAMs with T_i^E, $T_{i'}^E$, T_i^D and $T_{i'}^D$, where T^E and T^D denote encryption and decryption T-tables, respectively, with their corresponding last round counterparts. Note, that this does not necessarily increase the data path due to the 6-input LUTs in the CLBs of a Virtex-5 device. Based on 6-input LUTs, a 4:1 multiplexer can be as efficiently implemented as a 2:1 multiplexer with only a single stage of logic. An alternative is to dynamically reconfigure the content of the BRAMs with the decryption T-tables; this can be done from an external source, or even from within the FPGA using the internal configuration access port (ICAP) [37] with a storage BRAM for reloading content through the T-table BRAMs' data input port.

Finally, the AES specification requires an initial key addition of the input with the main key which has not covered by the AES module so far. Most straightforward, this can be done by adding one to four DSP blocks (alternatively, the XOR elements can be implemented in CLB logic) as a pre-stage to the round operation.

4.2.2 Round and Loop-Unrolled Modules

Since the single AES round requires the computation of four 32-bit columns, we can replicate the basic construct four times and add 8, 16, and 24-bit registers at the inputs of the columns. All instances are connected to a 128-bit bus (32 bits per instance) of which selected bytes are routed to corresponding instances by fixed wires. Note that only one byte per 32-bit column output remains within the same instance, the other three bytes will be processed by the other instances in the next round. The latency of this construction is still 80 clock cycles as before, but allows us to interleave eight 128-bit inputs instead of two. In contrast to the basic module, however, the input byte arrangements allow that the T-tables remain static so the 32-bit 2:1 multiplexers are no longer required. This simplifies the data paths between the BRAMs and DSP blocks since any byte shifting can be fixed in routing. The control logic is simple as well, comprising of a 3-bit counter and a 1-bit control signal for choosing the last round's T-tables.

Finally, we implemented a fully unrolled AES design for achieving maximum throughput by connecting ten instances of the round design presented above. We yield an architecture with an 80-stage pipeline, producing a 128-bit output every clock cycle at a resource consumption of 80 BRAMs and 160 DSP blocks.

4.2.3 Key Schedule Implementation

Considering the key schedule, many designers (e.g., [4]) prefer a shared S-Box and/or data path for deriving subkeys and the AES round function. This approach needs additional multiplexing and control signals to switch the central data path between subkey computations and data encryption which may lead to decreased performance in practice. Furthermore, key precomputation is mostly preferred over on-the-fly key expansion because the first relaxes the constraints on data dependencies, i.e., the computation is only dependent on the availability of the previous state and not additionally on completion of key computations.

In case that high throughput is not required but the key schedule needs to be precomputed on chip without adversely increasing logic resource utilization, our basic AES module can be modified to support key generation. Remember that we already store T-tables $T_{[0..3]'}$ for the last round in the BRAMs without the MixColumns operation so that the values of these tables are basically a byte-rotated 8-bit S-Box value. These values are perfectly suited for generating a 32-bit round key from S-Box lookups and our data path has been specifically designed for 32-bit XOR operations based on the DSP unit. Hence, with additional input multiplexers, control logic and a separate BRAM as key-store, we can integrate a key scheduler in our existing design. However, although this is possible, the additional overhead (i.e., additional multiplexers) will potentially degrade the performance of the AES rounds.

The second approach for the key schedule is a dedicated circuit to preserve the regularity of the basic module and the option to operate the design at maximum device frequency. For a minimal footprint, we propose to add another dual-ported BRAM to the design used for storing the expanded 32-bit subkeys (44 words for AES-128), the round constants (10 32-bit values) and S-Box entries with 8-bit each. The design of

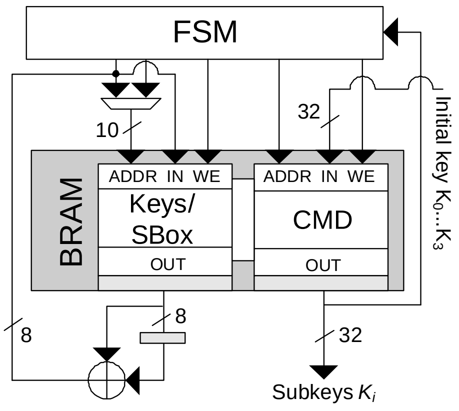


Fig. 4: Block diagram of the key schedule implementation. Complex instructions of the finite state maschine, S-boxes, round constants and 32-bit subkeys are stored in the dual-port BRAM.

our key schedule implementation is shown in Figure 4: port A of the BRAM is 32-bit wide which feeds the subkeys to the AES module, while port B is configured for 8-bit I/O enabling a minimal data path for the key expansion function. With an 8-bit multiplexer, register and XOR connected to port B data output, we can construct a minimal and byte-oriented key schedule that can compute the full key expansion.

The sequential and byte-wise nature of this approach for loading and storing the appropriate bytes from and to the BRAM requires a complex state machine. Recall that the BRAM provides 36 Kbits of memory of which 1408 to 1920 bits are required for subkeys (for AES-128 and AES-256, respectively), 2048 bits for S-Box entries and 80 bits for round constants, so the BRAM can still be used to store further data. Thus, we have decided that the most area economic approach is to encode all the required memory addresses as well as control signals for multiplexers and registers as 32-bit instructions, and store these instruction words in the yet unused portion of the BRAM. This method also ensures a constant and uniform signal propagation in all control signals since they do not need to be generated by combinatorial logic but loaded (and hardwired) from the BRAM. In particular, complex state machines and the latency within their combinatorial circuitry are usually the bottleneck of high-performance implementations since nested levels of logic to generate dozens of control signals are likely to emerge as the critical path. By encoding this complexity into the BRAM, we could avoid this performance degrade. Like the AES module it can be operated at full device frequency of 550 MHz and with the complete key expansion function requiring 524 clock cycles for AES-128.

4.3 Results

Our designs target a Virtex-5 LX30 and SX95T devices at their fastest speed grade (-3) using Xilinx Synthesis Technology (XST) and the ISE 9.2 implementation flow. For simulation we used Mentor's ModelSim 6.2g for both behavioral and post place-and-route stages. In addition, the routes to input and output ports were ignored for timing (“TIG” constraint) during synthesis, as we consider the cores as a stand-alone function.

The basic AES module as shown in Figure 3 passed timing (post place-and-route) for a frequency just over 550 MHz, the maximum frequency rating of the device. The design requires the following resources: 247 flip-flops, 96 ($8 \cdot 3 \cdot 4$) for the input shift registers plus 128 ($4 \cdot 32$) for the pipeline stages in between the BRAMs and DSPs, with the rest used for control logic; 275 look-up tables, mostly functioning as multiplexers; and finally, two 36 Kbit dual-port BRAM (32 Kbit used in each) and four DSP blocks. We calculate throughput as follows: given that there are 80 processing cycles operating at 550 MHz and we maintain state of 256 bits in the pipeline stages, we achieve $550 \cdot 10^6 \cdot 256/80 = 1.76$ Gbit/s of throughput. This assumes that the pipeline stages are always full, meaning that the module is processing two 128-bit inputs at any given time; if only one input is processed, the throughput is halved. As we have mentioned, the eight pipeline stages were implemented for the purpose of interleaving two inputs or using a parallel mode of operation like Counter (CTR) mode, though the designer can remove pipeline stages to reduce resources. Removing pipeline stages reduces latency, though it may also reduce the maximum frequency, so there is a trade-off that needs to be assessed according to the application.

Finally, the unrolled implementation produces 128-bits of output every clock cycle once the initial latency is complete. We have experimented with eliminating the pipeline stage between the BRAM and DSP to see if it adversely affects performance; this will save us 5,120 registers. We found that the performance degradation is minimal, with the added benefit of having an initial latency of only 70 clock cycles instead of 80. The resulting throughput is $430 \cdot 10^6 \cdot 128 = 55$ Gbit/s. This design operates at a maximum frequency of over 430 MHz and uses 992 flip-flops, 672 look-up tables, 80 36 Kbit BRAMs (only 16 Kbit in each for dec/enc or 32 Kbit for both), and 160 DSP blocks; the same balancing act of FF-LUT ratio by the synthesizer occurs here as well. There are very few flip-flops and LUTs compared to what is available in the large SX95T device: 1.68% and 1.14%, respectively, though we use 32% of BRAMs and 25% of DSP blocks.

Our results are summarized in Table 1. We extended the list of our findings with previous result available in the literature. However, please be aware of the unfairness of direct comparison. Due to the different architectures of Spartan-2/3 (S2/S3) and Virtex-2/E Pro (V2/V2P/VE) and Virtex-5 (V5) FPGAs we cannot directly compare soft metrics like "‘slices"’. This is due to the different concepts of contained LUTs (6-input LUTs in Virtex-5 and 4-input LUTs in all others) as well as the different number of LUTs and flip-flops per slice (e.g., a slice in Spartan and Virtex-2 FPGAs consists a combination of 2 LUTs/FF but 4 LUTs/FF in Virtex-5 devices). Even the amount of memory contained in BRAMs is different: Virtex-5 FPGAs provide block memories capable to store 36 KBits of data, twice as much as with Virtex-2 devices. Beside device-specific differences, the implementations also target different applications and requirements: some can operate in more complex modes of operations, others include a key schedule in the data path or support natively encryption and decryption with the same circuit. This all leads to the conclusion that comparisons with other publications based on different FPGAs and application goals are mostly misleading, e.g., meaningful comparisons are only possible when the same device/technology is used and the compared cryptographic implementations comply to a predefined application setup or framework.

Note that our results for the AES modules are all based on the assumption that the set of subkeys are externally provided. In case that all subkeys should be generated on the same device, these modules can be augmented with the key schedule precomputing all subkeys and storing them in a dedicated BRAM. As shown in Section 4.2.3, our key schedule is optimized for a minimal footprint and allows operation at maximum device frequency of 550 MHz. The complexity of the state machine, which is the most expensive part in terms of logic, is mostly hidden within the encoded 32-bit instructions stored in the BRAM. Hence, since only a small stub of the state machine in the user logic is required to address the individual instructions words, the overall resources consumption of the full key schedule is only 1 BRAM, 55 LUTs and 41 flip-flops. All key schedule related data is presented in Table 2 supporting key sizes of 128, 192 and 256 bits.

5. Implementing ECC Cryptosystems

In this section, we present a new design strategy for an FPGA-based, high performance ECC implementation over standardized prime fields according to NIST FIPS 186-3. Our architecture makes intensive use of embedded arithmetic units in FPGAs originally designed to accelerate digital signal processing algorithms.

5.1 Mathematical Background

In this section, we will briefly introduce to the mathematical background relevant for this work. We will start with a short review of the Elliptic Curve Cryptosystems (ECC). Please note that only ECC over prime fields $\mathbb{F}_p$ will be subject of this work since binary extension fields $GF(2^m)$ require binary arithmetic which is not (yet) natively supported by DSP blocks.

Let p be a prime with $p > 3$ and $\mathbb{F}_p = GF(p)$ the Galois Field over p. Given the Weierstrass equation of an elliptic curve

$$\mathcal{E} : y^2 \quad = \quad x^3 + ax + b,$$

with $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \neq 0$, points $\mathcal{P}_i \in \mathcal{E}$, we can compute tuples (x, y) also considered as points on this elliptic curve $\mathcal{E}$. Based on a group of points defined over this curve, ECC arithmetic defines the addition $\mathcal{R} = \mathcal{P} + \mathcal{Q}$ of two points $\mathcal{P}, \mathcal{Q}$ using the *tangent-and-chord* rule as group

	Design	Dec/ Key	FPGA	Resources					f	Perf.
				slices	LUT	FF	BRAM	DSP	MHz	Gbit/s
Ours	**Compact**[a]	○/○	V5	93	274	245	2	4	550	1.76
	Round[a]	○/○	V5	277	204	601	8	16	485	6.21
	Unrolled	●/○	V5	428	672	992	80	160	430	55
Basic	Good *et al.*	●/●	S2	67	*n/a*	*n/a*	2	0	67	0.002
	Chodowiec *et al.*	●/●	S2	222	*n/a*	*n/a*	3	0	60	0.166
	Rouvroy *et al.*	●/●	S3	163	293	126	3	0	71	0.208
	Algotronix	○/○	V5	161	*n/a*	*n/a*	2	0	250	0.8
Round	Standaert *et al.*	○/●	VE	2257	3846	2517	2	0	169	2.008
	Helion	○/●	V5	349	*n/a*	*n/a*	0	0	350	4.07
	Bulens *et al.*	○/●	V5	400	*n/a*	*n/a*	0	0	350	4.1
	Chaves *et al.*	●/○	V2P	515	*n/a*	*n/a*	12	0	182	2.33
Unrolled	Kotturi *et al.*	●/○	V2P	10816	*n/a*	*n/a*	400	0	126	16
	Järvinen *et al.*	○/●	V2	10750	*n/a*	*n/a*	0	0	139	17.8
	Hodjat *et al.*	○/○	V2P	5177	*n/a*	*n/a*	84	0	168	21.5
	Chaves *et al.*	●/○	V2P	3513	*n/a*	*n/a*	80	0	272	34.7

[a] For "basic" and "round" implementations, decryption can be achieved by adding 32-bit muxes in the data path between BRAM and DSP.

Table 1: Our results along with recent academic and commercial implementations. Decryption (Dec.) and Key expansion (Key) are included when denoted by ●, by ○ otherwise. Note the structural differences between the FPGA types: Virtex-5 (V5) has 4 FF and 4 6-LUT per slice and a 36 Kbit BRAM, while Spartan-3 (S3), Virtex-E (VE), Virtex-II (PRO) (V2/V2P) has 2 FF and 2 4-LUT per slice and an 18 Kbit BRAM. Spartan-II (S2) devices only provide 4 KBit BRAMs.

Key schedule	Resources					f	Cycles
	slices	LUT	FF	BRAM	DSPs	(MHz)	
AES-128	37	55	41	1	0	550	524
AES-192							628
AES-256							732

Table 2: Implementation results for the AES key schedule. Most state machine encoding and control logic has been incorporated into the BRAM to save on logic resources.

operation. This group operation distinguishes the case for $\mathcal{P} = \mathcal{Q}$ (*point doubling*) and $\mathcal{P} \neq \mathcal{Q}$ (*point addition*). Furthermore, formulas for these operations vary for affine and projective coordinate representations. Since affine coordinates require the availability of fast modular inversion, we will focus on projective point representation to avoid the implementation of a costly inversion circuit. Given two points $\mathcal{P}_1, \mathcal{P}_2$ with $\mathcal{P}_i = (X_i, Y_i, Z_i)$ and $\mathcal{P}_1 \neq \mathcal{P}_2$, the sum $\mathcal{P}_3 = \mathcal{P}_1 + \mathcal{P}_2$ is defined by

$$\begin{aligned} A &= Y_2Z_1 - Y_1Z_2 & C &= X_2Z_1 - X_1Z_2 \\ B &= A^2Z_1Z_2 - C^3 - 2C^2X_1Z_2 & X_3 &= BC \\ Y_3 &= A(C^2X_1Z_2 - B) - C^3Y_1Z_2 & Z_3 &= C^3Z_1Z_2, \end{aligned}$$

where A, B, C are auxiliary variables and $\mathcal{P}_3 = (X_3, Y_3, Z_3)$ is the resulting point in projective coordinates. Similarly, for $\mathcal{P}_1 = \mathcal{P}_2$ the point doubling $\mathcal{P}_3 = 2\mathcal{P}_1$ is defined by

$$\begin{aligned} A &= aZ^2 + 3X^2 & B &= YZ \\ C &= XYB & D &= A^2 - 8C \\ X_3 &= 2BD & Y_3 &= A(4C - D) - 8B^2Y^2 \\ Z_3 &= 8B^3. \end{aligned}$$

Most ECC-based cryptosystems rely on the Elliptic Curve Discrete Logarithm Problem (ECDLP) and thus employ the technique of point multiplication $k \cdot \mathcal{P}$ as cryptographic primitive, i.e., a k times repeated point addition of a base point $\mathcal{P}$. More precisely, the ECDLP is the fundamental cryptographic problem used in protocols and crypto schemes like the Elliptic Curve Diffie-Hellman key exchange, the Elliptic Curve Integrated Encryption Scheme (ECIES) and the Elliptic Curve Digital Signature Algorithm (ECDSA) [19].

The arithmetic for ECC point multiplication is based on modular computations over a prime field $\mathbb{F}_p$. These computations always include a subsequent step to reduce the result to the domain of the underlying field. Since the reduction is very costly for general primes due to the demand for a multi-precision division, special primes have been proposed by Solinas [34] which have been finally standardized in [27]. These primes provide efficient reduction algorithms based on a sequence of multi-precision addition and subtractions only and eliminate the need for the costly division. Special primes P-l with bit sizes $l = \{192, 224, 256, 384, 521\}$ are part of the standard. But we expect that the primes P-224 and P-256 will become the most relevant bit sizes for practical implementations of the next decades. Algorithm 1 presents the modular reduction for P-256 requiring two doublings, four

256-bit subtractions and four 256-bit additions. Based on the computation $Z = z_1 + 2z_2 + 2z_3 + z_4 + z_5 - z_6 - z_7 - z_8 - z_9$, the interval of the possible result is $-4p < Z < 5p$.

Algorithm 1 NIST Reduction with P-256 $= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

```
Input: Double-sized integer $c = (c_{15}, \ldots, c_2, c_1, c_0)$ in base $2^{32}$ and $0 \leq c <$ P-256$^2$
Output: Single-sized integer $c \bmod$ P-256.
1: Concatenate $c_i$ to following 256-bit integers $z_j$:
       $z_1 = (c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,
       $z_2 = (c_{15}, c_{14}, c_{13}, c_{12}, c_{11}, 0, 0, 0)$,
       $z_3 = (0, c_{15}, c_{14}, c_{13}, c_{12}, 0, 0, 0)$,
       $z_4 = (c_{15}, c_{14}, 0, 0, 0, c_{10}, c_9, c_8)$,
       $z_5 = (c_8, c_{13}, c_{15}, c_{14}, c_{13}, c_{11}, c_{10}, c_9)$,
       $z_6 = (c_{10}, c_8, 0, 0, 0, c_{13}, c_{12}, c_{11})$,
       $z_7 = (c_{11}, c_9, 0, 0, c_{15}, c_{14}, c_{13}, c_{12})$,
       $z_8 = (c_{12}, 0, c_{10}, c_9, c_8, c_{15}, c_{14}, c_{13})$,
       $z_9 = (c_{13}, 0, c_{11}, c_{10}, c_9, 0, c_{15}, c_{14})$
2: Compute $c = (z_1 + 2z_2 + 2z_3 + z_4 + z_5 - z_6 - z_7 - z_8 - z_9 \bmod$ P-256)
```

5.2 Arithmetic Units

According to the EC arithmetic introduced above, an ECC engine over $\mathbb{F}_p$ based on projective coordinates requires functionality for modular addition, subtraction and multiplication. Since modular addition and subtraction is very similar, both operation are combined. In the following description we will assume a Virtex-4 FPGA as reference device and corresponding DSP block arithmetic with word sizes $l_A = 32$ and $l_M = 16$ for unsigned addition and multiplication, respectively. Recall from Section 3 that native support by the DSP blocks on a Virtex-4 device is available for up to 48-bit signed addition and 18-bit signed multiplication.

5.2.1 Modular Addition/Subtraction

Let $A, B \in GF(P)$ be two multi-precision operands with lengths $|A|, |B| \leq l$ and $l = \lfloor \log_2 P \rfloor + 1$. Modular addition $C = A + B \bmod P$ and subtraction $C = A - B \bmod P$ can be efficiently computed according to Algorithm 2:

Algorithm 2 Modular addition and subtraction

```
Input: $A, B, P$ with $0 \leq A, B < P$;
       Flag $f \in \{0, 1\}$ denotes a subtraction when $f = 1$ and addition otherwise
Output: $C = A \pm B \bmod P$
1: $(C_{\text{IN0}}, S_0) = A + (-1)^f B$;
2: $(C_{\text{IN1}}, S_1) = S_0 + (-1)^{1-f} P$;
3: Return $S_{|f - C_{\text{IN}f}|}$;
```

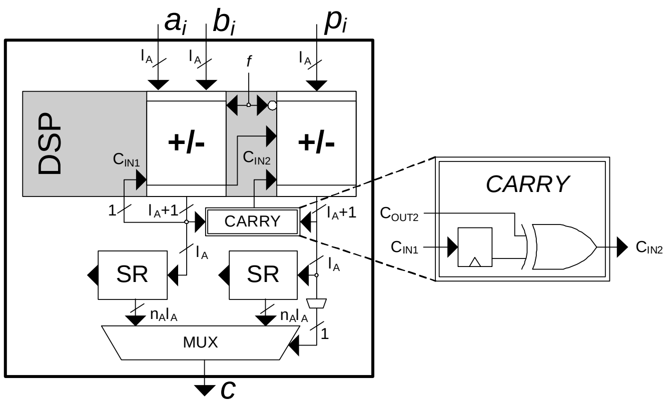


Fig. 5: Modular addition/subtraction based on DSP-blocks.

For using DSP blocks, we need to divide the l-bit operands into multiple words each having a maximum size of l_A bits due to the limited width of the DSP input port. Thus, all inputs A, B and P to the DSP blocks can be represented in the form $X = \sum_{i=0}^{n_A - 1} x_i \cdot 2^{i \cdot l_A}$, where $n_A = \lceil l/l_A \rceil$ denotes the number of words of an operand. According to Algorithm 2, we employ two cascaded DSP blocks, one for computing $s_{(0,i)} = a_i \pm (b_i + C_{\text{IN0}})$ and a second for $s_{(1,i)} = s_{(0,i)} \mp (p_i + C_{\text{IN1}})$. The resulting values $s_{(0,i)}$ and $s_{(1,i)}$ each of size $|s_{(j,i)}| \leq l_A + 1$ are temporarily stored and recombined to S_0 and S_1 using shift registers (SR). Finally, a 2-to-1 l-bit output multiplexer selects the appropriate value $C = S_i$. Figure 5 presents a schematic overview of a combined modular addition and subtraction based on two DSP blocks. Note that DSP blocks on Virtex-4 FPGAs provide a dedicated carry input c_{IN} but *no* carry output c_{OUT}. Particularly, this fact requires extra logic to compensate for duplicate carry propagation to the second DSP which is due to the fixed cascaded routing path between the DSP blocks. In this architecture, each carry is considered twice, namely in $s_{0,i+1}$ and $s_{1,i}$ what needs to be corrected. This special carry treatment requires a wait cycle to be introduced so that one l_A-bit word can be processed each two clock cycles. However, this is no restriction for our architecture since we design for *parallel* addition and multiplication so that the (shorter) runtime of an addition is completely hidden in the duration of a concurrent multiplication operation.

5.2.2 Modular Multiplication

The most straightforward multiplication algorithm to implement the multiplication with subsequent NIST prime reduction is the schoolbook multiplication method with a time complexity of $\mathcal{O}(n^2)$ for n-bit inputs. Other methods, like the Karatsuba algorithm [23], trade multiplications for additions using a divide-and-conquer approach. However Karatsuba computing the product $C = A \times B$ for $A = a_1 2^n + a_0$ and $B = b_1 2^n + b_0$ requires to store the intermediate results for $a_1 a_0$ and $b_1 b_2$ for later reuse in the algorithm. Although this is certainly possible, this requires a much more complex data and memory handling

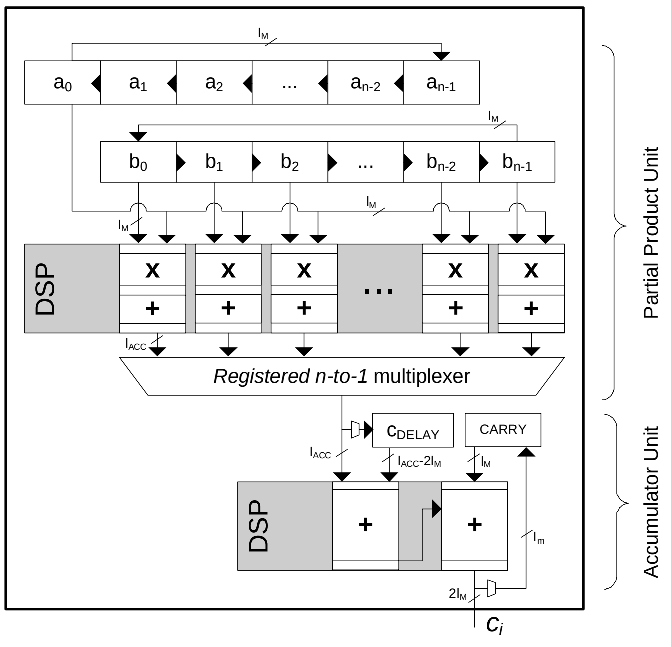


Fig. 6: An l-bit multiplication circuit employing a cascade of parallelly operating DSP blocks.

and cannot be solely done within DSP blocks. Since many parts of the Karatsuba multiplier would require generic logic of CLBs, we are likely to lose the gain of performance of the fast arithmetic in DSP blocks.

We thus use a variant of the schoolbook multiplication, known as Comba multiplication [7] which combines carry handling and reduces write accesses to the memory. These optimizations result in improved performance with respect to the original pen-and-paper method. Let $A, B \in GF(P)$ be two multi-precision integers with bit length $l \leq \lfloor \log_2 P \rfloor + 1$. According to the limited input size l_M of DSP blocks, we split now the values A, B in $n_M = \lceil l/l_M \rceil$ words represented as $X = \sum_{i=0}^{n_M-1} x_i \cdot 2^{il_M}$. Straightforward multiplication computes $C = A \cdot B$ based on accumulation of $(n_M)^2$ products $C = \sum_{i=0}^{2n_M} 2^{i \cdot n_M} \sum_{j=0}^{i} a_j b_{i-j}$ providing a result C of size $|C| \leq 2n_M$. For parallel execution on n_M DSP units, we compacted the order of inner product computations used for Comba's algorithm. All n_M DSP blocks operate in a loadable *Multiply-and-Accumulate* mode (MACC) so that intermediate results remain in the corresponding DSP block until an inner product $s_i = \sum_{j=0}^{i} a_j b_{i-j}$ is fully computed.

Figure 6 gives a schematic overview of the multiplication circuit returning the full-size product C. This result has to be reduced using the fast NIST prime reduction scheme discussed in the next section.

5.2.3 Modular Reduction

At this point we will discuss the subsequent modular reduction of the $2n_M$-bit multiplication result C using the NIST reduction scheme. All fast NIST reduction algorithms rely on a reduction step (1) defined as a series multi-precision

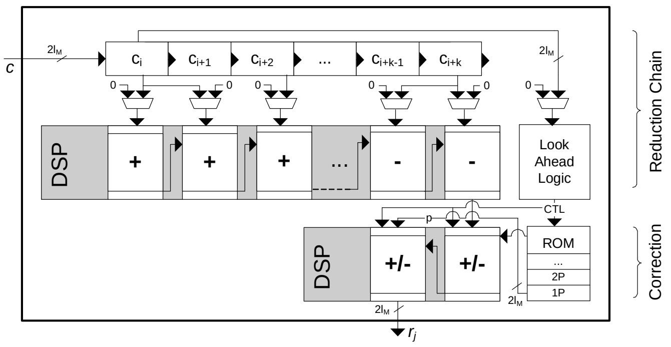


Fig. 7: Modular reduction for NIST-P-224 and P-256 using DSP blocks.

additions and subtractions followed by a correction step (2) to achieve a final value in the interval $[0, \dots, P-1]$ (cf. Algorithm 1). To implement (1), we decided to use one DSP-block for each individual addition or subtraction, e.g., for the P-256 reduction we reserved a cascade of 8 DSP blocks. Each DSP performs one addition or subtraction and stores the result in a register whose output is taken as input to the neighboring block (*pipeline*).

For the correction step (2), we need to determine *in advance* the possible overflow or underflow of the result returned by (1) to avoid wait or idle cycles in the pipeline. Hence, we introduced a Look-Ahead Logic (LAL) consisting of a separate DSP block which exclusively computes the expected overflow or underflow. Then, the output of the LAL is used to select a corresponding reduction value which are stored as multiple $\{0, \dots, 5P\}$ in a ROM table. The ROM values are added or subtracted to the result of (1) by a sequence of two DSP blocks ensuring that the final result is always in $\{0, \dots, P-1\}$. Figure 7 depicts the general structure of the reduction circuit which is applicable for both primes P-224 and P-256.

5.3 Core Architecture

With the basic field operations for $l - bit$ computations at hand supporting NIST primes P-224 and P-256, we have combined a modular multiplier and a modular subtraction/addition component with dual-port RAM modules (BRAM) and a state machine to build an ECC core. We have implemented an asymmetric data path supporting two different operand lengths: the first operand provides full l-bit of data whereas the second operand is limited to 32-bit words so that several words need to be transferred serially to generate the full l-bit input. This approach allows for direct memory accesses of our *serial-to-parallel* multiplier architecture. Note further that we introduced different clock domains for the core arithmetic based on the DSP blocks and the state machines for upper layers (running at half clock frequency only). An overview of the entire ECC core is shown in Figure 8. We implemented ECC group operations based on projective Chudnowsky coor-

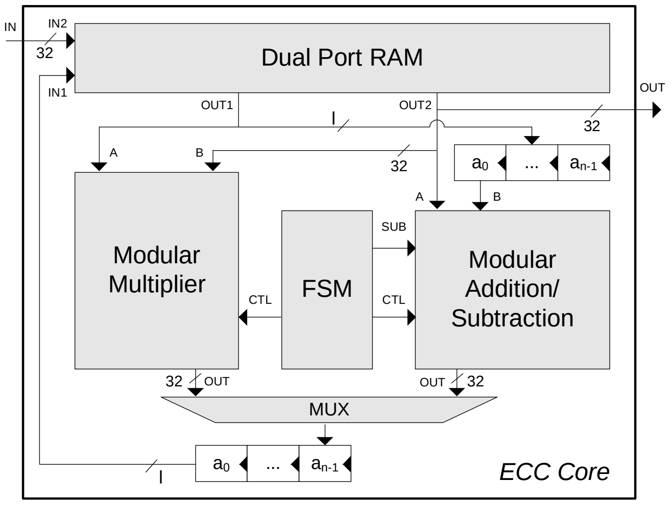


Fig. 8: Schematic overview of a single ECC core.

dinates[1] since the implementation should support to compute a point multiplication $k \cdot \mathcal{P}$ *as well as* a corresponding linear combination $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ based on a fixed base point $\mathcal{P} \in \mathcal{E}$ where $k, r \in \{1, \ldots, ord(\mathcal{P}) - 1\}$ and $\mathcal{Q} \in \langle \mathcal{P} \rangle$. Both operations can be considered as basic ECC primitives, e.g., used for ECDSA signature generation and verification [1]. The computation of $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ can make use of *Shamir-Strauss' trick* to efficiently compute several point products simultaneously [19]. For this first implementation of the point multiplication and the sake of simplicity, we used a standard double-and-add (binary method) algorithm [19], but more efficient windowing methods [2] can also be implemented without significantly increasing the resource consumption.

5.4 Core Parallelism

Due the intensive use of DSP blocks to implement the core functionality of ECC, the resulting implementation requires only few reconfigurable logic elements on the FPGA. This allows for efficient multiple-core implementations on a single FPGA improving the overall system throughput by a linear factor n dependent on the number of cores. Note that most other high-performance implementations occupy the full FPGA due to their immense resource consumption so that these cannot easily be instantiated several times.
Based on our synthesis results, the limiting factor of our architecture is the number of available DSP blocks of a specific FPGA device (cf. Section 5.5).

5.5 Implementation

The proposed architecture has been synthesized and implemented for the smallest available Xilinx Virtex-4 device (XC4VFX12-12SF363). This FPGA offers 5,472 slices (12,288 4-input LUTs and flip-flops) of reconfigurable logic, 32 DSP blocks and can be operated at a maximum clock frequency of 500 MHz. Furthermore, to demonstrate how many ECC computations can be performed using ECC core parallelism, we take a second device, the large Xilinx Virtex-4 XC4VSX55-12FF1148 providing the maximum number of 512 DSP blocks and 24,576 slices (49,152 4-input LUTs and flip-flops) as a reference for a multi-core architecture.

[1]ECC operations for elliptic curves in Weierstrass form based on mixed affine-Jacobian coordinates are more efficient but more complex in hardware. This is due to the required conversion of precomputed points from Jacobian to affine coordinates which is necessary when computing $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ for a ECDSA signature verification. In that case modular inversion is required.

Aspect	**Core**P-224	**Core** P-256
Slices occupied	1,580 (29%)	1,715 (31%)
4-input LUTs	1,825	2,589
Flip-flops	1,892	2,028
DSP blocks	26	32
BRAMs	11	11
Frequency (arithmetic)	486 MHz	490 MHz
Frequency (control)	243 MHz	245 MHz

Table 3: Resource requirements of a single ECC core on a Virtex-4 FX 12 after PAR. Note the different clock domains for arithmetic (DSP) and control logic.

5.5.1 Implementation Results

Based on the Post-Place and Route (PAR) results using Xilinx ISE 9.1 we can present the following performance and area details for ECC cores for primes P-224 and P-256 on the small XC4VFX12 device as shown in Table 3.

5.5.2 Throughput of a Single ECC Core

Given an ECC core with a separate adder/subtracter and multiplier unit, we can perform a field multiplication and field addition simultaneously. By optimizing the execution order of the basic field operations, it is possible to perform all additions/subtraction required for the ECC group operation in parallel to a multiplication. Based on the runtimes of a single field multiplication, we can determine the number of required clock cycles for the operations $k \cdot \mathcal{P}$ and $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ using the implemented Double-and-Add algorithm. Moreover, we also give estimates concerning their performance when using a window-based method [2] based on a window size $w = 4$.

Note that the specified timing considers signal propagation after complete PAR excluding the timing constraints from I/O pins ("TIG" constraint) since no underlying data communication layer was implemented. Hence, when being combined with an I/O protocol of an application, the clock frequency can be slightly lower than specified in Table 3 and Table 5.

5.5.3 Multi-Core Architecture

Since a single ECC core has obviously moderate resource requirements, it is possible to place multiple instances of the core on a larger FPGA. On a *single* XC4VSX55 device, we can implement, depending on the underlying prime field, between 16 to 18 ECC cores in parallel (cf. Table 5). Due the small amount of LUTs and flip-flops required for a single core, the number of available DSP blocks (and routing resources) on the FPGA is here the limiting factor.

Scheme	**Device**	**Design**	**Logic**	**Clock**	**Time**
This work	XC4VFX12-12	ECC NIST P-224	1580 LS/26 DSP	487 MHz	365 μs
	XC4VFX12-12	ECC NIST P-256	1715 LS/32 DSP	490 MHz	495 μs
	XC4VSX55-12	ECC NIST P-224	24452 LS/468 DSP	372 MHz	26.5 μs
	XC4VSX55-12	ECC NIST P-256	24574 LS/512 DSP	375 MHz	40.5 μs
[31]	XCV1000E	ECC NIST P-192	5708 LS	40 MHz	3 ms
[24]	XC2VP125-7	ECC P-256 all	15755 LS/256 MUL	39.5 MHz	3.84 ms
[18]	XC5VLX110	Mersenne 256-bit	1439 LS/56 DSP	50 MHz	188 μs
[13]	Intel Core2 Duo	ECC NIST P-256	64-bit μP	2.13 GHz	669[a] μs
[16]	Intel Core2 Duo	ECC GF($2^{255}-19$)	64-bit μP	2.66 GHz	145 μs
[3]	XC40250XV	RSA-1024	6826 CLB	45.2 MHz	3.1 ms
[36]	XC4VFX12-10	RSA-1024	3937 LS/17 DSP	400 MHz	1.71 ms

[a]Note that this figure reflects a full ECDSA signature generation rather than a point multiplication.

Table 6: Selected high-performance implementations of public-key cryptosystems.

Aspect	**Core** P-224	**Core** P-256
Cycles per MUL in $\mathbb{F}_p$	58	70
Cycles per ADD/SUB in $\mathbb{F}_p$	16	18
Cycles per ECC Addition	812	980
Cycles per ECC Doubling	580	700
Cycles $k\mathcal{P}$ (D&A)	219,878	303,450
Cycles $k\mathcal{P}$ (W)	178,000*	243,000*
Cycles $k\mathcal{P}+r\mathcal{Q}$ (D&A)	265,959	366,905
Cycles $k\mathcal{P}+r\mathcal{Q}$ (W)	194,000*	264,000*
Time&OP/s $k\mathcal{P}$ (D&A)	452 μs/2214	620 μs/1614
Time&OP/s $k\mathcal{P}$ (W)	365 μs*/2740*	495 μs*/2020*
Time&OP/s $k\mathcal{P}+r\mathcal{Q}$ (D&A)	546 μs/1831	749 μs/1335
Time&OP/s $k\mathcal{P}+r\mathcal{Q}$ (W)	398 μs*/2510*	540 μs*/1850*

Table 4: Performance of ECC operations based on a single ECC core using projective Chudnowsky coordinates on a Virtex-4 XC4VFX12. Note that the figures for Window method (W) denoted with an asterisk are estimates, results for the Double&Add (D&A) method were implemented on the device.

Aspect	**Core** P-224	**Core** P-256
Number of Cores	18	16
Slices occupied	24,452 (99%)	24,574 (99%)
4-input LUTs	32,688	34,896
Flip-flops	34,166	32,430
DSP blocks	468	512
BRAMs	198	176
Frequency (arithmetic)	372 MHz	374 MHz
Frequency (control)	186 MHz	187 MHz
OP/s kP (D&A)	30,438	19,760
OP/s kP (W)	37,700*	24,700*
OP/s $kP+rQ$ (D&A)	25,164	16,352
OP/s $kP+rQ$ (W)	34,500*	22,700*

Table 5: Results of a multi-core architecture on a Virtex-4 XC4VSX55 device for ECC over prime fields P-224 and P-256 (Figures with an asterisk are estimates).

5.6 Comparison

Based on our architecture, we can estimate a throughput of more than 37,000 point multiplications on the standardized elliptic curve P-224 per second. A detailed comparison with other implementations is presented in Table 6.

At this point we like to point out that the field of highly efficient *prime field* arithmetic is believed to be predominated by implementations on general purpose microprocessors rather than on FPGAs. Hence, we will also compare our hardware implementation against the performance of software solutions on recent microprocessors. Since most performance figures for software implementations are given in cycles rather than absolute times, we assumed for comparing throughputs that uninterrupted, repeated computations can be performed simultaneously on *all* available cores of a modern microprocessor with no further cycles spent, e.g., on scheduling or other administrative tasks. Note that this is indeed a very optimistic assumption possibly overrating the performance of software implementations with respect to actual applications.

We compare our design to recent software results, e.g., obtained from ECRYPT's eBATS project. According to [13], an Intel *Core2 Duo* running at 2.13 GHz is able to generate 1868 and 1494 ECDSA signatures based on the OpenSSL implementation for P-224 and P-256, respectively. Taking latest *Core i* microprocessors with four or six cores into account, these performance figures might even double or triple. We also compare our work to the very fast software implementation by [16] using an Intel Core2 Duo system at 2.66 GHz. However, in this contribution the special Montgomery and non-standard curve over $\mathbb{F}_{2^{255}-19}$ is used instead of a standardized NIST prime. Despite of that, for the design based on this curve the authors report the impressive throughput of 6700 point multiplications per second.

For a fair comparison with software solutions it should be considered that a single Virtex-4 SX 55 costs about US$ 1,123.[2] Recent microprocessors like the Intel Core2 Duo, however, are available at less than a quarter of that price. With this in mind, we are not able to beat all software implementation in terms of the cost-performance ratio, but we still like to point out

[2]Market price for a single device in March 2011.

that our FPGA-based design - as the fastest reported hardware implementation so far - definitely closes the performance gap between software and hardware implementations for ECC over prime fields.

6. Conclusions

In this work, we presented alternative design strategies for standardized symmetric and asymmetric cryptosystems that involve the hard cores available on modern FPGA devices. According to our results, the AES and ECC implementations are among the ones providing the highest performance and throughput reported in the open literature. Note that the source code for our AES implementations is freely available so that our results can also be externally verified [11].

Future work encompasses the thorough analysis and inclusion of additional countermeasures against side-channel attacks (SCA). Note that the implementations already provide a basic protection against SCA by design due to the very parallel data processing operating at high clock frequencies.[3] However, although these properties of our design will have a severe impact on the difficulty of a successful SCA, those attacks are still not completely impossible what need to be assessed in future work.

References

[1] ANSI X9.62-2005. American National Standard X9.62: The Elliptic Curve Digital Signature Algorithm (ECDSA), 2005.

[2] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2005.

[3] T. Blum and C. Paar. High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware. *IEEE Transactions on Computers*, 50(7):759–764, 2001.

[4] P. Bulens, F. Standaert, J.-J. Quisquater, P. Pellegrin, and G. Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In S. Vaudenay, editor, *AFRICACRYPT 2008*, volume 5023 of *LNCS Series*, pages 16–26, 2008.

[5] R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa. Reconfigurable Memory-based AES Co-Processor. In *Workshop on Reconfigurable Architectures*, page 192, 2006.

[6] P. Chodowiec and K. Gaj. Very Compact FPGA Implementation of the AES Algorithm. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *CHES*, volume 2779 of *LNCS*, pages 319–333, 2003.

[7] P. G. Comba. Exponentiation Cryptosystems on the IBM PC. *IBM Systems Journal*, Vol. 29(4):526–538, 1990.

[8] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[9] A. Daly, W. Marnane, T. Kerins, and E. Popovici. An FPGA Implementation of a GF(p) ALU for Encryption Processors. *Elsevier - Microprocessors and Microsystems*, 28(5–6):253–260, 2004.

[10] G. M. de Dormale and J.-J. Quisquater. High-Speed Hardware Implementations of Elliptic Curve Cryptography: A Survey. *Journal of Systems Architecture*, 53(2-3):72–84, 2007.

[11] S. Drimer, T. Güneysu, and C. Paar. DSPs, BRAMs and a Pinch of Logic: New Recipes for AES on FPGAs. In *IEEE Symposium FCCM 2000*, pages 99–108. IEEE Computer Society, April 2008. Source code available at: `http://www.cl.cam.ac.uk/~sd410/aes/`.

[12] H. Eberle, N. Gura, and S. Chang-Shantz. A Cryptographic Processor for Arbitrary Elliptic Curves over GF(2^m). In *Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 444–454, 2003.

[13] ECRYPT. *eBATS: ECRYPT Benchmarking of Asymmetric Systems*, March 2007. Available at `http://www.ecrypt.eu.org/ebats/`.

[14] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, 9(4):545–557, 2001.

[15] V. Fischer and M. Drutarovský. Two Methods of Rijndael Implementation in Reconfigurable Hardware. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *CHES*, volume 2162 of *LNCS*, pages 77–92, 2001.

[16] P. Gaudry and E. Thomé. The mp$\mathbb{F}q$ Library and implementing Curve-based Key Exchanges. *Workshop on Software Performance Enhancement for Encryption and Decryption (SPEED 2007)*, 2007.

[17] T. Güneysu and C. Paar. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In E. Oswald and P. Rohatgi, editors, *CHES*, volume 5154 of *LNCS*, pages 62–78, 2008.

[18] M. Hamilton and W. P. Marnane. FPGA Implementation of an Elliptic Curve Processor Using the GLV Method. In *Reconfigurable Computing and FPGAs (ReConFig)*, pages 249–254, 2009.

[19] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, New York, 2004.

[20] T. Ichikawa, T. Kasuya, and M. Matsui. Hardware Evaluation of the AES Finalists. *AES Candidate Conference*, pages 13–14, 2000.

[21] K. U. Järvinen. *Studies on High-Speed Hardware Implementations of Cryptographic Algorithms*. PhD thesis, Helsinki University of Technology, 2008.

[22] K. U. Järvinen, M. T. Tommiska, and J. O. Skyttä. A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA 2003)*, pages 207–215, New York, NY, USA, 2003. ACM Press.

[23] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics – Doklady*, 7(7):595–596, 1963.

[24] C. McIvor, M. McLoone, and J. McCanny. An FPGA Elliptic Curve Cryptographic Accelerator over GF(p). In *Irish Signals and Systems Conference (ISSC)*, pages 589–594, 2004.

[25] M. McLoone and J. McCanny. High Performance Single-chip FPGA Rijndael Algorithm Implementations. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *CHES*, volume 2162 of *LNCS*, pages 65–76, 2001.

[26] M. McLoone and J. McCanny. Rijndael FPGA Implementations Utilising Look-up Tables. *The Journal of VLSI Signal Processing*, 34(3):261–275, 2003.

[27] National Institute of Standards and Technology (NIST). Recommended Elliptic Curves for Federal Government Use, July 1999.

[28] National Institute of Standards and Technology (NIST). FIPS PUB 197: Advanced Encryption Standard, 2001.

[29] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS) (FIPS 186-3), June 2009.

[30] G. Orlando and C. Paar. A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m). In Ç. K. Koç and C. Paar, editors, *CHES*, volume 1965 of *LNCS*, pages 41–56, 2000.

[31] G. Orlando and C. Paar. A Scalable $GF(p)$ Elliptic Curve Processor Architecture for Programmable Hardware. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *CHES*, volume 2162 of *LNCS*, pages 356–371, 2001.

[32] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat. Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications. *International Conference on Information Technology*, 2:583, 2004.

[33] A. Satoh and K. Takano. A Scalable Dual-Field Elliptic Curve Cryptographic Processor. *IEEE Transactions on Computers*, 52(4):449–460, 2003.

[34] J. A. Solinas. Generalized Mersenne Numbers. Technical report, National Security Agency (NSA), Sept. 1999.

[35] F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat. Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *CHES*, volume 2779 of *LNCS*, pages 334–350, 2003.

[36] D. Suzuki. How to Maximize the Potential of FPGA Resources for Modular Exponentiation. In P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *LNCS*, pages 272–288, 2007.

[37] Xilinx Inc. *UG190: Virtex-5 User Guide*, 2006. Available at `http://www.xilinx.com/support/documentation/user_guides/ug190.pdf`.

[3]We here assume that driving the implementation at much lower clock frequencies is not easily possible due to internal restrictions of the Digital Clock Manager inside the FPGA.

Elliptic Curve Cryptography on FPGAs: How Fast Can We Go with a Single Chip?

Kimmo Järvinen
Aalto University, School of Science
Department of Information and Computer Science
P.O. Box 15400, FI-00076 Aalto, Finland
kimmo.jarvinen@aalto.fi

Abstract—*In this paper we present an extremely high throughput implementation of an elliptic curve cryptosystem. The work builds on the author's previous work which has resulted in a high throughput processor architecture for a specific family of elliptic curves called Koblitz curves. The architecture extensively utilizes the fact that FPGA based designs can be carefully optimized for fixed parameters (e.g., for a single elliptic curve) because parameter flexibility (e.g., support for different curves) can be achieved through reprogrammability. In this paper the work is extended by exploring optimal solutions in a situation where several parallel instances of the processor architecture are placed on a single chip. It is shown that by utilizing the suggested parallel architecture a single modern FPGA chip, Stratix IV GX EP4SGX230KF40C2, can deliver a throughput of about 1,700,000 public-key cryptography operations (scalar multiplications on a secure elliptic curve) per second. This far exceeds any values thus far reported in the literature.*

Keywords: Elliptic curve cryptography, field-programmable gate array, parallel implementation, Koblitz curve

1. Introduction

Neal Koblitz and Victor Miller independently proposed the use of elliptic curves for public-key cryptography in 1985 [1], [2]. Since then, elliptic curve cryptography (ECC) has been intensively studied because it offers both shorter keys [3] and faster performance [4], [5] compared to more traditional public-key cryptosystems, such as RSA [6]. Hardware implementation of ECC has also gained considerable interest and, as a consequence, many descriptions of hardware implementations exist in the literature; see [7] for a comprehensive review.

Field programmable gate arrays (FPGAs) have proven to be highly feasible platforms for implementing cryptographic algorithms because of the combination of programmability and high speed. Several advantages of FPGAs in cryptographic applications were listed in [8]. One of the advantages, "Architecture efficiency," follows from the fact that, in an FPGA-based design, optimizations for specific parameters can be done without major restrictions in the generality of the system because, if other parameters are needed, the FPGA can be reprogrammed to support the new parameters [8].

"Architecture efficiency" has been exploited in numerous papers describing FPGA-based implementations of ECC [7]. However, a vast majority of papers optimize only field arithmetic units for one specific field while the higher levels of ECC still remain unoptimized and use more generic architectures. This approach seems rather pointless because fixing the underlying field already restricts the number of usable elliptic curves to very few. For instance, fixing the field to $\mathbb{F}_{2^{163}}$ means that from the total of fifteen curves recommended by U.S. National Institute of Standards and Technology (NIST) in [9] only two curves, namely B-163 and K-163, could be used. Hence, if the field is fixed in order to increase performance, one should optimize the architecture also on higher levels for a specific curve. In this paper, we describe an FPGA-based processor that is optimized specifically for Koblitz curves [10].

1.1 Related work

The first FPGA-based implementation using Koblitz curves was presented in [11], where one scalar multiplication was shown to require 45.6 ms on the NIST K-163 curve with an Altera Flex 10K FPGA. They concluded that Koblitz curves are approximately twice as fast as general curves. [12] presented an implementation which computes scalar multiplication in 75 μs on NIST K-163 in a Xilinx Virtex-E FPGA. Neither of the two designs includes a circuitry for conversions that are mandatory for Koblitz curves (see Sec. 2.3). [13], [14] proposed a multiple-base expansion which can be used for increasing the speed of Koblitz curve computations and presented FPGA implementations for both elliptic curve scalar multiplication and conversion. Scalar multiplication was shown to require 35.75 μs on NIST K-163 with a Xilinx Virtex-II whereas the conversion requires 3.81 μs in [13]. [14] presented a parallelized version of the processor of [13] achieving computation delay of 17.15 μs on Stratix II including the conversion. [15] presented a high-speed processor using parallel formulations of scalar multiplication on Koblitz curves. Their processor achieves a very fast computation delay of 7.22 μs on NIST K-233 with Virtex-II, but it also neglects the conversions.

Our recent work considering scalar multiplication on Koblitz curves in FPGAs consists of [16], [17], [18], [19], [20]. It was shown in [16] that up to 166,000 signature verifications can be computed using a single Stratix II FPGA with parallel processing. More general parallelization studies were presented in [17] and they resulted in an implementation that computes scalar multiplication in 25.81 μs. We showed that even shorter computation delay of only 4.91 μs (without the conversion) can be achieved on NIST K-163 with interleaved operations [18]. A complete FPGA-based processor architecture utilizing this method was described in [19]. This was architecture was improved by using more efficient algorithms and by redesigning the processor architecture in [20].

1.2 Contributions of the paper

In this article, we explore parallel implementations of the processor architecture presented in [20]. The processor of [20] was optimized to deliver the maximum throughput. This optimization strategy is not necessarily optimal if we have several parallel instances of the processor because a higher overall throughput might be achievable with a larger number of slower but smaller processors. Hence, the target in this work is to maximize the throughput-area ratio of the processor. We show that a setup similar to the one used in [20] is the optimal one also in this case.

It was conjectured in [20] that a parallel implementation on a modern FPGA could achieve throughputs of over 1,000,000 scalar multiplications per second. In this paper, we show that this conjecture was correct. We demonstrate that a single Altera Stratix IV chip is capable of delivering a throughput of 1,700,000 scalar multiplications per second on a standardized elliptic curve NIST K-163. Such an extremely fast accelerator could have applications, for instance, in very heavily loaded network servers or in cryptanalytic hardware.

1.3 Structure of the paper

The remaining of the paper is structured as follows. Sec. 2 presents the preliminaries of finite fields, elliptic curves, and Koblitz curves. Sec. 3 introduces algorithms that are used in the proposed implementation. The processor architecture from [20] that we use as the base architecture is described in Sec. 4. Sec. 5 discusses parallel implementations of the processor architecture described in Sec. 4 and finds out the parameters that provide the maximum throughput. The results on an Altera Stratix IV GX FPGA are presented in Sec. 6. Finally, we conclude the paper in Sec. 7.

2. Preliminaries

2.1 Finite fields

Elliptic curves defined over finite fields $\mathbb{F}_q$ are used in cryptography and only curves over binary fields, where $q = 2^m$, with polynomial basis are considered in this paper. Polynomial bases are commonly used in elliptic curve cryptosystems because they provide fast performance on both software and hardware. Another commonly used basis, normal basis, provides very efficient squaring but multiplication is more complicated.

Elements of $\mathbb{F}_{2^m}$ with polynomial basis are represented as binary polynomials with degrees less than m as $a(x) = \sum_{i=0}^{m-1} a_i x^i$. Arithmetic operations in $\mathbb{F}_{2^m}$ are computed modulo an irreducible polynomial[1] with a degree m. Because sparse polynomials offer considerable computational advantages, trinomials (three nonzero terms) or pentanomials (five nonzero terms) are used in practice. The curve, NIST K-163, considered in this paper is defined over $\mathbb{F}_{2^{163}}$ with the pentanomial $p(x) = x^{163} + x^7 + x^6 + x^3 + 1$ [9].

Addition, $a(x) + b(x)$, in $\mathbb{F}_{2^m}$ is a bitwise exclusive-or (XOR). Multiplication, $a(x)b(x)$, is more involved and it consists of two steps: ordinary multiplication of polynomials and reduction modulo $p(x)$. If both multiplicands are the same, the operation is called squaring, $a^2(x)$. Squaring is cheaper than multiplication because the multiplication of polynomials is performed simply by adding zeros to the bit vector. Reduction modulo $p(x)$ can be performed with a small number of XORs if $p(x)$ is sparse and fixed, i.e. the same $p(x)$ is always used, which is the case in this paper. Repeated squaring denotes several successive squarings, i.e., exponentiation $a^{2^e}(x)$. Inversion, $a^{-1}(x)$, is an operation which finds $b(x)$ such that $a(x)b(x) = 1$ for a given $a(x)$. Inversion is the most complex operation and it can be computed either with the Extended Euclidean Algorithm or Fermat's Little Theorem (e.g., as suggested in [21]) that gives $a^{-1}(x) = a^{2^m-2}(x)$.

Multiplication has the most crucial effect on performance of an elliptic curve cryptosystem. A digit-serial multiplier computes D bits of the output in one cycle resulting in a total latency of $\lceil m/D \rceil$ cycles. We use hardware modifications of the multiplier described in [22]. Instead of using pre-computed look-up tables as in [22], our multiplier computes everything on-the-fly similarly as in [12]. Repeated squarings can be computed efficiently with the repeated squarers presented in [23] which are components that compute $a^{2^e}(x)$ directly in one clock cycle.

2.2 Scalar multiplication

Let E be an elliptic curve defined over a finite field $\mathbb{F}_q$. Points on E form an additive Abelian group, $E(\mathbb{F}_q)$, together with a point called the point at infinity, $\mathcal{O}$, acting as the zero element. The group operation is called point addition. Let P_1 and P_2 be two points in $E(\mathbb{F}_q)$. Point addition $P_1 + P_2$ where $P_1 = P_2$ is called point doubling. In order to avoid

[1] A polynomial, $f(x) \in \mathbb{F}[x]$, with a positive degree is irreducible over $\mathbb{F}$ if it cannot be presented as a product of two polynomials in $\mathbb{F}[x]$ with positive degrees.

confusion, point addition henceforth refers solely to the case $P_1 \neq \pm P_2$.

The principal operation of elliptic curve cryptosystems is scalar multiplication kP where k is an integer and $P \in E(\mathbb{F}_q)$ is called the base point. The most straightforward practical algorithm for scalar multiplication is the double-and-add algorithm (binary algorithm) where k is represented as a binary expansion $\sum_{i=0}^{\ell-1} k_i 2^i$ with $k_i \in \{0,1\}$. Each bit in the representation results in a point doubling and an additional point addition is computed if $k_i = 1$. Let w denote the Hamming weight of k, i.e., the number of nonzeros in the expansion. Depending on whether the algorithm starts from the most significant ($k_{\ell-1}$) or the least significant (k_0) bit of the expansion, the algorithm is called either left-to-right or right-to-left double-and-add algorithm. They are shown in Alg. 1 and 2, respectively. They both have the same costs: $\ell - 1$ point doublings and $w - 1$ point additions (the first operations are simply substitutions).

```
Input: Integer $k = \sum_{i=0}^{\ell-1} k_i 2^i$, point $P$
Output: Point $Q = kP$
$Q \leftarrow \mathcal{O}$
for $i = \ell - 1$ to $0$ do
    $Q \leftarrow 2Q$
    if $k_i = 1$ then $Q \leftarrow Q + P$
```

Algorithm 1: Left-to-right scalar multiplication

```
Input: Integer $k = \sum_{i=0}^{\ell-1} k_i 2^i$, point $P$
Output: Point $Q = kP$
$Q \leftarrow \mathcal{O}$
for $i = 0$ to $\ell - 1$ do
    if $k_i = 1$ then $Q \leftarrow Q + P$
    $P \leftarrow 2P$
```

Algorithm 2: Right-to-left scalar multiplication

If points on E are represented traditionally with two coordinates as (x, y), referred to as the affine coordinates, or $\mathcal{A}$ for short, both point addition and point doubling require one inversion in $\mathbb{F}_{2^m}$. Inversions are expensive as discussed in Sec. 2.1. Hence, it is often beneficial to represent points using projective coordinates, (X, Y, Z), where point additions and point doublings can be computed without inversions, but have an expense of a larger number of other operations (multiplications, squarings, and additions). In this paper, we use López-Dahab coordinates [24], or $\mathcal{LD}$ for short, where the point (X, Y, Z) represents the point $(X/Z, Y/Z^2)$ in $\mathcal{A}$. The $\mathcal{LD}$ coordinates offer, in particular, very efficient point additions, $P_1 + P_2$, if P_1 is in $\mathcal{LD}$ and P_2 is in $\mathcal{A}$. We extensively utilize these so called point additions with mixed coordinates [25] in our implementation.

2.3 Koblitz curves

Koblitz curves [10] are a family of elliptic curves defined over $\mathbb{F}_{2^m}$ by the following equation:

$$E_K : y^2 + xy = x^3 + ax^2 + 1 \tag{1}$$

where $a \in \{0, 1\}$. Koblitz curves are appealing because they offer considerable computational advantages over general curves. These advantages are based on the fact that an algorithm similar to the double-and-add can be devised so that point doublings are replaced by Frobenius endomorphisms. The Frobenius endomorphism, ϕ, for a point $P = (x, y)$ is a map such that

$$\phi : (x, y) \mapsto (x^2, y^2) \quad \text{and} \quad \mathcal{O} \mapsto \mathcal{O}. \tag{2}$$

Obviously, Frobenius endomorphism is very cheap: only two or three squarings depending on the coordinate system. Several successive Frobenius maps, i.e. $\phi^e(P)$, can be computed with two (or three) repeated squarings: x^{2^e} and y^{2^e}.

Replacing point doublings with Frobenius endomorphisms requires manipulations on k. It stands for all points in $E_K(\mathbb{F}_{2^m})$ that $\mu\phi(P) - \phi^2(P) = 2P$ where $\mu = (-1)^{1-a}$. Thus, ϕ can be seen as a complex number, τ, satisfying $\mu\tau - \tau^2 = 2$ which gives $\tau = (\mu + \sqrt{-7})/2$. Moving from a bit to another in a representation of k corresponds to an application of ϕ if k is given in a τ-adic representation:

$$k = \sum_{i=0}^{\ell-1} \epsilon_i \tau^i. \tag{3}$$

Hence in order to utilize fast Frobenius endomorphisms, k must be given in a τ-adic representation [10].

Efficient conversion algorithms were presented by Solinas in [26]. The basic algorithm returns the so-called τ-adic non-adjacent form (τNAF) where k is represented with the signed-binary format, i.e., $\epsilon_i \in \{0, \pm 1\}$. Henceforth, we denote $\bar{1} = -1$. The average length of τNAF is the same as the binary length of k, i.e., ℓ. τNAF has $w \approx \ell/3$ and one of two adjacent digits is always zero. Because $\ell \approx m$, scalar multiplication with k in τNAF requires on average $m/3 - 1$ point additions or subtractions and $m - 1$ applications of ϕ.

2.4 Width-ω τNAF

If enough storage space is available, point multiplication can be sped up with window methods which involve precomputations with P. We consider window methods only on Koblitz curves in order to keep the discussion focused, although analogous algorithms exist also for general curves. A left-to-right scalar multiplication algorithm with precomputations on Koblitz curves is shown in Alg. 3; see Sec. 3.1 for details about the scalar encoding.

Solinas presented an algorithm for producing width-ω τNAF in [26]. Instead of using that algorithm, we use the τNAF algorithm which is simpler to implement in hardware and interpret its results as a width-ω τNAF by replacing

```
Input: Integer $k = \sum_{i=0}^{\ell-1} \epsilon_i \tau^i$, point $P$
Output: Point $Q = kP$
$(k_0, f_0), (k_1, f_1), \dots, (k_{w-1}, f_{w-1}) \leftarrow$ $\mathrm{ConvertAndEncode}(k)$
$(P_1, P_2, \dots, P_N) \leftarrow \mathrm{Precompute}(P)$
$Q \leftarrow \mathcal{O}$
for $i = w - 1$ to $0$ do
    $Q \leftarrow Q + \mathrm{sign}(k_i) P_{|k_i|}$
    $Q \leftarrow \phi^{f_i}(Q)$
```

Algorithm 3: Left-to-right scalar multiplication on Koblitz curves with precomputations

```
Input: Integer $k = \sum_{i=0}^{\ell-1} \epsilon_i \tau^i$, point $P$
Output: Point $Q = kP$
$(k_0, f_0), (k_1, f_1), \dots, (k_{w-1}, f_{w-1}) \leftarrow$ $\mathrm{ConvertAndEncode}(k)$
$(P_1, P_2, \dots, P_N) \leftarrow \mathrm{Precompute}(P)$
$Q \leftarrow \mathcal{O}$
for $i = 0$ to $w - 1$ do
    for $j = 1$ to $N$ do
        $P_j \leftarrow \phi^{f_i}(P_j)$
    $Q \leftarrow Q + \mathrm{sign}(k_i) P_{|k_i|}$
```

Algorithm 4: Right-to-left scalar multiplication on Koblitz curves with precomputations

certain strings of 0, 1, and $\bar{1}$'s with window values. The resulting representation has a weight of $w \approx \ell/(\omega + 1)$.

3. Algorithms for the processor

3.1 Encoding for τ-adic expansions

In the implementation presented in this paper, we encode k as follows: $(k_0, f_0), (k_1, f_1), \dots, (k_{w-1}, f_{w-1})$, where $k_i \neq 0$ are the nonzero coefficients from (3) and f_i are the number of Frobenius maps between point additions (i.e., the number of zeros plus one). Similar encodings have been used prior to this work at least in [13], [14], [15], [16], [17], [19], [20]. For example, the expansion $\langle 10000\bar{3}0000\bar{1}00000050 \rangle$ results in $(1, 5)$, $(\bar{3}, 5)$, $(\bar{1}, 7)$, $(5, 1)$ for a left-to-right algorithm and $(5, 1)$, $(\bar{1}, 7)$, $(\bar{3}, 5)$, $(1, 5)$ for a right-to-left algorithm; i.e., they are mirror-images.

3.2 Right-to-left algorithms

As shown in Alg. 2, when scalar multiplication is computed from right to left, the base point P is doubled (Frobenius mapped) instead of the point Q. Point additions and point doublings (Frobenius maps) can be processed in parallel making these algorithms feasible for implementations supporting parallel processing of point operations. However, a right-to-left algorithm is, in general, unpractical if it involves precomputations with the base point P because all precomputed points need to be doubled in each iteration.

We can overcome this disadvantage by utilizing the inexpensiveness of Frobenius maps. Because Frobenius maps are cheap, we can map all precomputed points simultaneously while processing a point addition. We attach a repeated squarer directly to the register bank containing precomputed points. One repeated squarer can compute $\phi^{f_i}(x, y)$ in two clock cycles if $f_i \leq e_{\max}$ where $e_{\max}$ is the predefined maximum exponent [23]. Hence, N points require $2N$ clock cycles. If the latency of one point addition is longer than $2N$, then Frobenius maps reduce from the critical path. The right-to-left scalar multiplication algorithm on Koblitz curves with precomputations is shown in Alg. 4.

4. Description of the processor

In this section we review the processor architecture originally presented in [20]. The processor implements Alg. 4. It consists of four main components as shown in the toplevel view of the processor given in Fig. 1, and each of them is discussed in detail in Secs. 4.1–4.4.

The processor operates as follows. The converter (see Sec. 4.1) converts the integer k into width-4 τNAF and encodes it as described in Sec. 3.1. The precomputations are performed in the preprocessor (see Sec. 4.2) simultaneously with the conversion. When both of these computations are ready, the main for-loop of Alg. 4 is executed in the main processor (see Sec. 4.3). Finally, the postprocessor (see Sec. 4.4) maps the result point Q from $\mathcal{LD}$ to $\mathcal{A}$ and returns $Q = (x, y)$.

The processor comprises a three-stage pipeline and is capable of processing three scalar multiplications simultaneously. The converter and the preprocessor form the first stage, the main processor is the second stage and, finally, the postprocessor is the last stage, as shown in Fig. 1.

4.1 τNAF converter

As noted in Sec. 2.3, k must be given in a τ-adic representation when using Koblitz curves. We use the τNAF converter presented in [27] for converting the integer k into width-2 τNAF. After this, the width-2 τNAF is converted into width-4 τNAF by using a simple string replacement circuitry.

4.2 Preprocessor

The preprocessor computes the precomputed points, $P_1, \dots, P_N$, required by Alg. 4. In this paper, $N = 5$: We need 4 precomputed points with width-4 τNAF and one extra point in order to ensure that Frobenius maps do not appear on the critical path (i.e., $f_i \leq e_{\max}$; see Sec. 3.2 and [20] for more details). Because these precomputations do not depend on k, the τNAF conversion and the precomputations can be performed in parallel. The preprocessor is implemented using the architecture described in [17].

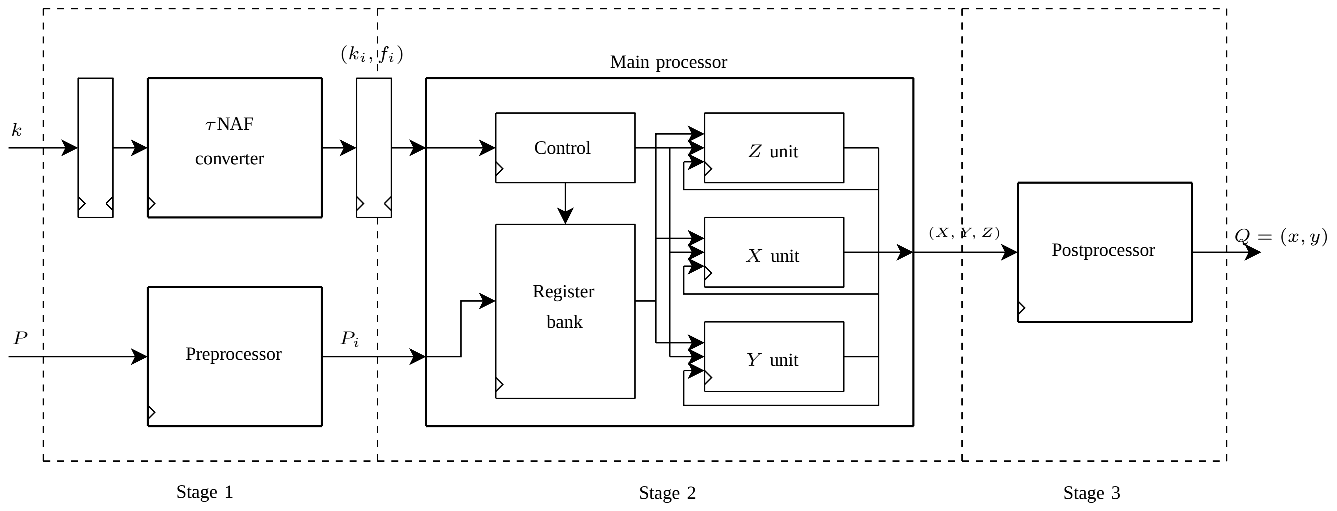


Fig. 1: Toplevel view of the processor

4.3 Main processor

The main processor computes the for-loop of the scalar multiplication by computing point additions and Frobenius maps. The way how this task is performed is based on an idea presented in [18] (and used also in [19], [20]). An adaptation of this idea to Alg. 4, the right-to-left algorithm with precomputations, was introduced in [20]. The advantage of this adaptation compared to [18], [19] is that the Frobenius maps are computed in parallel with the point additions, as discussed in Sec. 3.2. As a consequence, performance is bounded only by the latency of multiplication in $\mathbb{F}_{2^m}$.

First, we shall recap how point additions are computed in [18]. The main processor computes point addition with mixed coordinates, $(X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (x_2, y_2)$, using the formulae proposed by Al-Daoud et al. [25]:

$$\begin{aligned} A = Y_1 + y_2 Z_1^2; \quad B = X_1 + x_2 Z_1; \\ C = BZ_1; \quad Z_3 = C^2; \quad D = x_2 Z_3; \\ X_3 = A^2 + C(A + B^2 + aC); \\ Y_3 = (D + X_3)(AC + Z_3) + (x_2 + y_2)Z_3^2; \end{aligned} \tag{4}$$

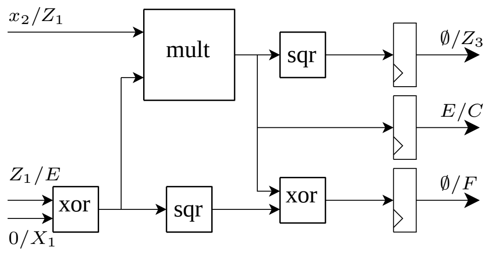
Clearly, there are eight multiplications in the above formulae (aC does not require multiplication because $a \in \{0, 1\}$), and they have a critical path of four multiplications with three or more multipliers. The main observation of [18] was that despite this critical path, it is possible to reduce the effective critical path to only two multiplications per point addition by interleaving the computation of successive point additions and Frobenius maps if one uses four multipliers.

As shown in (4), computing Z_3 requires two multiplications (in B and C computations). Computing X_3 requires two additional multiplications and Y_3 requires the remaining four. The second multiplication of X_3 cannot be started before C is available, i.e., one must wait for the result of the second multiplication of Z_3. All multiplications of Y_3 require that both multiplications of Z_3 are ready. The computation can be interleaved so that one computes the multiplications of Z_3 while simultaneously still processing the Y_3 coordinate of the previous point addition. The X_3 computation is started when the first multiplication of Z_3 is ready.

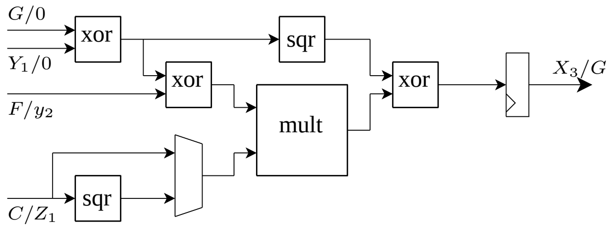
The main processor includes separate processing units for computing X, Y, and Z coordinates, henceforth referred to as the X, Y, and Z units. The X and Z units both contain one multiplier whereas the Y unit has two multipliers. These units are depicted in Fig. 2. The figure also shows how to set their inputs and outputs in order to compute (4); i.e., all units must be applied only twice to compute a point addition. Contrary to the main processor of [18], [19], the units do not have squarers computing Frobenius maps. The Frobenius maps are computed with a single repeated squarer [23] attached to the register bank containing the precomputed points. Fig. 3 shows the register bank.

The following procedure shows how (almost all) Frobenius maps can be removed from the critical path if one uses a right-to-left scalar multiplication algorithm:

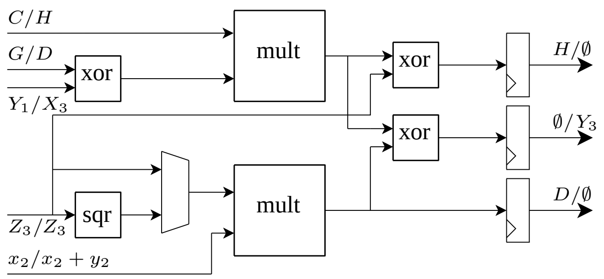
1) All precomputed points, $P_1, \ldots, P_N$, are stored in the register bank;
2) One computes and stores $\phi^{f_0}(P_1), \ldots, \phi^{f_0}(P_N)$ with one repeated squarer attached to the register bank which takes $2N$ clock cycles;
3) Immediately after this, one initializes the scalar multiplication by setting $Q = \text{sign}(k_0)P_{|k_0|}$; that is, $X = x_{|k_0|}$, $Z = 1$, and $Y = y_{|k_0|}$ if $\text{sign}(k_0) = 1$ and $Y = x_{|k_0|} + y_{|k_0|}$ if $\text{sign}(k_1) = -1$;
4) One computes and stores $\phi^{f_1}(P_1), \ldots, \phi^{f_1}(P_N)$ and when they are ready, begins computing the point addition $Q + \text{sign}(k_1)P_{|k_1|}$;
5) While the multipliers compute the above



(a)



(b)



(c)

Fig. 2: (a) Z, (b) X, and (c) Y units

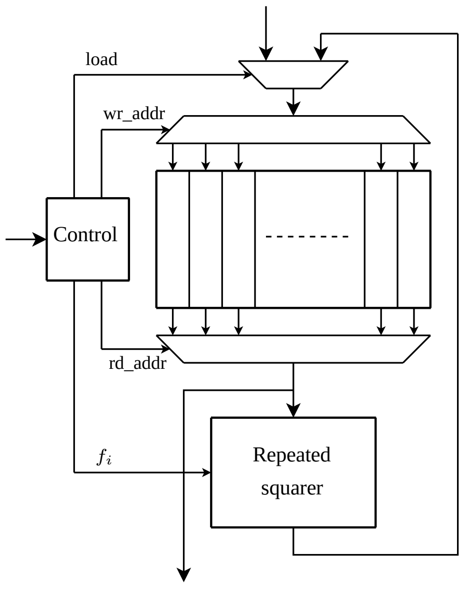


Fig. 3: Register bank for storing the precomputed points and a repeated squarer for computing $\phi^{f_i}(P_j)$ for $j = 1, \ldots, N$

point addition, the repeated squarer performs $\phi^{f_2}(P_1), \ldots, \phi^{f_2}(P_N)$;

6) After this, scalar multiplication proceeds so that when the multipliers are computing interleaved point additions with k_i and k_{i-1}, the repeated squarer is already updating the register bank by computing $\phi^{f_{i+1}}(P_1), \ldots, \phi^{f_{i+1}}(P_N)$.

Obviously, the critical path consists only of the point additions (and Frobenius maps with f_0 and f_1) if the Frobenius maps are faster than the point additions, which require only two multiplications as shown above.

Notice that because the encoding presented in Sec. 3.1 was designed so that it ensures that $f_i \leq e_{\max}$ for all i and, consequently, each $\phi^{f_i}(P_j)$ has a constant latency of two clock cycles, it is easy to design the main processor so that the Frobenius maps are always faster than two multiplications. In order to ensure that, the main processor architecture must satisfy

$$2N + p \leq 2M_2 \quad (5)$$

where M_2 is the latency of multiplication and p is the number of pipeline stages in the repeated squarer. Above we assumed that each repeated squaring requires only one clock cycle, i.e., $p = 0$ but the repeated squarer can be pipelined.

In our implementation of the processor, the repeated squarer attached to the register bank has $e_{\max} = 10$ and its computation was pipelined with $p = 1$. Therefore, it can compute $\phi^{f_i}(P_i)$, where $f_i \leq 10$, for N points in $2N + 1$ clock cycles. Because the width-4 τNAF is used, the number of points is $N = 5$ (four precomputed points and an extra point). As a consequence, Frobenius maps require 11 clock cycles and, in order to ensure that Frobenius maps are not on the critical path, one must select $M_2 \geq 6$ for the multipliers in the main processor.

4.4 Postprocessor

The postprocessor maps the result point $Q = (X, Y, Z)$ from the main processor to affine coordinates by computing $(x = X/Z, y = Y/Z^2)$. It includes a multiplier and a repeated squarer [23] for computing a^{2^e} where $e \in \{1, 2, 4\}$. The postprocessor computes the inversion as proposed in [21].

4.5 Latency of scalar multiplication

Let D_1, D_2, and D_3 denote the digit sizes and M_1, M_2, and M_3 the latencies of the multipliers in the preprocessor, the main processor, and the postprocessor, respectively. The latencies of different operations in clock cycles are given in Table 1. As shown in Table 1, the latencies of all other operations except τNAF conversion can be tuned by varying the latencies of multipliers, which are given by $M_i = \lceil m/D_i \rceil + 1$, where $i \in \{1, 2, 3\}$.

Fig. 4 shows an example scalar multiplication with the processor (shown in white). It also shows how the pipeline

Table 1: Latencies

Operation	Latency (clock cycles)
Conversion, width-4 τNAF	335
Precomputation	$18M_1 + 327$
For-loop	$2(w-2)(M_2+1) + 3(2N+p) + 18$
Coordinate conversion	$11M_3 + 59$

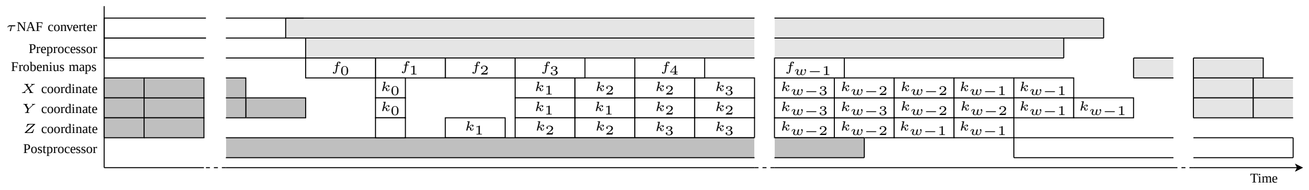


Fig. 4: Example computation schedule for computing scalar multiplications with the processor. The previous and the next scalar multiplication processed in the pipeline are shown in dark and light grey, respectively.

works by presenting the previous (shown in dark grey) and the next (shown in light grey) scalar multiplication. The computation proceeds so that, first, the τ-adic conversion and the precomputation are performed concurrently. The τ-adic conversion could be performed concurrently also with the main for-loop but this would not bring any benefit, because precomputation would still be on the critical path. When both are ready, the for-loop computation is started in the main processor. The computation proceeds as discussed in Sec. 4.3, and the Frobenius maps are computed simultaneously with the point additions. It is clearly visible that the X, Y, and Z units are processing two point additions simultaneously; each block in Fig. 4 denotes one iteration of the units. When the for-loop is ready, the result point (X, Y, Z) is converted into $\mathcal{A}$ in the postprocessor. This computation can be started immediately when Z is ready; i.e., while X and Y are still being computed.

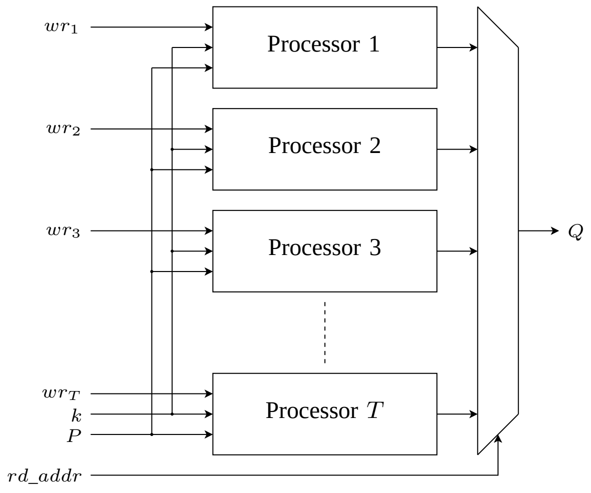


Fig. 5: Parallel architecture

5. Parallelization

It was conjectured in [20] that a throughput of over 1,000,000 scalar multiplications per second can be achieved with a single modern FPGA. In this section, we seek to verify this conjecture by studying what is the maximum throughput achievable with a single FPGA implementation consisting of several parallel instances of the processor presented above.

The target for the optimizations presented in [20] was to maximize the throughput of a single processor. In order to achieve this goal, the multipliers used in the main processor used the digit size of $D_2 = 17$ ($M_2 = 11$). This digit size was the largest one that still ensured that the critical path of computations remained in the main processor. In this paper, however, our target is to maximize the throughput of several parallel instances of the processor. Instead of maximizing the throughput of the processor, we focus on maximizing the throughput-area ratio of the processor. In that case, it is possible that the optimum is reached with different digit sizes. We opted to maximize the throughput-area ratio of a single processor and then replicate several instances of that processor although it is possible that one could be able to achieve slightly higher throughput by aiming to fill the entire FPGA (or some fixed percentage of it). Our approach, however, provides more general results and, of course, a better throughput-area ratio.

The approach of our implementation is simple: We replicate T parallel instances of the processor architecture presented in Sec. 4, all with the parameters that maximize the throughput-area ratio of a single processor, and provide a common interface for them. This architecture is shown in Fig. 5.

As explained above, the dominating component in our processor architecture is the main processor. Hence, we fit the parameters of the other components based on the

Table 2: Balanced setups

Setup	D_1 M_1	D_2 M_2	D_3 M_3
1	2 (83)	6 (29)	1 (164)
2	3 (56)	7 (25)	2 (83)
3	3 (56)	8 (22)	2 (83)
4	3 (56)	9 (20)	2 (83)
5	4 (42)	10 (18)	2 (83)
6	4 (42)	11 (16)	2 (83)
7	5 (34)	12 (15)	2 (83)
8	5 (34)	13 (14)	3 (56)
9	6 (29)	14 (13)	3 (56)
10	7 (25)	15 (12)	3 (56)
11	7 (25)	17 (11)	3 (56)

parameters of the main processor. As shown in Table 1, the latencies are determined solely by the latencies of the multipliers ($w = 4$, $N = 5$, and $p = 1$ are fixed). We choose the digit sizes of the multipliers in the preprocessor and the postprocessor, D_1 and D_3, so that they are the smallest digit sizes that still ensure that the bottleneck (i.e., the longest latency) is in the main processor. We call a setup (D_1, D_2, D_3) that satisfies this condition a balanced setup.

The (average) latency of the conversion is constant: 335 clock cycles; i.e., the latency cannot be tuned by choosing design parameters (e.g., multiplier digit size). Moreover, the converter uses a different clock than the rest of the architecture. Based on the work in [20], we assume that we can use about 2.2 times faster clock for the other parts of the architecture (in [20], we had 85 MHz and 185 MHz clocks). Hence, we estimate that the latency of the converter corresponds to about $2.2 \times 335 \approx 740$ clock cycles with the other clock. The latency of the main processor should not get shorter than this latency or else the converter will become the bottleneck and all area that is used for making the main processor faster does not improve the overall throughput of the processor. Based on Table 1, it is clear that if $D_2 > 17$, the latency of the main processor is smaller than 740. In Sec. 4.3, we derived a lower bound of $D_2 \geq 6$ for the the same digit size that ensured that the Frobenius maps do not appear on the critical path. Hence, only digit sizes $6 \leq D_2 \leq 17$ are viable. The balanced setups satisfying these constraints are collected in Table 2.

Because there are no other trustworthy methods to get exact (post-place&route) area requirements of an FPGA implementation besides running synthesis and place&route for the design, we determined throughput-area ratios by compiling processors with different balanced setups for Stratix IV GX 4SGX230NC2 with Quartus II ver. 10.1. This FPGA is used, for instance, in Stratix IV GX FPGA Development Board [28]. The results are collected in Table 3. We started compilations from setup 11 downwards (see Table 2) and it soon became apparent that the best throughput-area ratio is achieved with setup 11, the largest balanced setup. This means that the maximum throughput-area ratio of the main processor is achieved with $D_2 \geq 17$. Unfortunately, as noted above, we cannot utilize setups with $D_2 > 17$ because then the throughput will be bounded by the throughput of the converter which roughly equals the throughput of the main processor with $D_2 = 17$.

As shown in Table 3, setup 11 occupies approximately 15 % of the resources (ALMs) available on the FPGA. This implies that we could fit six such processors on a single chip by using approximately 90 % of the resources. However, area requirements tend to grow more than linearly when the size of the design approaches the limits of the device because place&route has a more difficult task to fulfill timing constraints. Hence, we use only five parallel instances of the processor in our implementation. That is, we prepared a prototype implementation using the architecture depicted in Fig. 5 with $T = 5$. The results for this implementation are given in Sec. 6.

6. Results

The parallel architecture shown in Fig. 5 and described in Sec. 4 and 5 was realized with the parameters obtained in Sec. 5, i.e., $T = 5$, $D_1 = 7$, $D_2 = 17$, and $D_3 = 3$. The implementation was described in VHDL and compiled for an Altera Stratix IV GX EP4SGX230KF40C2 FPGA with Quartus 10.1. We emphasize that the processor architecture is not restricted to any specific FPGA but the optimal parameters may vary between different FPGAs.

The area consumptions are collected in Table 4. Timing constraints of 120 MHz and 266 MHz were set for the τ-adic converter and the rest of the processor, respectively, and the compiler was able to meet these constraints.

Using the clock frequencies of 120 MHz and 266 MHz and the latencies derived in Sec. 4.5, we get the following average timings: width-4 τNAF conversion $335/120 = 2.79\,\mu$s, precomputations $777/266 = 2.92\,\mu$s (constant), for-loop $785.4/266 = 2.95\,\mu$s, and coordinate conversion $675/266 = 2.53\,\mu$s (constant). The throughput of the processor is bounded by the main processor; hence, the theoretical maximum throughput is 338,680 scalar multiplications per second for a single processor and 1,693,000 scalar multiplications per second for the parallel implementation of five processors. The average timing for a single scalar multiplication is $8.3\,\mu$s.

7. Conclusions

We described a parallel implementation of elliptic curve cryptography with several parallel instances of the processor introduced in [20]. This implementation is capable of delivering a theoretical maximum throughput of about 1,700,000 scalar multiplications per second on the standardized elliptic curve NIST K-163 [9].

Contrary to many other published works, this study and the earlier versions of this work presented in [19], [20] focused on maximizing the throughput, i.e., scalar multiplications per second instead of minimizing the computating

Table 3: Throughput-area ratios of selected processors with balanced setups on Stratix IV GX FPGA

Setup	ALUTs	Regs	ALMs	Memory M9K	Time (μs)	Throughput (1/s)	Throughput/Area (1 / s·ALM)
11	16,001	12,377	13,768	21	8.3	339,000	26.51
10	14,568	12,360	13,163	21	8.6	314,000	23.87
9	14,531	12,371	12,778	21	9.1	293,000	21.28

Table 4: Results on Stratix IV GX EP4SGX230KF40C2.

ALUTs	78,695 (43 %)
Regs	61,871 (34 %)
ALMs	74,750 (82 %)
M9K	105 (9 %)
Clock, converter	120 MHz
Clock, others	266 MHz
Time	8.3 μs
Throughput	1,693,000

time of a single scalar multiplication. The difference to [19], [20] is that they focused on maximizing the throughput of a single processor, whereas this paper presented a study which aimed to maximizing the throughput of an implementation comprising several processors. The results showed that both the maximum throughput and the maximum throughput-area ratio are achieved with similar setups. The main component of the processor would be capable of producing even higher throughput-area ratios but, in that case, another component, namely the converter, would become the limiting factor. Hence, there is an obvious need for even faster converter architectures. Very recently a faster (but slightly larger) converter was presented in [29] and it would allow using setups with even higher throughput-area ratios.

In this paper, we focused on a single FPGA; namely, Altera Stratix IV GX EP4SGX230KF40C2. It is possible (and even likely) that a better performance-price ratio is achieved with budget FPGAs, e.g., from Altera Cyclone family. Although the balanced setup that delivers the best performance (in this case, it was setup 11 from Table 2) may vary between diffferent FPGAs, the implementation architecture and the methodology are generic.

The results show that modern FPGAs are able to deliver extremely high throughtputs for secure public-key cryptography, reaching as high as 1,700,000 scalar multiplications per second on a secure elliptic curve. The implementation presented in this paper can have applications, e.g., in accelerating cryptographic operations in very highly loaded network servers. Other interesting applications could be found in cryptanalytic hardware. For example, the FPGA-based machine designed for cryptanalytic purposes called COPACOBANA [30] could be programmed to implement the proposed parallel architecture (or a modification of it) which could have some cryptanalytic importance.

References

[1] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comput.*, vol. 48, pp. 203–209, 1987.

[2] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology, CRYPTO 1985*, ser. Lecture Notes in Comput. Sci., vol. 218. Springer, 1986, pp. 417–426.

[3] A. K. Lenstra and E. R. Verheul, "Selecting cryptographic key sizes," *J. Cryptol.*, vol. 14, no. 4, pp. 255–293, Dec. 2001.

[4] H. Eberle, N. Gura, S. C. Shantz, V. Gupta, L. Rarick, and S. Sundaram, "A public-key cryptographic processor for RSA and ECC," in *Proc. 15th IEEE Int. Conf. Application-Specific Systems, Architectures and Processors, ASAP 2004*, 2004, pp. 98–110.

[5] K. Sakiyama, N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede, "Reconfigurable modular arithmetic logic unit supporting high-performance RSA and ECC over $GF(p)$," *Int. J. Electron.*, vol. 94, no. 5, pp. 501–514, May 2007.

[6] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[7] G. Meurice de Dormale and J.-J. Quisquater, "High-speed hardware implementations of elliptic curve cryptography: A survey," *J. Syst. Architect.*, vol. 53, no. 2-3, pp. 72–84, Feb.-Mar. 2007.

[8] T. Wollinger, J. Guajardo, and C. Paar, "Security on FPGAs: State-of-the-art implementations and attacks," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 534–574, Aug. 2004.

[9] National Institute of Standards and Technology (NIST), "Digital signature standard (DSS)," Federal Information Processing Standard, FIPS PUB 186-3, June 2009.

[10] N. Koblitz, "CM-curves with good cryptographic properties," in *Advances in Cryptology, CRYPTO '91*, ser. Lecture Notes in Comput. Sci., vol. 576. Springer, 1991, pp. 279–287.

[11] S. Okada, N. Torii, K. Itoh, and M. Takenaka, "Implementation of elliptic curve cryptographic coprocessor over $GF(2^m)$ on an FPGA," in *Cryptographic Hardware and Embedded Systems, CHES 2000*, ser. Lecture Notes in Comput. Sci., vol. 1965. Springer, 2000, pp. 25–40.

[12] J. Lutz and A. Hasan, "High performance FPGA based elliptic curve cryptographic co-processor," in *Proc. Int. Conf. Information Technology: Coding and Computing, ITCC 2004*, vol. 2, 2004, pp. 486–492.

[13] V. S. Dimitrov, K. U. Järvinen, M. J. Jacobson, W. F. Chan, and Z. Huang, "FPGA implementation of point multiplication on Koblitz curves using Kleinian integers," in *Cryptographic Hardware and Embedded Systems, CHES 2006*, ser. Lecture Notes in Comput. Sci., vol. 4249. Springer, 2006, pp. 445–459.

[14] ——, "Provably sublinear point multiplication on Koblitz curves and its hardware implementation," *IEEE Trans. Comput.*, vol. 57, no. 11, pp. 1469–1481, Nov. 2008.

[15] O. Ahmadi, D. Hankerson, and F. Rodríguez-Henríquez, "Parallel formulations of scalar multiplication on Koblitz curves," *J. Univers. Comput. Sci.*, vol. 14, no. 3, pp. 481–504, 2008.

[16] K. Järvinen, J. Forsten, and J. Skyttä, "FPGA design of self-certified signature verification on Koblitz curves," in *Cryptographic Hardware and Embedded Systems, CHES 2007*, ser. Lecture Notes in Comput. Sci., vol. 4727. Springer, 2007, pp. 256–271.

[17] K. Järvinen and J. Skyttä, "On parallelization of high-speed processors for elliptic curve cryptography," *IEEE Trans. VLSI Syst.*, vol. 16, no. 9, pp. 1162–1175, Sept. 2008.

[18] ——, "Fast point multiplication on Koblitz curves: Parallelization method and implementations," *Microproc. Microsyst.*, vol. 33, no. 2, pp. 106–116, Mar. 2009.

[19] K. U. Järvinen and J. O. Skyttä, "High-speed elliptic curve cryptography accelerator for Koblitz curves," in *Proc. IEEE 16th IEEE Symp. Field-programmable Custom Computing Machines, FCCM 2008*. IEEE Computer Society, 2008, pp. 109–118.

[20] K. Järvinen, "Optimized FPGA-based elliptic curve cryptography processor for high-speed applications," *Integration—the VLSI Journal*, in press.

[21] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases," *Inform. Comput.*, vol. 78, no. 3, pp. 171–177, Sept. 1988.

[22] M. A. Hasan, "Look-up table-based large finite field multiplication in memory constrained cryptosystems," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 749–758, July 2000.

[23] K. U. Järvinen, "On repeated squarings in binary fields," in *Selected Areas in Cryptography, SAC 2009*, ser. Lecture Notes in Comput. Sci., vol. 5867. Springer, 2009, pp. 331–349.

[24] J. López and R. Dahab, "Improved algorithms for elliptic curve arithmetic in $GF(2^n)$," in *Selected Areas in Cryptography, SAC'98*, ser. Lecture Notes in Computer Science, vol. 1556. Springer, 1999, pp. 201–212.

[25] E. Al-Daoud, R. Mahmod, M. Rushdan, and A. Kilicman, "A new addition formula for elliptic curves over $GF(2^n)$," *IEEE Trans. Comput.*, vol. 51, no. 8, pp. 972–975, Aug. 2002.

[26] J. A. Solinas, "Efficient arithmetic on Koblitz curves," *Des. Codes Cryptography*, vol. 19, no. 2–3, pp. 195–249, 2000.

[27] B. B. Brumley and K. U. Järvinen, "Conversion algorithms and implementations for Koblitz curve cryptography," *IEEE Trans. Comput.*, vol. 59, no. 1, pp. 81–92, 2010.

[28] Altera, "Stratix IV GX FPGA development board: Reference manual," Aug. 2010, http://www.altera.com/literature/manual/rm_sivgx_fpga_dev_board.pdf.

[29] J. Adikari, V. Dimitrov, and K. Järvinen, "A fast hardware architecture for integer to τNAF conversion for Koblitz curves," *IEEE Trans. Comput.*, in press.

[30] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp, "Cryptanalysis with COPACOBANA," *IEEE Trans. Comput.*, vol. 57, no. 11, pp. 1498–1513, Nov. 2008.

SESSION

REGULAR PAPERS

Chair(s)

PROF. RYAN KASTNER

Hardware Architecture for Simultaneous Arithmetic Coding and Encryption

Amit Pande[1], **Joseph Zambreno**[2], **and Prasant Mohapatra**[1]
[1]Department of Computer Science, University of Calofirnia, Davis, CA, USA
[2]Electrical and Computer Engineering, Iowa State University, IA, USA
email: amit@cs.ucdavis.edu, zambreno@iastate.edu, prasant@cs.ucdavis.edu

Abstract—*Arithmetic coding is increasingly being used in upcoming image and video compression standards such as JPEG2000, and MPEG-4/H.264 AVC and SVC standards. It provides an efficient way of lossless compression and recently, it has been used for joint compression and encryption of video data. In this paper, we present an interpretation of arithmetic coding using chaotic maps. This interpretation greatly reduces the hardware complexity of decoder to use a single multiplier by using an alternative algorithm and enables encryption of video data at negligible computational cost. The encoding still requires two multiplications. Next, we present a hardware implementation using 64 bit fixed point arithmetic on Virtex-6 FPGA (with and without using DSP slices). The encoder resources are slightly higher than a traditional AC encoder, but there are savings in decoder performance. The architectures achieve clock frequency of 400-500 MHz on Virtex-6 xc6vlx75 device. We also advocate multiple symbol AC encoder design to increase throughput/slice of the device, obtaining a value of 4.*

Keywords: arithmetic coding, hardware implementation, chaotic maps, multimedia encryption

1. Introduction

The state-of-the-art video coding standards such as SVC (technically Annex G of MPEG-4/H.264 AVC) [1] is been widely adopted in current video application systems due to its outstanding coding performance, and scalable properties which allow deployment in fluctuating channel conditions and to serve heterogeneous clients. There are three entropy coding tools adopted in H.264/SVC. One is Context-based Adaptive Binary Arithmetic Coding (CABAC), based on arithmetic coder. The other are Context-based Adaptive Variable Length Coding (CAVLC) and Exp-Golomb coding (to code syntax elements). CABAC can achieve averaged bit-rate savings of 9% to 14% at the cost of higher computational complexity in comparison to CAVLC. However, the increased computational complexity and strong data dependencies significantly restrict the throughput of CABAC decoder. This restriction becomes a challenge in hardware design of CABAC coder making CAVLC more suitable for decoding in low-power embedded systems.

Arithmetic coding is a data compression technique that encodes data by creating a code string which represents a fractional value on the interval [0, 1). When a string is compressed using arithmetic coder, frequently-used characters are stored with fewer bits and not-so-frequently occurring characters are stored with more bits, resulting in fewer bits used in total [2]

This paper discusses arithmetic coding from a slightly different perspective. Recent work has established how arithmetic coding can be viewed as an iteration on piece-wise linear chaotic maps [3], [4]. Further, many researchers have studied the use of arithmetic coding for joint encryption and compression [5], [6], [7]. For example- In [8], a chaos-based adaptive arithmetic coding technique was proposed. The arithmetic coder's statistical model is made varying in nature according to a pseudo-random bitstream generated by coupled chaotic systems. Many other techniques based on varying the statistical model of entropy coders have been proposed in literature, however these techniques suffer from losses in compression efficiency that result from changes in entropy model statistics and are weak against known attacks [9]. Recently, Grangetto et al. [5] presented a Randomized Arithmetic Coding (RAC) scheme which achieves encryption by inserting some randomization in the arithmetic coding procedure at no expense in terms of coding efficiency. RAC needs a key of length 1-bit per encoded symbol. Kim et al. [6] presented a generalization of this procedure, called as Secure Arithmetic Coding (SAC). The SAC coder builds over a Key-Splitting Arithmetic Coding where a key is used to split the intervals of an arithmetic coder, adding input and output permutation to increase the coder's security.

In this paper, we extend this discussion to hardware community - to study the hardware optimizations in design of such schemes. Particularly, we study the implementation of arithmetic coding using piece-wise chaotic maps [3], [4]. As we shall study, this implementation has lower decoder requirements than the commercial implementations. Apart from these, chaotic maps have also been used in cryptography and for pseudo random number generation [10].

The reduced decoding efficiency of arithmetic coding allows it to trend towards the low computational complexity of Huffman coders, allowing BAC to enter embedded systems market. The aspects of context-modeling and adaptation

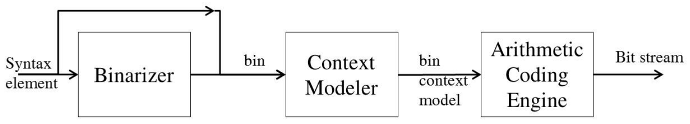


Fig. 1: Block diagram of CABAC coder

and renormalization, as done in CABAC coder are beyond the scope of this work, where we focus on architectural optimizations on encoder and decoder processes.

Why another design?

An inquisitive question which comes to mind at this point is the need for hardware implementation of chaotic maps. When arithmetic coding is already been done using traditional ways, why do we need yet another architecture?

The motivation to develop a hardware architecture for chaotic maps iterations is summarized below:

1) Arithmetic coding done using chaotic maps is asymmetric in nature, (explained in later sections) making the decoder architecture simpler than existing framework for AC. The reduced decoder complexity is highly desired to reduce the power and computational requirements of video decoding in low power mobile devices. Current mobile video profiles use Huffman coding instead of Arithmetic coding to reduce the computational complexity, which leads to average compression inefficiency of 15%, particularly poor performance in coding events with symbol probabilities greater than 0.5, due to the fundamental lower limit of 1 bit/symbol on Huffman coding [11].
2) Recently, arithmetic coding based encryption schemes have been proposed in research literature for joint compression and encryption purposes [7], [12]. It would be interesting to integrate both coding and encryption using chaotic maps at a computational complexity lower than existing implementations. This motivates the need of coding and encryption architecture using chaotic maps.
3) Chaotic maps can be used to Pseudo-Random Number Generation (PRNG) [10] and stream ciphers [13], apart from arithmetic coding. These have been found to be light weight and simple.

Contributions

The main contributions of this paper are as follows:

1) We introduce arithmetic coder architecture using chaotic maps which has potential advantages in reducing decoder complexity and allows combined encryption.
2) We present two architectures for FPGA implementation of the proposed scheme: one using explicit multipliers from DSP48E1 slices on Virtex-6 FPGA, while other using reconfigurable multipliers and mapping to hardware 6-LUTs.
3) We advocate the multiple-symbol encoding which makes sense for throughput/ area.

Scope of the work

In the regular coding mode, prior to the actual arithmetic coding process the given binary data enters the context modeling stage, where a probability model is selected such that the corresponding choice may depend on previously encoded syntax elements. Then, after the assignment of a context model, the bin value along with its associated model is passed to the regular coding engine, where the final stage of arithmetic encoding together with a subsequent model updating takes place (see Figure 1). We shall restrict the focus of further discussions on the final arithmetic encoding (and decoding) stages of CABAC coder.

2. Literature Review

Adaptive minimum-redundancy (Huffman) coding is expensive in both time and memory space, and is handsomely outperformed by adaptive AC besides the advantage of AC in compression effectiveness [14]. FenwickŠs structure requires just n words of memory to manage an n-symbol alphabet, whereas the various implementations of dynamic Huffman coding [15], [16] consume more than 10 times as much memory [17].

Hardware architectures have been proposed in research literature for arithmetic coding using CACM model [18] or related works [14], [19], [20]. CABAC or Context-Adaptive Binary Arithmetic Coder is used in H.264 AVC and SVC. The critical path of coder is the multiplier, which is removed in CABAC and recent implementations [21], [22], [23] by using a look-up approximation (leading to some compression inefficiency).

There has been little work [24], [25], however, in implementation of chaotic maps on hardware. However, the recent trend toward joint compression and encryption using chaotic maps and arithmetic coding for low power embedded systems would be greatly complimented by an efficient hardware architecture, as presented in this paper.

Binary Arithmetic Coding (BAC)

Binary arithmetic coding is based on the principle of recursive interval subdivision. We start with an initial interval

[0,1) and keep dividing it into subintervals based on the probability of incoming symbols. A good detailed overview of BAC is presented in [14]

3. How do we interpret AC using Chaotic Maps?

A description of equivalence between binary arithmetic coding and chaotic maps is given in earlier works [4], [7]. In this section, we gave a brief overview of N-alphabet arithmetic coding to familiarize the reader with coding using piece-wise linear chaotic maps.

Scenario: We have a string $S = x_1, x_2, ...x_M$ consisting of M symbols (N unique symbols) to be encoded. The probability of occurrence of a symbol $s_i,\ i \in 1, 2, ...n$ is given by p_i such that $p_i = N_i/N$ and N_i is the number of times the symbol s_i appears in the given string S.

Description: Consider a piece-wise linear map (ρ) with the following properties:

- It is defined on the interval $[0, 1)$ to $[0, 1)$ i.e.
$$\rho : \ [0, 1) \longrightarrow [0, 1)$$
- The map can be decomposed into N piece-wise linear parts ϱ_k i.e.
$$\rho = \bigcup_{k=1}^{N} \varrho_k$$
- Each part ϱ_k maps the region on x axis $[beg_k, end_k)$ to the interval $[0, 1)$ i.e.
$$\varrho_k : \ [beg_k, end_k) \longrightarrow [0, 1]$$
The last two propositions lead to:
$$\bigcup_{k=1}^{N} [beg_k, end_k) = [0, 1)$$
- The map ϱ_k is one-one and onto i.e.:
$$\forall x \in [beg_k, end_k)$$
$$\exists y \in [0, 1) : y = \varrho_k(x), \text{ and}$$
$$\forall y \in [0, 1)$$
$$\exists x \in [beg_k, end_k) : \varrho_k(x) = y$$
- ρ is a many-one mapping from $[0, 1)$ to $[0, 1)$. This implies that the decomposed linear maps (ϱ_k) don't intersect each other i.e.
$$\forall (k \neq j) : [beg_k, end_k) \bigcap [beg_j, end_j) = 0$$
- Each linear map ϱ_k is associated uniquely with one symbol s_i. The mapping $\varrho_k \longrightarrow s_i$ is defined arbitrarily but one-one relationship must hold.
- The valid-input width of each map (ϱ_k), given by $(end_k - beg_k)$ is proportional to a probability of occurrence of symbol s_i.
$$end_k - beg_k \propto p_i$$
$$\Rightarrow end_k - beg_k = C \times p_i$$
We recall that $\sum_{k=1}^{N} (end_k - beg_k)$ is same as the input width of $\bigcup_{k=1}^{N} \varrho_k = \rho$, which is 1. Also, $\sum_{i=1}^{N} p_i = 1$. Thus, we get the value of constant C to be 1.
$$\Rightarrow end_k - beg_k = p_i$$

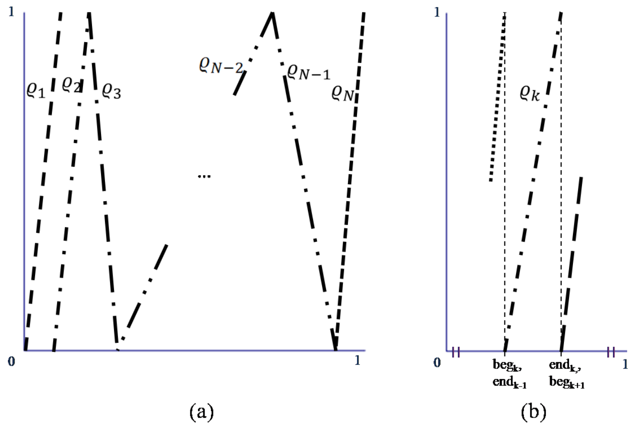


Fig. 2: A sample piece-wise linear map for arithmetic coding like compression (a) The entire map is shown (ρ) (b) A single linear part of the map (ϱ_k) is zoomed. It can have a positive or negative slope depending on choice

Figure 2 shows a sample map fulfilling these properties. Figure 2(a) shows the full map with different parts $\varrho_1, \varrho_2, ...\varrho_N$ present while Figure 2(b) zooms into individual linear part ϱ_k. The maps are placed adjacent to each other so that each input point is mapped into an output point in the range $[0, 1)$.

Encoding/ Decoding

The decoding process is quite simple. The encoded value is considered as an initial value IV. This value is iterated over the piece-wise linear map ρ, M times to get M iterated values IV_i. Each value is mapped to piece-wise linear part ϱ_i and thus to corresponding s_i.

The encoding process is done by reversing the input string to $x_M, x_{M-1}, ...x_1$. Each input character is mapped to unique symbols s_i and then to piece-wise linear maps ϱ_i. Thus, we get a sequence of piece-wise linear maps corresponding to input string $\varrho_{x_M}, \varrho_{x_{M-1}} ... \varrho_{x_1}$. We start with the initial interval [0,1) and back-iterate this interval over chaotic maps using the string $\varrho_{x_M}, \varrho_{x_{M-1}} ... \varrho_{x_1}$ to get a final interval. The output codeword is chosen as the shortest binary number from final interval.

Compression Efficiency and Equivalence

Arithmetic coding has been shown to be achieve Shannon's limit on compression efficiency asymptotically. The same result holds true for coding using piecewise linear maps because of the following observations:

The width of final interval is given by $\lceil \prod_{i=1}^{N} f_i^{n_i} \rceil$, where f_i is the probability of occurance of symbol s_i, and n_i is the frequency of occurance of symbol s_i. This value asymptotically approaches Shannon's value for maximum entropy []. It can be observed that while CAC scales the codeword or initial value to map them to the intervals corresponding to different symbols, the standard arithmetic coder keeps the codeword constant and instead scales the map in every iteration to find the symbol. It is immaterial - whether one scales the map to suit the codeword or scales the codeword to suit the map - the relative ratios remain the same, hence output of both procedures is the same.

Use of Chaotic Maps in Encryption

[12], [7] present two different scenarios of using chaotic maps for arithmetic encryption. The first case uses N-ary arithmetic coding and has high cryptographic strength and implementation cost, while the second case uses binary arithmetic coding to encrypt data with low computational resources. In both the cases, the choice of multiple piece-wise linear maps to encode the input symbol is used for key generation. This property is used for encryption, for without knowledge of the correct map, an adversary cannot decode the input stream correctly.

Applications

The CAC can be used as a joint compression-cum-encryption technique for data encryption. It is particularly beneficial for data-intensive tasks such as multimedia encryption and compression and can be integrated into the standard video compression algorithms such as JPEG2000, JPEG, MPEG etc.

CAC can be used for full or selective encryption of multimedia data. For full encryption, the entire volume of multimedia data is passed through BCAC (Binary CAC) encoder while in case of selective encryption only the important parts of data are passed through BCAC encoder. If we reveal the first K bits of the key publicly, then a part of the bitstream can be decoded correctly while decoding the entire bitstream will require knowledge of the entire key. Thus, BCAC can be used to provide conditional access to the multimedia content.

4. Hardware architecture

In this section, we discuss the hardware architecture for arithmetic coding using chaotic maps, and N-ary chaotic arithmetic encryption.

The chaotic encoder operation inverse inverse mapping of interval [0,1) on the chaotic map according to input symbol. For binary arithmetic coder, we have a fixed map to be iterated in each cycle.

Figure 3(a) shows the basic architecture for coding using chaotic maps. The control unit receives the input bit stream, which is passed on to the chaotic map Iterator (CMI). The control unit passes the bitstream, one symbol per cycle (unless in the case of multiple symbol encoding, which will be discussed later). For encoding, the initial interval passed to CMI is [0,1), which is transmitted as the beginning (B_n) and end (E_n) interval values. Both the intervals are then iterated over CMI (using two instances of CMI), and then the output is sorted so that $B_n < E_n$. If the difference $(D_n = E_n - D_n)$ is lower than a threshold, we need to renormalize the encoder. The renormalization procedure for arithmetic coding has been discussed in [14]. A similar extension of renormalization procedure may be possible for chaotic maps. But, for the evaluation designs considered in this work, we have considered 64 bit encoder without any renormalization procedure.

In case of decoding, Control Unit (CU) transmits the coded symbol into CMI, which is then iterated over Piece-wise linear map and reported back to CU. The CU makes a comparison with chaotic map indicated by the key and outputs a single bit output.

CMI has a multiplier and an adder to perform chaotic iteration. The internal details of this operation are given in Figure 3(b). The multiplication and addition coefficients are obtained from a look-up table/ RAM collating the input symbol, key value and probability value as the input address. The Look-ed up value or a word is demultiplexed to obtain the multiplication and addition coefficients. This option can work fine for at most binary case, and for the case where p value is limited to fixed precision, say 8 bits. Such fixed precision approximations have been introduced in CABAC [11], however it leads to approximation of results. Alternatively, we can use a multiplexer which can implement look-up using physical circuits to compute the return values. The second approach has been implemented in this work, as it allows more flexibility in design and accuracy in computation.

For implementation, the input and output intervals to the Chaotic Map Iterator are represented in 64 fixed point (0 bits integer and 64 bits fraction, shortly I.F 0.64) arithmetic. The symbol probability has been quantized to 8 bits (I.F 0.8).

Binary Arithmetic Coder (BAC) architecture

To implement BAC in proposed architecture, we target a design with processes 1 symbol (1 bit in this case) per cycle. The CMI has 1 bit symbol input, 8 bit symbol probability and no bits for choice of chaotic map (there is only one map in this case). The 9 bit lookup can be implemented using a 512 words RAM or Look-up Table. One word is 16 bits - 8 bits each for multiplication and addition coefficients. Alternatively, this can be implemented using a multiplexer and hardware adder/ subtracter to obtain the coefficients. The later approach was used for BAC implementation. The design was synthesized in Xilinx Virtex-6 XC6VLX75t

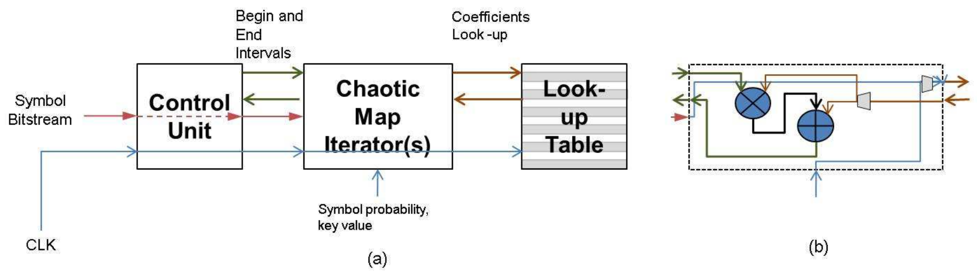


Fig. 3: Generalized Hardware Architecture for Chaotic Maps. (a) Generalized architecture and (b) Circuit details for Chaotic map Iterator

FPGA using Xilinx ISE Design Suite 12.0 environment. The same target FPGA, which is one of the low end Virtex-6 family member is used in all synthesis/ translate/ map/ place and routes.

The two 64x8 bit multiplications are mapped in hardware into 10 DSP48E slices. A slice usage of 302 was obtained and the design achieved a clock frequency of 510 MHz, with one symbol per clock cycle. The optimized implementation of multiplication, using carry-chains of FPGA fabric was synthesized to remove the use of DSP slices. This implementation requires 1585 slices and achieves a clock frequency of 500 MHz. The throughput of this implementation is 1 bit per cycle with a 500 MHz clock, i.e. 500 Mbps.

Binary Chaotic Arithmetic Coder and Encryption (BCAC) architecture

The architecture for BCAC differs from binary arithmetic coder in the sense that, the choice of chaotic map is made based on a key value, and is not precomputed. For this implementation, the CMI has 1 bit symbol input, 8 bit symbol probability and 3 bits for choice of chaotic map (for binary case $N = 2$, hence number of different chaotic maps is $N2^N = 8$. The 12 bit lookup can be implemented using a 512 words RAM or Look-up Table, with 16 bits word. Alternatively, we used 8-to-1 multiplexer to obtain the coefficients corresponding to a key, each cefficient being generated based on value in Table 1 in [7]. The implementation on target FPGA gave a clock frequency of 500 MHz, utilizing 321 slices and 10 DSP48E1 slices (which have optimized multiplier and accumulator operation implemented in VLSI). Mapping these multiplication to FPGA logic increased the slice usage to 1474, without any change in achievable clock frequency.

The BCAC decoder hardware utilization was 173 slice LUT with 5 DSP slices (806 slice LUTs with LUT multiplier) with a clock frequency of 510 MHz (500 MHz). The 64x8 bit multiplier is implemented by ISE into 5 DSP slices. However, the same multiplier can be optimized and implemented without hardware multipliers using other multiplier such as square root multiplier, reconfigurable constant multipliers etc. The hardware requirements are basically dependent on size of Look-up logic which increases exponentially with increase of N. The throughput of this implementation is 1 bit per cycle with a 510 MHz clock, i.e. 510 Mbps. To consider the area effectiveness of this design, we consider throughput per slice, with the second implementation where we implement multiplication in LUTs rather than using DSP48E1 slices present in device. The throughput/ slice for this design is obtained as 322 Kb/slice.

Cost of encryption

Comparing the BAC and BCAC architectures, we obtain a zero latency, same throughput and little hardware overhead (20 slice LUTs) in implementing this encryption scheme against AES or other schemes which have significant overhead. For instance,Chang et al. [26] reports AES implementation using 156 slices, 2 Block RAMs to obtain a lower clock of 306 MHz.

To increase the throughput per slice for a bitstream, we intuitively consider the dimension of increasing the number of symbols in dictionary used in arithmetic coding. For example - considering 3 or 4 symbols in the dictionary.

N-ary Chaotic Arithmetic Coder and Encryption (NCAC) coding

N-ary arithmetic encryption using the entire possible key space quickly turns out-of-bounds for a FPGA device. Moving from 2 to 3 piece-wise linear maps, we have a tremendous increase in key-size. We implemented tri-nary CAC coder in FPGA device to obtain a device usage of 492 slices and 10 DSP48E slices (1800 slices without DSP slices), but the achievable clock frequency dropped to 127 MHz. The tri-nary decoder hardware utilization was 419 slice LUT with 5 DSP slices (1052 slice LUTs with LUT multiplier) with a clock frequency of 442 MHz (369 MHz). The hardware requirements are basically dependent on size of Look-up logic which increases exponentially with

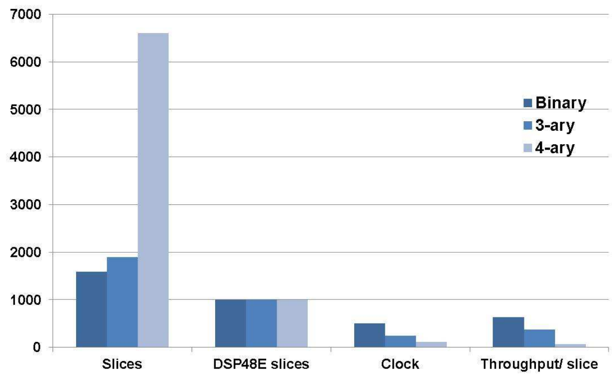


Fig. 4: N-ary arithmetic coding and encryption architectures: Comparative performance. The # of slices, # of DSP slices (x100), clock frequency (MHz) and throughput per slice (x1000) are reported in the figure. It can be observed that increasing the size of dictionary significantly reduces the throughput. The figure is drawn by scaling the throughput/slice legend to consider the fact that a 4 symbol dictionary will require half the words as a 2 symbol dictionary.

increase of N ($N|!2^N$), making it infeasible to scale-up the throughput/slice.

A simple way to restrict this bandwidth explosion is to used the algorithm for encryption proposed in [12]. They restrict the keyspace and instead use only a small fragment of keys from the entire range, for encryption. However, the approach presented in [12] has other computationally-inefficient parts.

The results are shown in Figure 4. The number of slice LUTs is reported directly, number of DSP slices is scaled directy and clock frequency is measured in MHz. The throughput comparison is tricky because using a 4-symbol dictionary (4-ary coding) will lead to reduced bitstream (around 50% reduction) than the bitstream generated by 2-symbol dictionary. Thus, to compare these values on a graph, we multiply each throughput with N value (2 for binary) to indicate relative throughput. It can be observed that increasing the size of dictionary significantly reduces the throughput, even after such considerations due to exponential increase in hardware usage for key implementation.

Although our experiment to scale to multiple-symbol dictionary failed, the reason is not the same as for traditional designs for arithmetic coding [11]. Rather, the key explosion is the main reason for such limitations. We next consider increasing the system throughput by encoding multiple binary symbols in a single pass. This approach is different than the previous approach in the sense that multiple probability values are not involved.

Multiple symbol per cycle arithmetic coding

Let us consider the case of arithmetic coding where we want to encode two symbols in a single iteration of chaotic map. In this case, the chaotic map will spit into multiple (four instead of two) piece-wise maps. Arithmetic coding with encryption is still going to suffer with band-width expansion, but we observe that the bandwidth expansion is much less (or order of 2^N) instead of $N2^N$. Consider, for example the case where we want to encode two symbols together (‘01’ instead of ‘0’ and ‘1’ in two separate iterations) using BAC. In this case, the resultant chaotic iterator will have 4 (instead of 2) piece-wise linear maps and their precision of implementation will be increased (16 instead of 8 bits). This analysis can be extended to three, four or more symbols.

In this case, the increase is caused by increase in fixed point precision of coefficients (and hence multipliers and adders), and increase in number of piece-wise maps. However, against the case of MCAC where there was a bandwidth explosion due to increase in key size, we observe a considerable different result of implementation on Virtex-6 device. These results are reported in Figure 5. The results are interesting to note, because contrasting with the traditional notion of one-symbol per cycle, we show that we can scale upto 4 symbols per cycle and achieve a higher throughput per slice. As we go from 2 to 4 case, we observe a increase in throughput which is then checked by the exponential increase in hardware resources caused by multiple symbols use. This value of 4 cannot be a device constraint (restrictions due to finite area or size of device) because the pure LUT mapping based implementation requires only 5480 slices out of 43000 slices present in target xc6vls75 device. The highest throughput achievable is 431 Kbits per slice for 4 symbols case.

For the sake of brevity, we have restricted our discussion in last sections to NCAC and multiple symbol BAC encoder, but the same trend follows for the decoder also.

5. Conclusion

In this paper, we presented architecture for simultaneous coding and encryption using chaotic maps. After presenting the hardware requirements and computations involved in chaotic maps, we mapped these designs into a Virtex-6 FPGA to obtain a performance analysis on real hardware. We investigated the key-explosion problem which avoided the implementation of simultaneous coding and encryption using larger dictionaries. However, we found that the hardware resource explosion is not much in case of multiple character coding using BAC (indicating 5 symbols be encoded simultaneously). This work is one of the earliest hardware implementation of chaotic maps, first reported implementation of chaotic maps for simultaneous coding and encryption. It achieves encryption at insignificant hardware

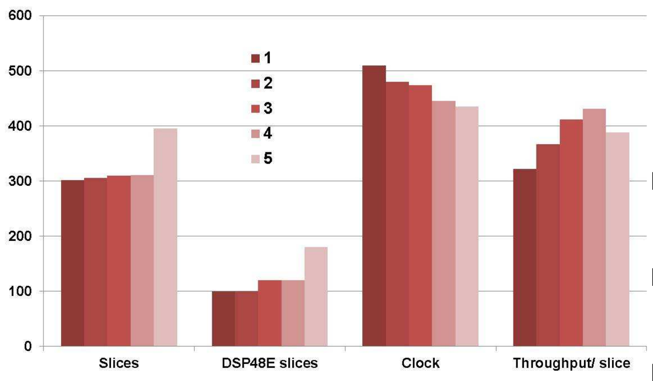


Fig. 5: Multiple symbols per cycle (BAC): Comparative performance. The # of slices, # of DSP slices (x10), clock frequency (MHz) and throughput per slice (x1000) are reported in the figure. It can be observed that 4 symbols per cycle achieve highest throughput before LUT explosion due to increased precision and maps.

cost, against use of encryption ciphers such as AES which require separate modules for encryption operation.

We are looking for, and encourage other readers also for future work in two directions:

1) Looking for ways to solve key-explosion problem using circuit level techniques.
2) Incorporating re-normalization and context to this encoder, so that it can be added to CABAC or other encoders.

Acknowledgement

This research is supported by the National Science Foundation under Grant #1019343 to the Computing Research Association for the CIFellows Project.

References

[1] H. Schwarz, D. Marpe, and T. Wiegand, “Overview of the scalable video coding extension of the H. 264/AVC standard,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 9, pp. 1103–1120, 2007.

[2] G. Langdon and J. Rissanen, “Compression of black-white images with arithmetic coding,” *IEEE Trans. Communications*, vol. 29, no. 6, pp. 858–867, Jun 1981.

[3] M. Luca, A. Serbanescu, S. Azou, and G. Burel, “A new compression method using a chaotic symbolic approach,” in *Proc. IEEE Commun. Conf.* Citeseer, 2004, pp. 3–5.

[4] N. Nagaraj, P. Vaidya, and K. Bhat, “Arithmetic coding as a non-linear dynamical system,” *Communications in Nonlinear Science and Numerical Simulation*, vol. 14, no. 4, pp. 1013–1020, 2009.

[5] M. Grangetto, E. Magli, and G. Olmo, “Multimedia selective encryption by means of randomized arithmetic coding,” *IEEE Trans. Multimedia*, vol. 8, no. 5, pp. 905–917, Oct. 2006.

[6] H. Kim, J. Wen, and J. Villasenor, “Secure arithmetic coding,” *IEEE Trans. Signal Processing*, vol. 55, no. 5, pp. 2263–2272, May 2007.

[7] A. Pande, J. Zambreno, and P. Mohapatra, “Joint video compression and encryption using arithmetic coding and chaos,” in *IEEE International Conference on Internet Multimedia Systems Architecture and Application*, 2010.

[8] R. Bose and S. Pathak, “A novel compression and encryption scheme using variable model arithmetic coding and coupled chaotic system,” *IEEE Trans. Circuits and Systems I*, vol. 53, no. 4, pp. 848–857, April 2006.

[9] G. Jakimoski and K. Subbalakshmi, “Cryptanalysis of some multimedia encryption schemes,” *IEEE Trans. Multimedia*, vol. 10, no. 3, pp. 330–338, April 2008.

[10] T. Stojanovski and L. Kocarev, “Chaos-based random number generators-part I: analysis [cryptography],” *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 48, no. 3, pp. 281–288, 2002.

[11] D. Marpe, H. Schwarz, G. Blättermann, G. Heising, and T. Wieg, “Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard,” *IEEE Trans. Circuits and Systems for Video Technology*, vol. 13, pp. 620–636, 2003.

[12] K.-W. Wong, Q. Lin, and J. Chen, “Simultaneous arithmetic coding and encryption using chaotic maps,” *IEEE Trans. Circuits and Systems*, vol. 57, pp. 146–150, February 2010. [Online]. Available: http://dx.doi.org/10.1109/TCSII.2010.2040315

[13] S. Lian, J. Sun, J. Wang, and Z. Wang, “A chaotic stream cipher and the usage in video protection,” *Chaos, Solitons & Fractals*, vol. 34, no. 3, pp. 851–859, 2007.

[14] A. Moffat, R. Neal, and I. Witten, “Arithmetic coding revisited,” *ACM Transactions on Information Systems (TOIS)*, vol. 16, no. 3, pp. 256–294, 1998.

[15] G. Cormack and R. MORSPOOL, “Algorithms for adaptive Huffman codes,” *Information Processing Letters*, vol. 18, no. 3, pp. 159–165, 1984.

[16] J. Vitter, “Design and analysis of dynamic Huffman codes,” *Journal of the ACM (JACM)*, vol. 34, no. 4, pp. 825–845, 1987.

[17] A. Moffat, N. Sharman, I. Witten, and T. Bell, “An empirical evaluation of coding methods for multi-symbol alphabets,” *Information Processing & Management*, vol. 30, no. 6, pp. 791–804, 1994.

[18] I. Witten, R. Neal, and J. Cleary, “Arithmetic coding for data compression,” *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[19] P. Howard and J. Vitter, “Analysis of arithmetic coding for data compression,” *Information Processing & Management*, vol. 28, no. 6, pp. 749–763, 1992.

[20] G. Langdon, “An introduction to arithmetic coding,” *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 135–149, 1984.

[21] R. Osorio and J. Bruguera, “Arithmetic coding architecture for H. 264/AVC CABAC compression system,” 2004.

[22] T. Chuang, Y. Chen, Y. Chen, S. Chien, and L. Chen, “Architecture Design of Fine Grain Quality Scalable Encoder with CABAC for H. 264/AVC Scalable Extension,” *Journal of Signal Processing Systems*, vol. 60, no. 3, pp. 363–375, 2010.

[23] C. Lo, S. Tsai, and M. Shieh, “Reconfigurable architecture for entropy decoding and inverse transform in H. 264,” *Consumer Electronics, IEEE Transactions on*, vol. 56, no. 3, pp. 1670–1676, 2010.

[24] T. Addabbo, M. Alioto, A. Fort, S. Rocchi, and V. Vignoli, “Low-hardware complexity prbgs based on a piecewise-linear chaotic map,” *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 53, no. 5, pp. 329 – 333, May 2006.

[25] A. Pande and J. Zambreno, “Design and hardware implementation of a chaotic encryption scheme for real-time embedded systems,” in *Signal Processing and Communications (SPCOM), 2010 International Conference on.* IEEE, 2010, pp. 1–5.

[26] C.-J. Chang, C.-W. Huang, K.-H. Chang, Y.-C. Chen, and C.-C. Hsieh, “High throughput 32-bit aes implementation in fpga,” in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, 30 2008.

SESSION

REGULAR SESSION - RECONFIGURABLE AND EVOLVABLE HARDWARE ARCHITECTURES

Chair(s)

Dr. ERIC STAHLBERG

Heterogeneous Accelerated Bioinformatics – Perspectives for Cancer Research

E. Stahlberg[1], T. Steinke[2], M. C. Smith[3], S. Chandrasekaran[4], B. Chapman[4]
[1]SAIC-Frederick, Frederick, Maryland
[2]Zuse Institute Berlin, Berlin Germany
[3]Holocombe Department of Electrical and Computer Engineering, Clemson University, Clemson, South Carolina
[4]Department of Computer Science, University of Houston, Houston, Texas

Abstract – *The demand for even higher performance in bioinformatics data analysis continues to grow rapidly as the volumes of data generated by next generation sequencing equipment soar. Traditional acceleration techniques historically used for faster bioinformatics application will individually be insufficient to meet the demand and increased analysis complexity, requiring an integrated heterogeneous accelerated computing environment. Current accelerated computing programming environments including reconfigurable computing technologies as well as multicore technologies are reviewed in the context of bioinformatics applications, highlighting opportunities and limitations of these technologies. Requirements for successful heterogeneous accelerated computing environments supporting bioinformatics applications are presented.*

Keywords: Bioinformatics, Accelerated Computing, Next Generation Sequence Analysis

1 Introduction

Bioinformatics as a term continues to evolve and defy precise definition. Very generally, bioinformatics is simply analyzing information related to biology, typically involving data about the genomes and proteins. More recently, as instrumentation has enabled deeper and more precise probes of fundamental biological processes, bioinformatics encompass not only fundamental information about genomes and proteins, it encompasses information about a growing list biological intermediate (e.g. RNA, metabolites, etc.), processes and the interactions and relationships among them. This experimentally-enabled data revolution is only adding to the computational challenges and opportunities as more and more information is added to the growing global information database.

For the purposes of this paper, a narrowed segment of the bioinformatics spectrum receives our focus and attention. Next Generation Sequencing (NGS), really a generalized term that encompasses experimental approaches utilizing massively parallel (simultaneous) sequencing, creates some very important opportunities for accelerated computing, and reconfigurable computing in particular. The popularity of these experimental approaches is driven by the relatively low cost to sequence large volumes of DNA sequences, even including entire genomes. It is this comparatively low cost and high coverage that makes NGS sequencing an important step forward in the direction of personalized medicine, while also providing a tool to probe even more deeply into fundamental biological processes for cancer. The challenge is that such systems generate enormous amounts of information, with a single sequencer able to generate a terabyte or more of data each day already in 2009 [1].

2 Next Generation Sequencing

2.1 New Demands for Sequence Analysis

Reliable analysis involving Next Generation Sequencing (NGS) sequence data requires addressing the daunting challenge of using very large numbers of small pieces of variable quality data to provide supporting evidence to infer more complex biological behavior. A very recent review of NGS software has been prepared by Bao et al, providing an in depth overview of the common applications, their algorithms and relative performance [2]. With a very large number of available applications, predominant challenges and applications of immediate interest are discussed in this paper.

Traditional sequencing was able to deliver comparatively long sequences in smaller volumes than NGS, albeit more expensively. In contrast, current NGS is characterized by the simultaneous generation of extremely large volumes of comparatively short sequences of nucleotides, generally called tags. The need for extremely large numbers of tags is a direct result of the relatively short size of the tags. Tags range in size from a few tens of nucleotides to several hundred in length. In comparison to the size of the human genome for example (around 4 billion nucleotides), each tag contains around only 0.00001 percent of the information in the genome. As a result, extremely large numbers (tens of millions) of tags are required to resolve meaningful information using this technique.

A second challenge facing NGS sequencing is a result of the experimental technique itself. Massively parallel sequencing involves probing the DNA material at or near the molecular level using reliable but not absolutely reliable chemical processes and physical probes (optical and electronic). As a result, the quality of the information in the sequence tags can be highly variable as small changes in preparation, processing and probe analysis mount in different combinations. This characteristic only serves to further increase the need for ever larger number of tags to statistically compensate for errors in individual tags.

2.2 Accelerated Computing in Bioinformatics

Accelerated computing in bioinformatics is not new. A compendium of illustrative applications of parallel computing for bioinformatics has been compiled by Zomaya in 2006[3]. Many high-performance and customized systems have been developed over time to address the computational bottlenecks in taking experimental data and translating it to biological understanding. Examples include such systems from MasPar and TimeLogic [4], each providing accelerated solutions to sequence matching and comparison. MasPar employed large numbers of parallel processes to achieve performance in terms of high-throughput for sequence matching. In contrast, TimeLogic employed FPGA technology to develop card-based products to achieve great acceleration in sequence comparison speed relative to standard CPU processors [5].

Among many areas, many of the accelerated solutions focused on improving performance of sequence comparison, most prevalently using the BLAST (Basic Local Alignment Search Tool) algorithm [6]. Other commonly employed algorithms include Smith-Waterman [7] for optimized local alignment, Needleman-Wunsch [8] for optimized global alignment, and Hidden Markov Model [9] matching. Many examples [10] [11] can be found where accelerated computing (primarily FPGA-based systems) has been used to provide significant acceleration on a variety of different systems.

Multicore parallel computing has also played an important role in providing necessary throughput for these algorithms. Whether implemented in the form of a commodity cluster or a multiprocessor, multicore server, multicore CPUs have provided an affordable alternative to custom acceleration solutions. Not surprisingly, these solutions carry the bulk of the workload where the standard algorithms are employed.

2.3 Challenges for NGS Sequence Analysis

Unfortunately, though important in their own right, these past success cases for accelerated and parallel computing are inadequate for the challenges posed in NGS analysis. Primarily driven by the limited length of the tag sequence, sheer volume of tags and complicated by the limited reliability of any given tag, the simple processing of each tag through these standard algorithms is simply unable to meet the simultaneous throughput and accuracy demands of NGS sequence analysis. A new class of algorithms and computationally demanding applications has emerged, presenting tremendous opportunity for potential acceleration across the heterogeneous acceleration landscape.

2.4 Example Applications

2.4.1 TopHat

TopHat is a popular tool for conducting analysis of sequences of messenger RNA where splice junctions are not all previously known. This program maps known reads to a reference genome, while enabling mapping even if a new splice junction may be required. A comparatively fast application, the program maps reads at a rate of 2.2 million per CPU hour. [12]

2.4.2 BWA

Burrows-Wheeler Alignment tool (BWA) is used to efficiently align short sequence reads against a single reference sequence. Using the Burrows-Wheeler transform, allowing for matches of short sequences where gaps and mismatches can occur between read and target sequence. The program reports performance improvements of tenfold relative to earlier generation hash-table based methods. [13] This application is available as an open source application.

2.5 Accelerated Computing Options for NGS

Current systems are moving towards a heterogeneous architecture with a combination of traditional processors and accelerators. This paradigm shift will prove to be vital to provide great performance improvements for bioinformatics workload. This shift will also be essential to meet the power demands while solving the increased data analysis complexities, fast growth of biological databases among several other complexities prevalent in the bioinformatics domain. There are many popular algorithms such as Needleman-Wunsch [8], ClustalW [14] that are computationally intensive and it would be ideal to map them on a platform that supports both thread-level and data-level parallelism.

2.5.1 FPGA-based Application Environments

For the acceleration of bioinformatics applications with reconfigurable FPGA devices, various products past and present can be classified according their principal architecture of integrating the reconfigurable device into the system and the memory model presented to the application developer. The most common hardware integration is similar to GPGPU platforms where a FPGA-based module is connected via a PCIe link to a host (GiDEL, Nallatech). A more advance integration is realized through in-socket solutions where the connection is made via a CPU socket (Convey, XtremeData, DRC), or proprietary interconnects (SRC H Map, SGI Numalink, Cray Rapid Array Interconnect). In the advanced integration architectures, a memory model with a logical common address space and subsequently a comfortable programming environment can be implemented whereas a similar programming environment in PCIe based accelerator platforms is not known.

FPGA acceleration implies certain aspects that are unique for the reconfigurable computing sector and are not represented in manycore or GPGPU computing environments. FPGA-based computing requires, from the beginning, a sufficient understanding of the underlying hardware technology whereas working on platforms with standard CPUs more ignorance can be tolerated due to the optimization capabilities of available compilers. Nevertheless, there is progress to make FPGAs like software programmable devices for application developers by using high-level languages.

Reconfigurable computing with FPGA-based platforms offer unique features for bioinformatics applications compared to other processing devices:

- The processing resources for logical and mathematical operations can be adapted to accuracy requirements of algorithms and operands. Any bit-width representation can be used, not only the IEEE-conform set of data types supported by high-level languages and standard CPUs (32 or 64 bit wide representations[1]). This improves resource utilization and enables energy efficiency.

- Bandwidths to the internal memory of 1.3 TiB/s up to 2.5 TiB/s are implemented [15], .e.g. compared to 45 GiB/s to local cache on Nehalem CPU. External theoretical memory bandwidth to DDR3 type memory is substantially lower whereas the total specified transceiver bandwidth is an order of magnitude higher than for standard CPUs[2].

- Due to the low clock rate the power consumption of a FPGA device itself is lower compared to GPU and standard CPU devices.

FPGA platforms become popular in bioinformatics as the number of logical building-blocks increases to a level that computationally demanding functional kernels could be implemented on such a device. Probably the most famous example is the Smith-Watermann algorithm to perform a local sequence alignment. Recently, Convey published performance results of their Smith-Waterman implementation on Convey HC-1 and HC-1ex platforms. For a non-disclosed test case, a performance of 1603 GCUPS[3] (HC-1ex) and 688 GCUPS (HC-1) are reported, respectively, and this translates to a performance improvement compared to a SSE2 implementation on 8 core Nehalem of 50x and 20x [17].

Traditionally, implementations for an FPGA are made by a circuit design approach which is common with the design of ASIC or micro-controllers. This approach and consequential naming conventions and tool chains are developed for the domain of hardware design specialists.

With the availability of increasingly larger FPGA devices, the demand for FPGA program development tools for non-hardware specialists becomes a strategic question. Today, a number of so called "C-to-HDL" tools are available, both commercially as well as non-commercially, with a different range of supported FPGA platforms.

In this paper we take the view of an application programmer and focus our discussion on the software approach of using FPGAs for bioinformatics application acceleration. Specifically, we review three approaches offered by tool and platform vendors differing in their objectives and their required level of expertise. The first approach involves the "C-to-HDL" route to implement an algorithm on a FPGA device. In the second approach, the power of reconfigurable computing is made accessible through advanced compiler interfaces. The third approach uses customized software libraries to expose HDL designs developed by circuit-design experts at the software level and to hide any hardware dependencies from the application developer.

High-Level Language Approach

Easy programmability to achieve productivity is the key factor to adopt any processor technology. This holds in particular for the FPGA accelerator technology and can be accomplished by using high-level language tools. Over the last years, progress is made to improve robustness, performance and portability although not unsurprisingly at the same level as expected as known from compiler suites for standard CPUs.

There are several commercial and research based tool chains available to ease the programming process of mapping applications to accelerators. For example some of the commercial solutions include Impulse-C from Impulse Accelerated Technologies [18], C2H, from Altera [19] Forte Cynthesizer [20], PICO Express from Synfora [21] Mitrion-C from Mitrionics [22], AutoESL from Xilinx [23] BlueSpec Compiler from BlueSpec [19], CARTE from SRC Computers [25].

Some of the research-based toolsets include DEFACTO [34], SPARK [27], ROCCC [28] and DRESC [29]. Discussing few

[1] The C ISO standard 1999 supports bit-fields in structures but is among other drawbacks that are not portable

[2] For Xilinx Virtex 5 FPGA 48 GiB/s are specified as theoretical bandwidth to DDR3 SDRAM [1]

[3] Giga Cell Updates Per Second, performance of computing the matrix in the dynamics programming algorithm

of these solutions, (DEFACTO) [34] combines parallelizing technology with commercially available HDL tools. The challenge is to understand how much and what kind of transformations must be applied and the metrics that would be suitable to evaluate the design. The Riverside Optimizing Compiler for Configurable Computing (ROCCC) [28] is a C-based compiler that generates RTL-VHDL for implementing algorithms on FPGAs. This approach requires lot of effort to write the code in subsets of C. Many features of C are not supported such as pointers and conditional expressions. Moreover ROCCC supports specific kernels. SPARK is another C to VHDL compiler. It is particularly targeted to multimedia and image processing applications. SPARK requires a good hardware knowledge and considerable designer input, and does not always generate synthesizable VHDL. The delft workbench automated reconfigurable VHDL (DWARV) Compiler [30] transforms C functions identified with pragmas to VHDL designs. DWARV supports certain subsets of C. They support for- and if- statements. But the restrictions are not as stringent as that of SPARK and ROCCC. The dynamically reconfigurable embedded system compiler (DRESC) [29] is a re-targetable compiler for coarse-grained reconfigurable architectures relying on the IMPACT compiler framework [31] to perform other optimizations and scheduling. Graphs from the IMPACT compiler are complicated to analyze at the assembly level, so as to determine the dependencies and compute intensive portions of the code.

FPGA Acceleration through Extended Instruction Sets

With the HC-1, a hybrid-core architecture entered the market [32]. From the application developer point of view, we see three important features in this architecture:

I. The system provides a virtual shared, cache-coherent address space across the x86 and the FPGA-based accelerator subsystem,
II. The reconfigurable subsystem implements extensions of the x86 ISA (called personalities), and the vendor compiler suite will take advantage of it.
III. The system can be configured with a large memory locally to the FPGA accelerator subsystem in which data access is managed by a proprietary memory-controller supporting parallel data paths from each of the four FPGA-based accelerators to each of 16 memory banks.

Data placement and transfer between the host and accelerator memory are managed by the runtime system or through compiler directives. Convey provides a mechanism to integrate any new customer implemented FPGA-based instructions or functional kernels into the Convey compiler suite and runtime API. This means, that these developed personalities are easily accessible for application developers by using the Convey compiler.

Domain Specific Application Libraries

Despite the progress with high-level language tool chains still the highest performance on FPGA is achieved with circuit designs using a hardware description language. Usually, the required expertise for circuit design is outside the domain of scientific application developers. Therefore, providing a software library interface to kernel functions implemented on FPGA enables application scientists to take advantage of FPGA accelerators within software packages. For example, interfaces to Smith-Waterman implementations were provided Cray XD1, SGI RC100, and Convey HC-1. Opportunities still exist for validated libraries using common interfaces to contribute significantly.

Tool Chain Limitations

However there are several challenges using these toolsets.

- These toolsets appear to be not mature enough to bridge the gap between the device computing resources and programmer's conceptual model.
- The toolsets are not able to recognize the needs of the application and the capabilities of the device.
- The toolsets do not provide automatic solutions to map applications to hardware.
- The steep learning curve necessary to be able to use the tool chains efficiently is quite high even for the list of solutions mentioned above.
- Most of the toolsets use subsets of C as their input. C being an imperative/sequential language does not support the concept of clocks that is a significant hardware feature for devices like FPGAs.

With these drawbacks, we see that a lot of time and effort is spent in tapping the potentials of FPGAs.

Several limitations still must be addressed to fully impact next generation bioinformatics applications. These include:

- **HLL Portability:** Portability is not yet realized at different levels of the software development.
  Missing standardized high-level languages prevent portability, market penetration and potentially feigned bounds to vendors. Limited portability is given only if the tool chain is neutral across different system vendor platforms. Furthermore, there are no community standards for library interfaces to general algorithms implemented although such efforts were made by e.g. OpenFPGA.
- **Performance:** Performance-wise, native HDL designs are more attractive compared to C-to-HDL implementations. Further research is required to decrease this gap.
- **Financial barriers:** With a few exceptions, the business models for tool chains are oriented towards the commercial high-volume market. Compared to the license model of the programming tools for GPU or multicore

CPU - being either free or moderately priced - this is a serious barrier in particular in the academic environment.

2.5.2 Multicore Application Environments

Heterogeneous multicore systems are often comprised of specialized devices that have their own functionalities and instruction sets along with the general-purpose cores. A set of tools and programming models is necessary in order to efficiently use the computational capabilities of these multicore systems. A uniform and general-purpose programming model that enables the user to manage the low-level interfaces of different devices, mostly embedded, and that facilitates application programming on several types of embedded devices, would be ideal.

Recently many hardware and software companies formed the Multicore Association (MCA) [33] to solve the challenges persistent in the heterogeneous multicore environment. MCA is providing standards/APIs for inter-process communication (MCAPI) and resource management (MRAPI). It is also developing an API for task management (MTAPI). Although intended to facilitate portability, these are low-level APIs that could be tedious for application programmers. As a result, it would be worthwhile to consider a high-level programming model such as OpenMP [35] that can shield the user from the low-level complexities arising from heterogeneous systems. The OpenMP model has been implemented in many commercial and industrial compilers; more details can be found in [34]. Extensions are being considered to extend OpenMP for accelerators and there are already a few existing solutions that demonstrate the productivity and programmability metrics obtained by using OpenMP in this context. A current goal is to explore the use of OpenMP as a high-level programming model for application development on embedded systems by:

- Proposing appropriate extensions to the OpenMP API to support heterogeneous (accelerator) cores.
- Using the MCA libraries as an underlying API to make the OpenMP runtime library portable across systems supporting MCA routines.

2.5.3 GPGPU Application Environments

On General Purpose GPU platforms, algorithms designated for the SIMD programming model can be implemented efficiently. GPU platforms represent a co-processor architecture. High-end GPU devices with local memory are connected via PCIe to the host. The latest GPU divides support both single and double precision floating-point operations as known from multicore CPUs.

Within the co-processor model, distinct address spaces are realized. The application programmer must deal with an extended memory hierarchy with memory on the host side in addition to global device memory, SMT shared memory and thread local memory on the GPU device. Global device memory is limited and at most hardware configurations with 6 GB are known.

After the adoption of the first version of the OpenCL [36] standard in 2009, it is now implemented in programming development tools for GPUs from both vendors in the HPC market. Additionally, a large developer base evolved around the CUDA [37] programming environment with compiler and runtime stack specifically for Nvidia GPUs.

Data parallel algorithms are well suited for GPU implementations. To achieve optimal performance the thread topology and memory management must be carefully designed, which is of limited portability across device generations.

As reference, benchmark numbers of 17 and 30 GCUPS on GTX 280 and GTX 295 (dual GPU card) for a Smith-Waterman implementation are reported, respectively [37]. More recently, four-fold [39] and ten-fold [40] acceleration was reported for BLAST acceleration using graphics processors.

2.6 Requirements for NGS Applications

In a research environment, reproducibility of analysis results is of paramount concern. As a result, applications used in the research environment must be reliable and robust across multiple platforms as well as over time. Consequently, for heterogeneous accelerated computing to impact NGS applications, portability and reliability must be met while still achieving improvements in performance. This requirement is further emphasized when one examines the fact that many of the applications are open source, and therefore must maintain portability not only within a given lab, but industry-wide.

Standards are an important element of portability, and standards such as OpenMP and OpenCL provide a great degree of support for portability when employing multicore processors and graphics processors. Unfortunately, standards for reconfigurable computing platforms remain elusive despite efforts such as those of OpenFPGA [41].

Validation and verifiability of applications and central algorithms are also important elements for applications in a production research environment. Curated and maintained test suites used to confirm portability of a given application and application component are essential to assure portability is preserved even as implementations and runtime environments change.

Runtime environments will also be quite variable, particularly as the sheer size of the data involved imparts evaluation of new conditions, such as moving the application closer to the data rather than moving the data to the application. Optimization of data movement in the environment is an emerging major concern in evaluating next generation sequence data.

2.7 Accelerated Heterogeneous Computing: Application Environment Requirements

2.7.1 FPGA Requirements

The current state-of-the-art in FPGA-based acceleration is that the custom circuits must be carefully hand crafted, with significant manual input to identify the parallelism and the processing precision, in order to achieve the best speedup possible. As discussed previously, there is still a dearth of tools that are able to deliver VHDL/Verilog for FPGA-based devices automatically from a high-level language without manual intervention. Moreover, if FPGAs are being considered as potential devices to address heterogeneity, these devices must be adapted to a common programming model that supports both traditional processors as well as accelerators. This is currently one of the major requirements that FPGAs need to attend to.

Other requirements include:

- Providing advanced compilation and optimization features to obtain the best performance.
- Providing efficient scheduling strategies to best exploit the fine grained parallelism of FPGAs.
- Accelerating design cycles to provide more than two orders of magnitude.
- Providing debugging and verification tools.
- Ensuring portability of the code generated by the tools to the platforms considered.
- Adoption of standards for HLL but keep the flexibility of bit widths.
- Standards for runtime SW stack (device management, see OpenCL for GPU).
- Logical cache-coherent single address space.
- Lower investment costs for hardware and tools, target metric should reflect overall performance improvement and power reduction.
- Robust environments for software simulation at functional level and additionally as option at cycle level to enable easy integration of custom designs made by experts into programming tools, e.g. compiler.
- Faster Prototyping: Desired innovation cycle at same speed as for standard CPU environments

2.7.2 Multicore Requirements

Software development and tools for multicore systems are still in their infancy and need considerable work. While existing tools may be efficient enough to exploit a small number of cores, they are unlikely to be able to efficiently exploit large numbers of cores. For instance, cores are likely to double every 18 months and by 2017 embedded processors could support 4,096 cores, server CPUs might have 512 cores and desktop systems could have 128 cores. It is unlikely that existing tools will be able to scale, and instead, new tools and programming languages will need to be developed to fully exploit the hundreds or thousands of processors. Currently, applications are not designed to take advantage of the multicore systems since most software is still written sequentially, and needs to be rewritten using parallel-programming languages in order to efficiently exploit these new multicore systems.

Other requirements include providing:

- Effective algorithmic choices and good selection of data structures to support multi-core systems.
- Effective higher level of abstraction while using programming techniques.
- Effective software support to provide advanced compilation features/techniques.
- Effective programming model to support/integrate different accelerators on a single platform.
- Effective optimization features for good programmability/productivity.
- Effective automatic mapping techniques to map application to specific underlying modules (CPUs, GPUs)
- Effective performance-aware component model that will not only include FLOPs related improvements, but also power and memory characteristics.
- Effective scheduling mechanisms to schedule work and provide good data management strategies on heterogeneous architectures. These mechanisms include effective work-stealing strategies, data-aware scheduling techniques, and static/dynamic scheduling techniques.
- Effective feedback mechanisms to provide feedback about the runtime environments to higher-level components.
- Effecting power modeling strategies to limit the excessive usage of power. Since more and more cores when fabricated on a single piece of silicon, power consumption can eventually be a major concern.
- Effective debugging tools to be able to debug and verify the program and reducing the time and effort spent on producing the correct result.

2.7.3 Cross-technology Requirements

Independent of accelerator technology, several requirements are needed in production heterogeneous computing environments.

- Tighter integration of heterogeneous devices into system, i.e. devices should be equally managed by OS as standard CPU today. This allows scheduling of application tasks on heterogeneous functional units by the (extended) OS thread/process scheduler.
- Unified (standardized) either tool-assisted and/or automatic software – hardware partitioning; today we have diversity and mostly vendor-centric tools, e.g. SRC CARTE (FPGA), CAPS hmpp (GPU), PGI Accelerator compiler (GPU), Convey compiler (FPGA); Will OpenMP 4 solve

the missing-standard problem without substantial performance penalties?

- Robust and user friendly software tool chains which support parallelization / express parallel algorithms for O(1000) cores (CPU/GPU) or functional units (FPGA) per node.

3 Future Directions

Heterogeneous computing provides a tremendous opportunity for accelerating next generation bioinformatics data analysis applications. Fortunately, many of the requirements outlined in this paper are well aligned with objectives for extreme-scale computing. Future directions to advance sustained use of heterogeneous acceleration in bioinformatics applications will both inform and benefit from current efforts to define programming and run-time environments for extreme scale computing. Efforts are underway to fully integrate the proven strengths of each acceleration technology in bioinformatics applications, providing a promising future for heterogeneous accelerated computing in cancer research.

References

[1] S Tracy, "Next Generation Sequencing: Elements for Success", Scientific Computing, May/June 2009.

[2] S Bao, R Jiang, W Kwan, B Wang, X Ma, YQ Song, "Evaluation of next-generation sequencing software in mapping and assembly", J Human Genet, Apr 28, 2011

[3] A Zomaya (editor),"Parallel Computing for Bioinformatics and Computational Biology", Wiley and Sons, 2006

[4] http://www.timelogic.com/decypher_intro.html

[5] http://www.timelogic.com/fpga_performance.html

[6] http://blast.ncbi.nlm.nih.gov/Blast.cgi

[7] T.F. Smith, S.M. Waterman, "Identification of Common Molecular Subsequences". Journal of Molecular Biology 147: 195–197, 1981

[8] B.S. Needleman, D. C Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology 48 (3): 443–53, 1970

[9] L. R. Rabiner and B. H. Juang, `An introduction to hidden Markov models," IEEE ASSP Mag., pp 4--16, Jun. 1986.

[10] P. May, G. Klau, M. Bauer, and T. Steinke, "Accelerated microRNA-Precursor Detection Using the Smith-Waterman Algorithm on FPGAs," In Proceedings of GCCB 2006, LNBI, vol. 4360, pp. 19-32, 2007.

[11] O. Storaasli, W. Yu, D. Strenski, and J. Maltby, "Performance Evaluation of FPGA-Based Biological Applications," Cray Users Group Proceedings, 2007

[12] C Trapnell, L Pachter, and S Salzberg, "TopHat: discovering splice junctions with RNA-Seq", Bioinformatics, 2009, 25(9), 1105-1111

[13] H Li, R Durbi, "Fast and accurate short read alignment with Burrows-Wheeler transform", Bioinformatics, 2009, Jul 15; 25(14); 1754-60

[14] J.D. Thompson, D.G.,Higgins, T. J,, Gibson, "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice". Nucleic Acids Res 22 (22): 4673–4680, 1994

[15] J. Williams; RC Device Characterizations & Tradeoff Analysis, 30.08.2007, www.gstitt.ece.ufl.edu/courses/fall07/eel4930_5934/RC_lecture_F5.ppt

[16] A. Cosoroaba, F. Rivoallon; Achieving Higher System Performance with the Virtex-5 Family of FPGAs, www.xilinx.com/support/documentation/white_papers/wp245.pdf

[17] Smith-Waterman Performance, Convey Computer Corp., http://www.conveycomputers.com/performance.html (accessed 05.05.2011)

[18] Impulse Accelerated Technologies, http://www.impulseaccelerated.com/

[19] C2H, http://www.altera.com/devices/processor/nios2/tools/c2h/ni2-c2h.html

[20] Fort, http://www.forteds.com/products/index.asp

[21] SynF, http://www.synopsys.com/Community/Interoperability/SystemLevelCatalyst/Pages/MSynfora.aspx

[22] Mitrionics, http://www.mitrionics.com/?page=algorithm-development-for-fpgas

[23] AutoESL, http://www.xilinx.com/tools/autoesl.htm

[24] BlueSpec, http://www.bluespec.com/

[25] CARTE, http://www.srccomp.com/techpubs/carte.asp

[26] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, J. M. Hall, R. Jain, R. H. Ziegler, "DEFACTO: A design environment for adaptive computing technology", Journal of Parallel and Distributed Processing, pp 570-578, 1999

[27] SPARK, A parallelizing approach to the high-level synthesis of digital circuits. [Online]. Available: http://mesl.ucsd.edu/spark/

[28] Z. Guo, W. Najjar, "A compiler intermediate representation for reconfigurable fabrics", FPL 2006, pp 1-4, 2006

[29] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures", FPT 2002, pp 166-173, 2002

[30] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Liu S. Vassiliadis,, "DWARV: Delftworkbench automated reconfigurable VHDL generator", FPL 2007, pp. 697-701, 2007

[31] P. P. Chang, S . A. Mahlke, W. Y. Chen, N. J . Waner, and W. W. Hwu. "IMPACT A n architectural framework for multiple-instruction-issue processors," ISCA 1991, pp 266 – 275, 1991

[32] Convey Computer Corporation, http://www.conveycomputer.com/

[33] http://www.multicore-association.org/home.php

[34] OpenMP, http://openmp.org/wp/

[35] B. Chapman, G. Jost, and R. Van Der Pas. Using OpenMP: portable shared memory parallel programming, volume 10. The MIT Press, 2007.

[36] http://www.khronos.org/opencl/

[37] http://www.nvidia.com/object/cuda_home_new.html

[38] Y. Liu, B. Schmidt, D. L Maskell, "CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions", BMC Research Notes 2010, 3:93, doi:10.1186/1756-0500-3-93

[39] P. Voizis, N Sahandis, "GPU-BLAST: using graphics processors to accelerate sequence alignment", Bioinformatics, 2011, Jan 15; 27(2); 182-8

[40] W.Liu, B Schmidt, W. Muller-Wittig,"CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware", IEEE/ACM Trans Comput Biol Bioinform, Feb 15, 2011

[41] www.openfpga.org

Reconfigurable and Evolvable Architectures and their role in Designing Computational Systems

Andy M. Tyrrell

Abstract— **Biological inspiration in the design of computational systems has been specified in many ways, one continent way is to broadly consider three biological models: phylogenesis, evolution of species, ontogenesis, the development of an individual as directed by the genetic code, and epigenesis, the learning processes influenced both by their genetic code and by the environment. These models can be considered to share a common basis: the genome a one-dimensional description of the organism. If one would like to implement some or all of these ideas in hardware what do we need from specifically designed-for-purpose devices? What design structures might help us achieve our goals? This paper considers a novel device designed and built specifically for bio-inspired work. It also considers some of the novel mechanisms that might assist with bio-inspired designs and finally considers a possible novel processing medium.**

I. Introduction

Reconfigurable and evolvable architectures are often considered under the field of evolvable hardware. The term Evolvable Hardware (EH) refers to hardware (usually electronic hardware) that can change its configuration (and thus function) using artificial evolution, that is, approaches such as genetic algorithms, genetic programming [e.g. 1] or evolution strategies [e.g. 2, 3]. Evolvable hardware has been a topic of considerable academic research over the past 15 or so years. In the mid 1990's researchers began applying evolutionary algorithm to dynamically alter the functionality and physical connections of circuitry. This combination of evolutionary algorithms with programmable electronics devices such as field programmable gate arrays (FPGAs), field programmable analogue arrays (FPAAs) or field programmable transistor arrays (FPTAs) spawned evolvable hardware a new field of evolutionary computation [e.g. 4]. Since that time the EH field has expanded beyond the use of evolutionary algorithms on simple electronic devices to encompass many different combinations of evolutionary algorithms and biologically inspired algorithms with various physical devices (or simulations of physical devices) [e.g. 2, 5]. Evolvable hardware has been used in many areas for creative design and has proved particularly popular for the creation of electronic circuits. While successful on the surface, the actual designs struggle to match those produced by more traditional methods [2]. However, in some of these cases, where a particular feature is being considered (for example, reducing cell susceptibility to device variability) evolvable hardware has proven successful. Alternatively, one of the more interesting features of evolvable hardware is a capability to perform online adaptation on existing systems [2], that is, to produce adaptable systems. This feature has shown some promising results for online fault tolerance, for example.

When considering evolvable hardware there are a number of fundamental questions that must be answered before taking that particular path:

- What hardware platform will be used? Should it be completely on the final hardware, should it be only in simulation, should it be some combination of both of these? Should it be on custom hardware or off the shelf hardware? The answers to these questions will of course depend on the application, the ultimate goals for using evolvable hardware and sometimes just simply the availability of resources.
- What evolutionary algorithms will be best suited for the evolution? In addition to the "normal" considerations related to the use of evolutionary algorithms, within evolvable hardware one must consider resource constraints. If executing the evolution on the hardware there are not unlimited memory resources, sorting is not always that straight forward, hence crossover and large populations are rather more difficult to implement than on a PC.

Present research in the field of evolvable hardware can be split into the two distinct areas: original design and adaptive hardware. In original design evolutionary algorithms and biologically inspired algorithms are used to create physical devices and/or designs. Some successful examples include analogue and digital electronics, antennas [e.g. 6, 7], MEMS chips [e.g. 8], optical systems [e.g. 9] and quantum circuits [e.g. 10]. With adaptive hardware evolutionary algorithms and biologically inspired algorithms are used to endow physical systems with some adaptive characteristics. These adaptive characteristics are required to construct more robust components and systems to allow them to continue to operate successfully in a changing environment. For example, a circuit on an FPGA that "evolves" to heal damage resulting from faults in the system [e.g. 11].

This paper considers how such architectures, using various metaphors from biology, can be used to help design computational systems. The ultimate goal of much of this research is to enhance current designs with attractive biological characteristics, such as better reliability and adaptability.

A number of examples will be presented, some issues

The author is with the Department of Electronics at the University of York, York, UK (e-mail: amt@ohm.york.ac.uk).

related to the performance of such systems will be highlighted and some suggestions for future research given.

II. Hardware Platforms

Much of the early work in evolvable hardware (indeed much of the work to-date) used off-the-self hardware, usually Field Programmable Gate Arrays (FPGA) [e.g. 12, 13, 14]. While this is a viable option and some successful results have been achieved there are some problems with this method, not least of which is the ability to reconfigure small parts of the hardware fabric. One method to try and improve the reconfigurable nature of the hardware platform is to design dedicated hardware. A number of "low-level" devices have been produced, most notably the JPL field programmable transistor array [e.g. 15] and the field programmable transistor array designed by the group in Heildelberg [e.g. 16]. These focus on reconfiguring that hardware, as the names suggest, at the individual transistor level. What will be described here in more detail is a device that has a structure somewhat similar to a traditional FPGA, but had a number of extra features integrated into its design to assist reconfiguration and evolution.

The RISA device is a FPGA hardware device [17, 18]. Figures 1, 2 and 3 show high-level and low-level details of this device designed specifically for evolvable applications.

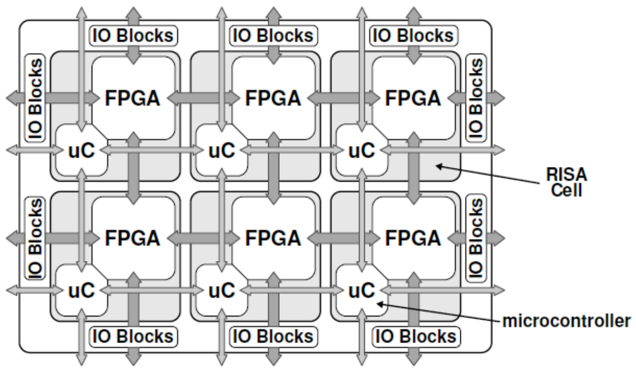


Figure 1. The RISA Architecture comprises an array of RISA Cells. Each cell contains a microcontroller and a section of FPGA fabric. Input/Output (IO) Blocks provide interfaces between FPGA sections at device boundaries. Inter-cell communication is provided by dedicated links between microcontrollers and FPGA fabrics [17].

The following features highlight the devices benefits over commercial devices when considering reconfiguration and particularly evolution:

- A fine-grained partial reconfiguration system. Bit-stream loading occurs without disrupting circuit operation and subsequent reconfiguration occurs in a single clock pulse.
- The architecture's multiplexer based configurable fabric cannot be configured into a contentious state. Therefore, it is possible to use random bit-stream without risk of device damage.
- Each RISA cell's microcontroller can be used to perform intrinsic reconfiguration of the FPGA fabric.
- Each microcontroller has a dedicated full-duplex, hardware flow controlled communication link with its four nearest neighbours.

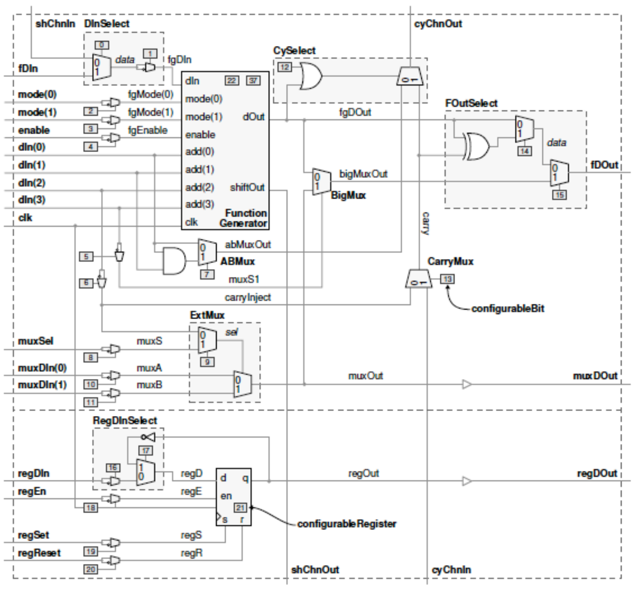


Figure 2. The RISA Function Unit is the lowest level of the FPGA fabric structure [17].

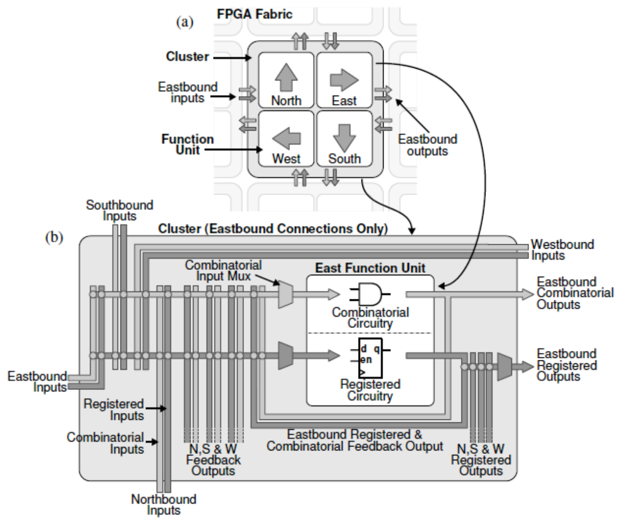


Figure 3. The multiplexer based FPGA routing design can be randomly configured without risk of forming combinatorial feedback paths or signal contentions. Combinatorial paths may only be connected within their assigned directions. Registered paths can connect to all signal directions [18].

The FPGA fabric provides hardware reconfigurable elements for the RISA architecture. The details of the Functional Unit are shown in Figure 2. The FPGA architecture does not attempt to compete with commercial devices in terms of density and flexibility of circuit implementation but aims to provide a more appropriate configuration system for bio-inspired systems. In particular, as mentioned earlier, by offering fine-grained partial reconfiguration and is designed to be random bit-stream safe, allowing unconstrained evolution to take place without the danger of destroying the device.

The main configurable component of the gate array is the Cluster. Each cluster contains logic circuitry and the routing for interconnecting internal circuitry and clusters. The Function Unit is the main configurable logic element. There are four Function Units in each Cluster. Figure 3 illustrates a cluster and connectivity within a cluster.

The circuitry of the Function Unit contains three core elements, a Function Generator, a 2-1 multiplexer a single D-type flip-flop. These may be used individually, but can be combined to provide extra functionality such as the following configurations:

- 4 input, 1 output Look-Up Table (LUT)
- 16x1 bit RAM
- 1 to 16 bit variable length Shift register block (extendible via a dedicated shift chain to other clusters)
- 4-1 multiplexer
- 1 bit full adder with fast carry chains for expansion into other clusters

Figure 3 also illustrates the connections for a Cluster's East Function Unit. Any input can be connected to a registered logic function, whereas the purely combinatorial circuitry can only accept inputs from eastbound combinatorial signals and registered signals.

The RISA architecture offers a new reconfigurable device for investigating bio-inspired systems. The architecture's design offers end-users maximum flexibility in device operation. The configuration system is simple, quick and provides a level of control not found in commercial FPGA devices. The integration of a microcontroller array adds a new dimension to reconfigurable architectures, providing a distributed reconfigurable software element.

A variety of evolutionary design techniques can be undertaken using the RISA architecture. Intrinsic evolution of electronic circuits can be performed using the FPGA fabric for implementing candidate solutions and the microcontroller to drive the evolutionary algorithm. What is more, the evolutionary process can be accelerated by performing multiple evaluations of candidate solutions in parallel. This is achieved by using the integrated microcontroller network to transfer fitness information.

The next section considers the use of biological-inspired developmental methods for designing complex systems. However, when considering this the implementation onto hardware, and specifically onto a RISA array, was always at the forefront of decisions made.

III. Artificial Development

According to Wolpert in his book Principles of Development [19]: "The development of multicellular organisms from a single cell – the fertilised egg – is a brilliant triumph of evolution". This short phrase expresses one of the fundamentals of natural development, namely the fact that a single cell may arguably be the most complex part of any organism, but a single (or few) cell organism is vastly limited in the tasks it can achieve. The evolution from single-celled to multi-celled organisms appears to have been the significant step in the development of complex structures. A question that has been asked by researchers in the field of bio-inspired architectures is can artificial developmental systems (ADSs) be useful in the design of complex computational architectures [e.g. 20, 21, 22]?

Given the evidence from biological systems it would appear that multi-cellularity is one of the identified key features required to achieve scalability and fault tolerance in systems, and at the same time sufficient complexity of cells needs to be ensured in order to achieve adaptivity, specialisation and the ability to perform all functions required to develop and maintain an organism. Therefore, compact cell programs are desired that achieve a high degree of functionality while providing a small resource footprint (that is to be implemented on a small processing architecture – in this case RISA). A number of researchers have considered the mapping of biological development to an engineering context. One way this can be achieved is by introducing a number of genetic representations to achieve structures similar to that of Genetic Regularity Network (GRNs) would in biological organisms. These GRNs feature small resource footprints and can be processed quickly due to their small size, while providing efficient encodings for evolutionary optimisation that exhibit a high degree of evolvability. The hypotheses of the work reported in [23] are:

- Variable length GRN representations have no drawbacks in terms of complexity, evolvability and success rate when compared to fixed length ones tested on the task presented.
- Variable length GRN representations feature increased compactness, require less computational effort during optimisation and running development, hence, provide shorter time-to-solution.
- The small size makes them more viable for resource critical applications like robotics and embedded systems. It also allows for applications where the target is to encode information in a compact fashion.

The artificial developmental system illustrated here is based on a gene regulatory network and focused on an implementation on hardware, the RISA architecture to be exact.

The core of the proposed developmental model is represented by a GRN, which is executed as shown in Figure 4. genomeADS is implemented as a string of characters that represent binding sites and gene actions as well as separators between genes and pre-/post-conditions. GenesADS consist of pre- and post-condition, which both comprise a number of binding sites. Each binding site has three integer parameters attached, which represent different things in different contexts: in the pre-conditional case they represent the activation threshold, the binding site type (excitatory, inhibitory) and the protein consumption rate. Note that

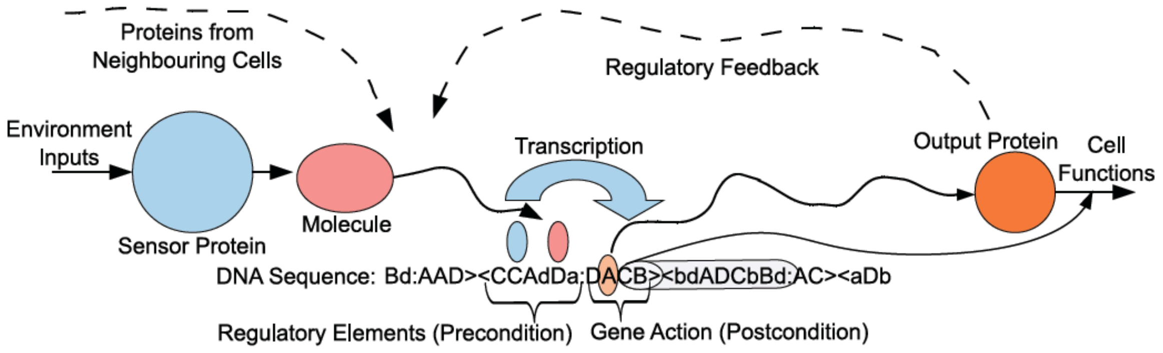

Proteins from Neighbouring Cells
Regulatory Feedback
Environment Inputs
Sensor Protein
Molecule
Transcription
Output Protein
Cell Functions
DNA Sequence: Bd:AAD><CCAdDa:DACB><bdADCbBd:AC><aDb
Regulatory Elements (Precondition)
Gene Action (Postcondition)

Figure 4. The genes in an artificial DNA sequence are activated when a sufficient amount of protein is produced by the GRN, which then causes transcription of the respective gene. In the example shown in the figure, the activity of the GRN results in the correct transcription factors (the promoter proteins) that bind to the sites of a gene sequence. This initiates the production of another protein, which can then affect cell functionality using further information that is encoded in the gene. The protein produced also provides feedback to the GRN, which regulates the transcription of other genes, thus achieving dynamic gene regulation. A gene is expressed or inhibited according to the result of the evaluation of the precondition [23].

proteins are only consumed when a binding site is activated, i.e. the concentration of the matching chemical is above the threshold. In the post-conditional case, the parameters represent the production rate of the respective protein and provide two integer parameters p1 and p2 as inputs to the associated protein function.

Wolpert [19] describes three different types of cell signaling (cell communication): protein diffusion, direct contact of complementary proteins on the cell's surface, and gap junctions (equivalent of Plasmodesmata in plants). Two different types of cell signaling are realised in the proposed model. Protein diffusion has been implemented in a similar way as it takes place in physical systems, e.g. a drop of ink dissolving in water. The second cell signaling mechanism is an implementation of Plasmodesmata in plants (protein tunnels), which effectively are tunnels between cells that allow chemical sharing. Cell growth is also based on the Plasmodesmata mechanism; if the neighbour cell, which is targeted by the tunnel, is an empty cell space, a new cell will grow into that empty cell space and become alive (i.e. will be processed by the ADS) in the next developmental step (see Figure 5). Cell death is not implemented at the current stage.

Protein levels are credited and debited after the GRN has processed the next developmental step for all cells [23]. Thus, the GRN always works with the original protein levels and the order of cell update should therefore not bias the behaviour of development or certain regions within the organism. Diffusion is a long range signaling mechanism that should help to create and maintain symmetries within the system.

Once this model was developed fully it has been used to explore the application of an ADS to the field of evolutionary robotics by investigating the capability of a GRN to control an e-puck robot. A GRN controller has been successfully evolved that exhibits a general ability to avoid obstacles in different maps (Figure 6) as well as when transferred to a real robot (Figure 7). It has been shown that GRN based controllers have the potential to adapt to different environments, due to the fact that the robot successfully managed to navigate through previously unknown maps and could be successfully transferred to a real robot without further modification of the controller. Hence, it is concluded that GRNs are a suitable approach for real-time robot control and can cope with variations inferred by changing environments and sensor noise of a real robot. The results further suggest that it is possible to specify a general purpose obstacle avoidance behaviour via a GRN.

While a relatively simple example it does show that ideas based on developmental biology can be mapped to the field of engineering with success. More needs to be done here, but artificial developmental systems still appear to be a good avenue to pursue if we are to evolve large complex engineering systems.

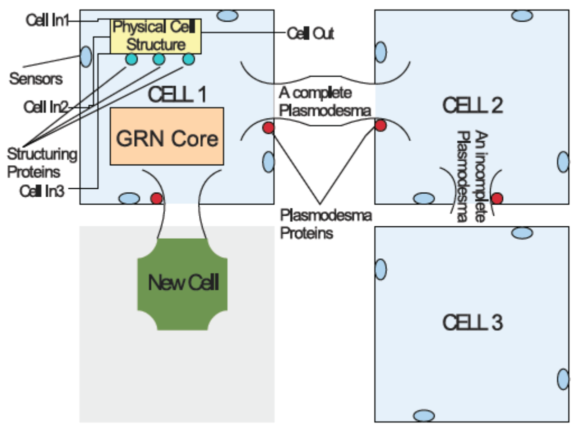


Figure 5. In a multi-cellular environment using the four basic protein types (ABCD), which are described in Table I, a cell is able to: interact with its environment, replicate (grow), structure itself, and form a complex multicellular organism. The basic functions of some proteins are demonstrated in this figure. Only cell 1 is drawn completely, certain components are omitted in other cells for clarity. In the example, cells 1 and 2 both have active Plasmodesmata proteins, which cause the formation of a channel on both cells towards the other, creating a Plasmodesmata to allow free movement of proteins from one cell to other. Cells 1 and 2 both also have active Plasmodesmata proteins on their southern sides. Cell 1's southern neighbour is a dead cell, so the active Plasmodesmata protein initiates a growth process in that direction. However, cell 2's southern

neighbour is an alive cell with no Plasmodesmata protein, thus cell 2 forms an unconnected channel on its southern wall. The direction in which the Plasmodesmata proteins are active, is encoded in the postcondition of the gene, hence, there is one species of Plasmodesmata protein with four different behaviours. The four sensors drawn monitor the outside activity on four sides of each cell and produce different messenger molecules with changing environment. The structuring proteins are produced by the GRN to change the physical structure of the cell, which is connected to the physical inputs and outputs of the cell [23].

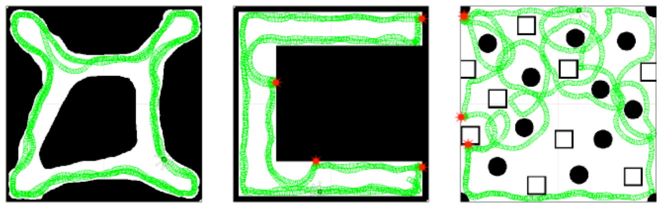


Figure 6. Comparison of the behaviour of the GRN controller in different maps [24].

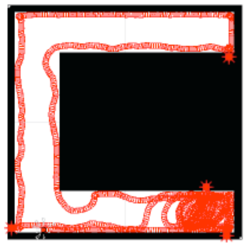


Figure 7. Results from experiments with a real e-puck [24].

IV. Evolving Silicon Architectures

Fundamental to the continued growth of the semiconductor industry is Moore’s Law, which states that the number of transistors integrated on a chip will double every 18 months, owing to the shrinking of devices through advances in technology. Recently, the scale of devices has approached the level where the precise placement of individual dopant atoms will affect the output characteristics of a device. As these intrinsic variations become more abundant, higher failure rates and lower yields will be observed from conventional designs. Coping with intrinsic variability has been recognised as one of the major unsolved challenges faced by the semiconductor industry [25].

The previous work reported in this paper has been focused on implementing evolvable mechanisms on hardware. The problem of cell design in the semi-conductor industry can have evolutionary mechanisms applied to it, but now this must be done using models of the devices (45nm and below) and simulators to test the efficacy of the resulting cell designs (circuits).

In this work the device models have been created by our collaborators at the University of Glasgow and the device simulator used was SPICE [25]. One of the issues that comes into focus when simulating, rather than implementing on the actual device, is a test bench to allow appropriate measurements to be made and hence objective functions to be compared. Figure 8 illustrates a typical test bench used in this work. Note that this does not simply use a single functional output to measure the circuit performance but a number of parameters which allows the evolutionary process to make a more realistic assessment of performance.

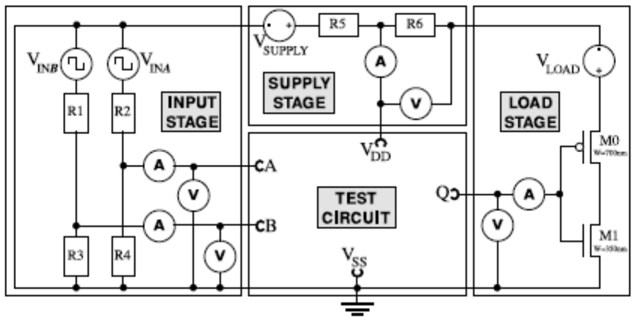


Figure 8. Typical test bench used during evolution [26].

By using this test bench evolution was able to assess circuits produced by evolution for important characteristics such as power and speed. Figure 9 illustrates one such experiment for the scaling of a XOR cell at 45nm [27].

An important part of this work was not simply to consider “standard” cell characteristics but additionally consider the affect of device variability on designs and to see whether evolution could reduce the effects of single transistor variability within a cell structure. In this case additional objective measures were considers and a multi-objective system was created to allow 6-12 objectives to be considered at the same time. Figure 10 illustrates one such evolutionary design for XOR cells [27].

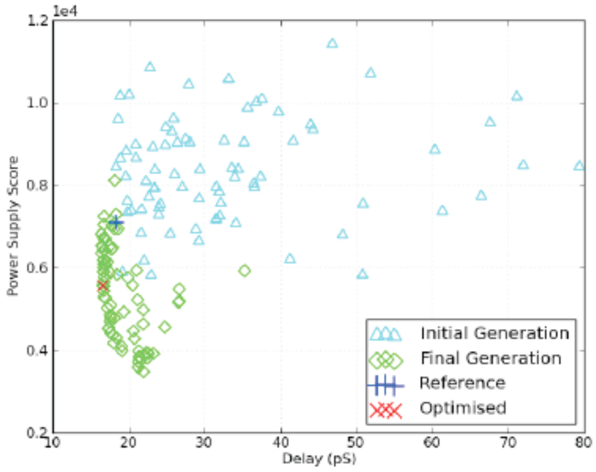


Figure 9. Comparison of initial random population, final optimised population and reference design for the XOR standard cell. Also shown is the optimised design that was chosen [25].

From this illustration it can be observed that not only are power and seed reduced, but the cloud of points (100,000 points) is smaller after evolution than before, illustrating that the variability of cell designs is less than before evolutionary mechanisms were applied.

What is clear from these early results is that evolution can play an important role in the design and production of future generations of silicon technology at the most fundamental level. If accepted by the microelectronics industries this could well represent a major (the major?) application area for evolvable hardware.

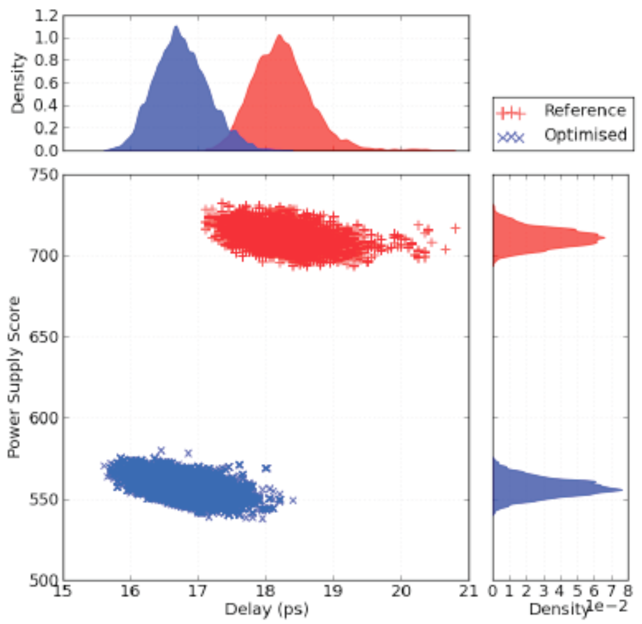


Figure 10. Comparison between reference and optimised designs for an XOR cell. Each scatter cloud contains 100,000 points generated from SPICE simulations of each design, which highlights the affect of intrinsic variability in terms of delay and power. The density plots (above and to the right on the scatter plot) show the distribution for the 100,000 simulations in terms of delay (x axis) and power (y axis) [27].

V. Metamorphic Systems

So far this paper has considered work that has been achieved using the general theme of evolvable hardware. In this last section it is interesting to consider some of the issues related to evolvable hardware and some future directions that might be taken. To date evolvable hardware research efforts have been typified by a number of features:

- Actual hardware/simulated hardware evolution.
- A reliance on reconfigurable devices when on actual hardware.
- A strong bias for electronic applications (almost exclusively defined by academics).
- A blurring between design optimization and design exploration.

Some of these have been illustrated in the earlier parts of this paper. Despite some successes, there are some underlying questions about the evolvable hardware field that remain largely ignored two of the main points being:

- Why do we want to evolve/adapt hardware? Why not just use conventional (i.e. non-evolutionary) techniques?
- What application areas should evolvable hardware techniques be applied to?

One line of research might be aimed at concentrating on reconfigurable systems, in general, where evolvable hardware is used only when it is really required (possibly when other more traditional techniques have failed, or do not exist) and to consider these reconfigurable systems in real-world scenarios, typically with quite tough constraints on systems requirements such as timing. In [28] these are called Metamorphic Systems (MS).

Consider the following: assume we have a system that can exist in several different variant forms S_A, S_B, S_C and so forth, each form exhibiting measurably different behavior. Let S_A be the current online variant. At some time later a transformation (reconfiguration) occurs that changes the system from S_A to S_B, which might be denote $S_A \rightarrow S_B$. There might be a large, albeit finite, set of systems S to choose from. Unlike most evolvable hardware systems, transformations in Metamorphic systems would be reversible—i.e., $S_A \rightarrow S_B \Rightarrow S_B \rightarrow S_A$ is always possible at some future time, this is probably quite important in a real-time system. More generally, if S_i, $S_j \in S$ then $S_i \rightarrow S_j$ is a valid transformation for all i and j ($i \neq j$) and the number of transformations between S_i and S_j is not limited.

There are many reasons why such a transformational system might be required including:

- Increased reliability and/or availability.
- To meet differing timing constraints.
- To change functionality.
- To make use of, or release, resources for higher priority functions.

Basically all of these, and others one might consider, can be summarized by the statement:

The performance of S_A is no longer good enough. Consequently, we need to switch to a different variant as soon as possible.

Figure 11 shows the metamorphic system block diagram [28]. Two main processing loops are shown: the innate loop and the acquired loop. The innate loop incorporates a priori knowledge about the way the system should operate; predetermined transformations are initiated whenever specific operational conditions are present. The acquired loop incorporates a learning capability that can analyze long-time trends in system behavior and can predict possible performance degradation; transformations are then initiated as needed to forestall any future problems.

The substrate is the actual material that constitutes the system (e.g., silicon in electronic devices or DNA, see a little later in this paper). Inside the substrate are pre-designed system variants that produce different behaviours in different operational environment changes. Each variant is fully functional and meets all design specifications in their respective operational environments. These variants can have widely diverse internal structure and do not have to be created from a common reconfigurable platform.

It is felt that such a system would allow evolvable hardware to be applied more easily, and probably more effectively, to real-world problems and could show its worth where it is really required, when traditional methods have failed. This is very new work and is currently being pursued further.

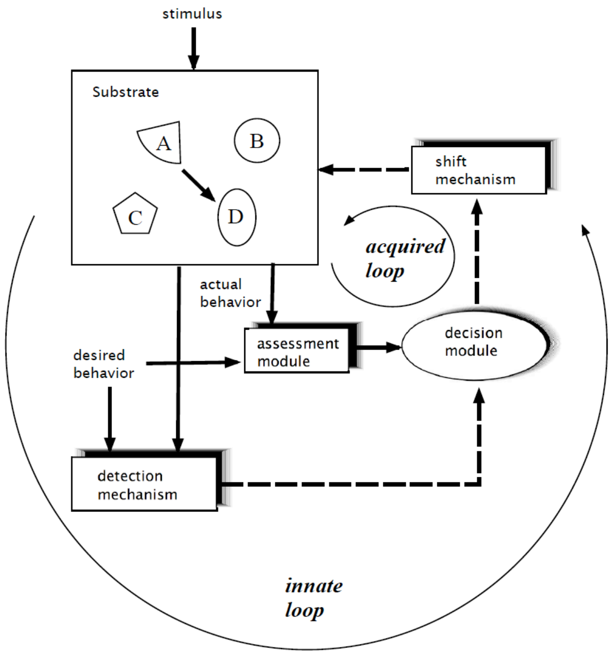


Figure 11. This figure shows the metamorphic system architecture. The substrate contains several system variants (labeled A, B, C and D). These variants are brought online as determined by either the innate loop or the acquired loop (see text). The internal architecture of the decision mechanism and the assessment module are often identical. The decision module is where the real intelligence resides and is expected to be implemented by a microcontroller or similar computing platform. This module decides when to switch to a new system variant and which variant in the substrate that should be. The selection mechanism is responsible for physically switching the variants [28].

One of the points highlighted here was that most of the work on evolvable hardware has been focused on electronic-based substrates. One recent advance is in the area of computation is creating processing structures with DNA strands. While this is a relatively new area of research and while evolution is not generally applied at this point (although you would imagine given the structure is DNA it should be well suited to evolution), one could easily see how it might fit into the Metamorphic system described here.

To start this process we need a set of mechanisms that will allow computation to be achieved with DNA strands from there we can start to consider evolving these within the Metamorphic paradigm.

Initial work has been started along this path. Figure 12 illustrates a simply Finite State Machine (FSM) typical of much computation, in this case with two states. Figure 13 illustrates how this simply FSM might be implemented using designed DNA structures and a clocking mechanism.

Once this mechanism can be implemented reliably and for larger systems, new areas of evolutionary hardware will open up.

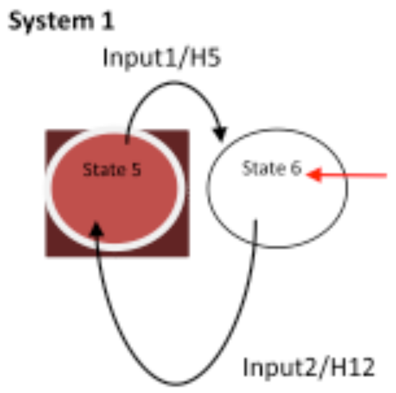


Figure 12. Simple two-state Finite State Machine.

VI. Discussion

In this paper a novel electronic hardware system has been described and some of the detailed architecture elements illustrated. This architecture was specifically designed and implemented with evolvable mechanisms in-mind. However, one of the limiting factors in evolutionary computing (particularly hardware implementations) is related to scalability.

The hardware architecture described was used in conjunction with developmental inspired set of mechanisms to consider more closely the design of large complex systems. While the example given in the paper is relatively straightforward (that of robot control) it does illustrate some progress in this direction.

There is a growth issue in the design of ever smaller silicon devices (transistors and basic building blocks) that of device variability. As devices sizes move towards 32nm and below atomistic issue cause significant (and potentially catastrophic) variability across silicon wafers. This appears to be a perfect example of an issue where evolution can play a significant and meaningful role in the future design of computational devices. Not only do results show improvements in cell power and speed but significant reduction of functional performance variability in the presence of device variability.

While mitigating against device variability appears to be an excellent application for evolvable hardware, really novel application are limited. The paper discusses a novel paradigm that incorporates evolution, when appropriate, but concentrates on adaptability in real-time systems: Metamorphic systems. While still in its inception, Metamorphic systems provide a mechanism to think about designs in a new way, and allow the mix of traditional and evolutional methods to be mixed. A significant difference between Metamorphic systems and other systems involving evolution is that different physical mediums are readily integrated into the system. The paper finishes by giving some early ideas of how DNA strands might provide a computation platform within a Metamorphic system.

Acknowledgements

The author would like to thank Andrew Greensted, Martin Trefzer, Tuze Kuyucu, James Walker, James Hilder, Cristina Santini and Garry Greenwood for their help in much of the work reported in this paper. Significant parts of the work reported in this paper was supported by various project grants funded by EPSRC (EP/E028381/1, EP/E001610/1).

References

[1] Koza, J., Yu, J., Keane, M.A. and Mydlowec. W. 'Use of conditional developmental operators and free variables in automatically synthesizing generalized circuits using genetic programming', 2nd NASA / DoD Workshop on Evolvable Hardware, pp 5 – 15, 2000.

[2] Greenwood, G.W. and Tyrrell, A.M. 'Introduction to Evolvable Hardware: A Practical Guide for Designing Self-adaptive Systems', John Wiley & Son, Inc., Publication, 2007.

[3] Yao, X. and Higuchi, T. 'Promises and challenges of evolvable hardware', IEEE Transactions on Systems, Man & Cybernetics, Part C 29(1),pp 87-97, 1999.

[4] Sanchez, E., Mange, D., Sipper, M., Tomassini, M., Perez-Uribe, P. and Stauffer, A. 'Phylogeny, Ontogeny, and Epigenesis: Three Sources of Biological Inspiration for Softening Hardware', Evolvable Systems: from Biology to Hardware, ICES 96, pp 35-54, 1996.

[5] Higuchi, T. et al, 'Real-World Applications of Analog and Digital Evolvable Hardware', IEEE Transactions on Evolutionary Computation, vol 3, no 3, pp 220-235, September 1999.

[6] Lohn, J., Kraus, W. and Linden, D. 'Evolutionary optimization of a quadrifilar helical antenna', IEEE International Symposium of the Antennas and Propagation Society, pp 814–817, 2002.

[7] Lohn, J.D., Hornby, G., Rodriguez-Arroyo, A., Linden, D., Kraus, W. and Seufert, S. 'Evolutionary Design of an X-Band Antenna for NASA's Space Technology 5 Mission', 3rd NASA/DoD Conference on Evolvable Hardware, pp 1-9, 2003.

[8] Lohn, J., Kraus, W. and Hornby, G. 'Automated design of a MEMS resonator', IEEE Congress on Evolutionary Computation (CEC07), pp 3486–3491, 2007.

[9] Oltean, M. 'Switchable glass: a possible medium for evolvable hardware', NASA/ESA Conference on Adaptive Hardware & Systems, pp 81–87, 2006.

[10] Lukac, M. and Perkowski, M. 'Evolving quantum circuits using genetic algorithm', NASA/DOD Conf. on Evolvable Hardware & Systems, pp 177–185, 2002.

[11] Haddow, P.C., Hartmann, M. and Djupdal, A. 'Addressing the Metric Challenge: Evolved versus Traditional Fault Tolerant Circuits', 2nd NASA/ESA Conference on Adaptive Hardware and Systems, pp 431-438, 2007.

[12] Spartan-3 Starter Kit Board User Guide, http://www.xilinx.com

[13] Torresen. J. 'A scalable Approach to Evolvable Hardware', International Conference on Evolvable Systems: from Biology to Hardware, (ICES98), pp 57-65, 1998.

[14] Sekanina, L. 'Evolvable Hardware: from Applications to Implications for the theory of Computation', in Unconventional Computation, LNCS, vol 5715, pp 24-36, 2009.

[15] Stoica, A., Keymeulen, D., Thakoor, A., Daud, T., Klimech, G., Jin, Y., Tawel, R. and Duong, V. 'Evolution of analog circuits on field programmable transistor arrays', NASA/DoD Workshop on Evolvable Hardware (EH2000), pp 99-108, 2000.

[16] Langeheine, J., Becker, J., Folling, F., Meier, K. and Schemmel, J. 'Initial studies of a new VLSI field programmable transistor array', 4th International Conference on Evolvable Systems: From Biology to Hardware, pp62-73, 2001.

[17] Greensted, A.J., and Tyrrell, A.M. 'RISA: A Hardware Platform for Evolutionary Design', IEEE Workshop on Evolvable and Adaptive Hardware, Hawaii, pp 1-7, April 2007.

[18] Greensted, A.J. and Tyrrell, A.M. 'Extrinsic Evolvable Hardware on the RISA Architecture', 7th International Conference on Evolvable Systems, Wuhan, China, pp 244-255, September 2007.

[19] Wolpert, L., Beddington, R., Jessell, T., Lawrence, P., Meyerowitz, E. and Smith, J. 'Principles of development', Oxford University Press, Oxford, 2002.

[20] Gordon, T. and Bentley, P.J. 'Towards development in evolvable hardware', NASA/DoD Conference on Evolvable Hardware, pp 241-250, 2002.

[21] Haddow, P.C., Tufte, G. and ven Remortel, P. 'Shrinking the genotype: L-systems for EHW?', International Conference on Evolvable Systems: from Biology to Hardware, pp128-139, 2001.

[22] Tufte, G. and Haddow, P.C. 'Towards Development on a silicon-based cellular computing machine', the Journal of Natural Computing, Vol 4, no. 4, pp 387-416, 2005.

[23] Kuyucu, T., Trefzer, M.A., Miller, J.F. and Tyrrell, A.M. 'An Investigation of the Importance of Mechanisms and Parameters in a Multi-cellular Developmental System', IEEE Transactions on Evolutionary Computation, **in press** 2011.

[24] Trefzer, M.A., Kuyucu, T., Miller, J.F. and Tyrrell, A.M. 'Evolution and Analysis of a Robot Controller Based on a Gene Regulatory Network', 9th International Conference on Evolvable Systems, Springer, York, pp 61-72, September 2010.

[25] Walker, J.A., Sinnott, R., Stewart, G., Hilder, J.A. and Tyrrell, A.M. 'Optimising Electronic Standard Cell Libraries for Variability Tolerance Through the Nano-CMOS Grid', Philosophical Transaction A of the Royal Society, Vol. 368, pp 3967-3981, August 2010.

[26] Walker, J., Hilder, J. and Tyrrell, A.M. 'Evolving Variability-Tolerant CMOS Designs', 8th International Conference on Evolvable Systems, Prague, Czech Republic, pp 308-319, September 2008.

[27] Walker, J., Hilder, J. and Tyrrell, A.M. 'Towards Evolving Industry-feasible Intrinsic Variability Tolerant CMOS Designs', 11th IEEE Congress on Evolutionary Computation, Trondheim, Norway, pp 1591-1598, May 2009.

[28] Greenwood, G. and Tyrrell, A.M. 'Metamorphic Systems: A New Model for Adaptive System Design', 12th IEEE Congress on Evolutionary Computation (CEC10), Barcelona, pp 3261-3268, July 2010.

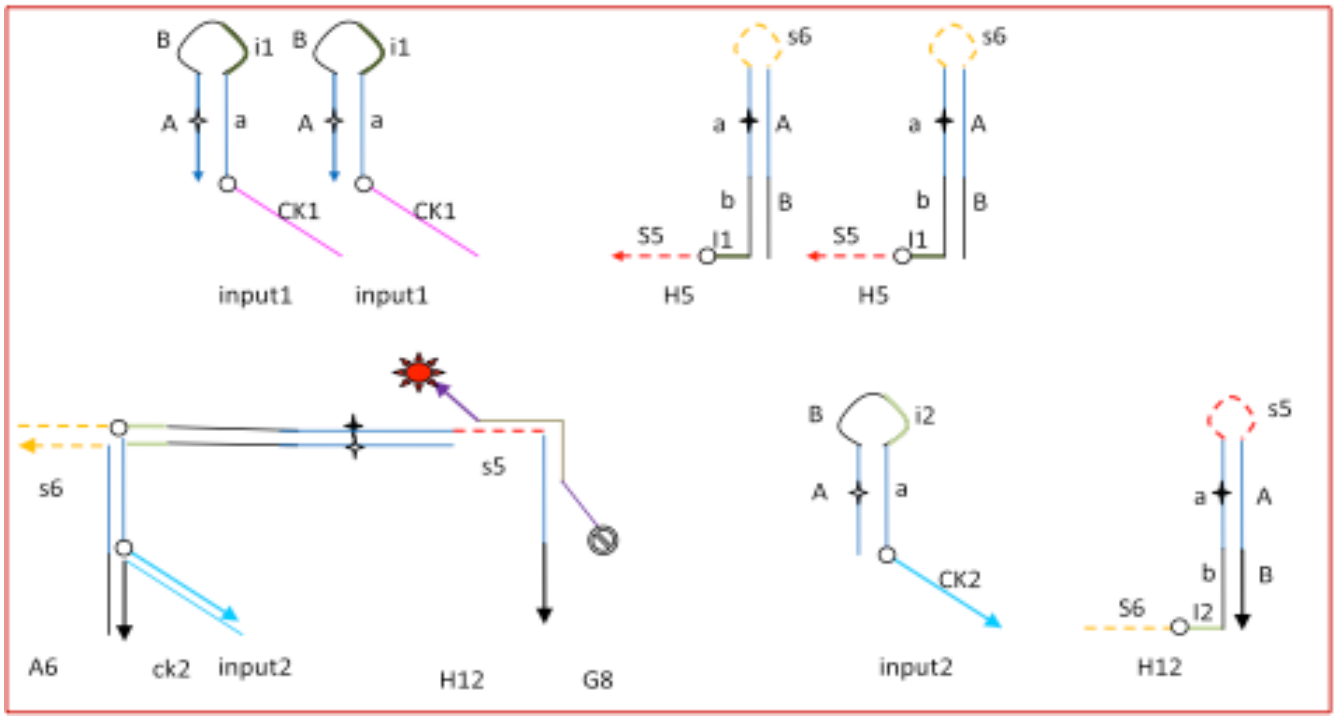


Figure 13. DNA structures and sequence, including clocking mechanisms, to implement the simple two-state Finite State Machine shown in Figure 12.

Next Generation Sequencing Data Processing
How reconfigurable computing can help?

Prof. Dominique Lavenier
CNRS/IRISA, Rennes, France

Abstract

With the fast progress of next generation sequencing (NGS) machines, genomics research is currently strongly shaken. These new biotechnologies generate impressive flow of raw genomic data from which pertinent and significant information must be extracted. To sustain this high data processing throughput, parallelism is the only way. Today, two major challenges must be considered: (1) develop new parallel algorithms for new applications coming from the wide possibilities open by NGS technologies; (2) develop new parallel architectures as an alternative to huge clusters currently used in bioinformatics computing centers. Reconfigurable computing can address these two challenges by providing dedicated parallel algorithms tailored to ad hoc hardware. The talk will present the current NGS technologies, the standard associated treatments and the challenges RC should be able to bring efficient solutions.

Design-Space Exploration of Systolic Arrays for Biosequence Algorithms

(Invited Paper)

Jeremy Buhler, Roger Chamberlain
Department of Computer Science and Engineering
Washington University
St. Louis, Missouri 63130
Email: {jbuhler,roger}@wustl.edu

Arpith Jacob
IBM T.J. Watson Research Center
Yorktown Heights, New York 10598
Email: jarpith@wustl.edu

***Abstract*—Next-generation sequencing techologies have dramatically increased the amount of new DNA and protein sequence being produced by biologists. These vast and growing data volumes challenge existing computational approaches to analyzing experimentally produced biosequences. One way to make very large analyses practical is to design customized computational accelerators to implement them on, e.g., FPGAs or chip multiprocessors. However, the cognitive burden associated with programming these devices limits their utility for bioinformaticians.**

In this abstract, we describe a high-level approach to the design of accelerators for biosequence analyses based on dynamic programming. These computations can be described at the level of recurrences, for which we can find the best mapping to a given hardware platform using the tools of polyhedral analysis. We recount recent advances by our group in automated design of efficient accelerators from recurrences and describe a vision for an end-to-end suite of tools for bioinformatics accelerator design.

I. Introduction

The advent of next-generation sequencing technologies in the middle of the last decade has caused a quantum leap in the speed and affordability of DNA sequencing. Whereas capillary instruments like the ABI 3730 delivered tens of kilobases of sequence per hour for around \$500 per megabase [1], today's Illumina and ABI instruments can produce data up to 1000-fold faster, for well under a dollar per megabase [2]. Projections for upcoming single-molecule instruments suggest that, within a few years, they might produce data up to 100-fold faster still, with correspondingly lower costs and substantially longer read lengths than today's instruments of choice.

This extreme growth in the speed and cost-efficiency of sequencing has huge implications for the practice of bioinformatics. While many sequence analysis problems exhibit a high degree of parallelism, the rate at which new data is being generated is outstripping Moore's Law-based increases in logic areal density, which control the rate at which we can pack more computing cores into our clusters. We must therefore qualitatively change our approach to computing on biological sequence data to meet the challenge of analyzing today's and tomorrow's data volumes. This abstract therefore focuses on improving performance by adapting the architecture of our computational engines to facilitate faster biosequence analysis.

II. Accelerating Dynamic Programming: Promise and Challenges

Many of the key algorithms used to analyze biological sequences use the paradigm of dynamic programming (DP) [3]. Such algorithms are used in, e.g., classical sequence comparison [4], RNA structure prediction [5], [6], and alignment to protein and RNA family models [7], [8]. These algorithms typically have running times quadratic, cubic, or worse in the sizes of their inputs and so can become a bottleneck when applied to large biological data sets. However, this bottleneck can be ameliorated if the algorithms are implemented in a way that exposes their inherently high degree of fine-grained parallelism. In particular, these DP recurrences can be mapped to parallel implementations in the form of *systolic arrays* [9] – collections of simple, synchronized processing elements connected by a regularly structured network of communication links.

A recurrence whose data dependencies are (or can be made) uniform linear can be realized as a systolic array [10], which in turn can be implemented as an accelerator in custom logic (ASIC or FPGA). Within bioinformatics, there is a long history of custom logic accelerators for pairwise DNA and protein sequence comparison [11], [12], [13] and more ambitious designs have recently tackled more complex recurrences such as those for RNA folding [14]. The best accelerators for these tasks can achieve speedups of two orders of magnitude or more relative to general-purpose processors.

Despite the need for high performance and the great potential of systolic array-based accelerators for bioinformatics, such accelerators are still relatively uncommon "in the wild." One reason for accelerators' lack of penetration in bioinformatics is the difficulty of accelerator design for bioinformatics programmers, and indeed for anyone trained primarily in software rather than hardware design. Accelerating algorithms on a given platform, while it has a high return in performance, entails adjustment to an entirely new programming paradigm, detailed knowledge of the target hardware, and adaptation to the quirks of its design and synthesis tools. Bioinformaticians are not willing to spend the time and effort needed to build an accelerator, particularly if the algorithm to be accelerated is itself still being tweaked and enhanced.

Previous work has sought to reduce the time and difficulty of hardware design by directly building circuit descriptions from a C-like language, such as ImpulseC or Handel-C. Such a paradigm presents a friendlier face to the average programmer than a hardware design language (HDL), so one might expect that it would lower the barrier to producing high-quality accelerators. Unfortunately, these frameworks still rely on the programmer to map a DP recurrence to a C-level specification (e.g. a loop nest). Choosing this mapping appropriately is far from trivial and is critical to achieve high performance. A good mapping balances conflicting needs for high performance, bounded utilization of logic and memory resources, and ability to handle problems of realistic size with a limited amount of logic. The demands of this difficult balancing act can make high-performance systolic array design, even with enhanced low-level tool support, inaccessible to all but a few expert designers.

III. A Vision of Rapid Accelerator Design for Bioinformatics

To make accelerator design easy and fast for bioinformatics, we propose two principles. First, the right level of abstraction for specifying biosequence analysis algorithms is typically a DP recurrence, or a formalism such as a hidden Markov model whose use entails such recurrences. Second, once the user specifies a recurrence, reducing it to an implementation for a given target should be entirely the job of a compiler. In particular, a compiler that produces systolic array implementations should have sufficient knowledge of the target platform and the typical range of input sizes to find a high-performance design automatically.

This vision has antecedents in both software and hardware. Dynamite [15] and its successor C4 [16] are explicitly designed to produce good C/C++ code for sequence analysis from general recurrence specifications. These tools are, however, CPU-focused and do not concern themselves with exploiting fine-grained parallelism. Similarly, tools such as MMAlpha [17], [18] and Paro [19] assist programmers in mapping arbitrary dynamic programming recurrences to efficient parallel or nested-loop implementations. However, these tools are not generally used for bioinformatics, are only partly automated, and still require considerable expertise to use.

The ideal toolkit for building bioinformatics accelerators would combine the best features of this prior work. On the one hand, it would limit its ambitions to the domain of biosequence analysis and would therefore, like Dynamite, make strong assumptions about its inputs and provide a high degree of automation. On the other, it would incorporates a deep understanding of how to map recurrences to efficient parallel systolic-array implementations. Realizing such a toolkit would greatly enhance bioinformaticians' ability to exploit the work of computer scientists and computing system designers.

IV. Progress and Future Work

In our work to date, we have devised new methods to design accelerators for throughput-oriented computations, such as are commonly found in bioinformatics, and have demonstrated large speedups for the RNA folding problem, which is challenging to accelerate with a systolic array. In the future, we plan to deploy these innovations as part of a more complete pipeline for bioinformatics accelerator design.

To achieve greater automation in searching for good accelerator designs, we devised a general technique for designing *throughput-optimal* systolic array mappings for a uniform recurrence [20]. Throughput measures the sustained rate at which a stream of problem instances can be computed by an accelerator; in contrast, *latency* measures the time to complete one instance. Traditional systolic array design focuses on minimizing latency; however, in a bioinformatics context, we typically need to solve a large collection of problems at once, e.g. aligning a query to a database of sequences or folding many candidate RNAs to find those that are highly structured. We formulated a throughput measure – the *block pipelining period*, or time required between submissions of successive problems to the array – and showed how to automatically search the domain of possible array designs to find those that optimize this measure subject to resource constraints.

We applied our techniques to the domain of RNA folding to automatically derive accelerators for the Nussinov folding algorithm [5] whose throughput is several times that of classical latency-space-optimal arrays for the problem. Moreover, modern circuit synthesis tools can easily be made to internally pipeline our arrays within each processing element to increase the resulting circuits' clock rates.

Because high-performance systolic arrays are often infeasible to construct for realistically-sized inputs on space-constrained hardware platforms, we also investigated the design of low-dimensional arrays that reduce parallelism but require substantially fewer processors for a given input size. For a complex recurrence, choosing among space-constrained designs can dramatically affect the resource usage of the resulting array. As a demonstration, we accelerated the Zuker RNA-folding algorithm [6], which uses a much more detailed, table-driven energy function than the Nussinov algorithm. By carefully choosing among array mappings, we ensured that most processing elements did not need the algorithm's space-intensive lookup tables [21]. As a result, we were able to fold RNAs of 273 bases on a single Xilinx Virtex 4 LX100-12 FPGA, achieving speedups of two orders of magnitude over CPU-based implementations and an order of magnitude over existing FPGA- and GPU-based implementations.

Our achievements to date form the core of a planned accelerator design pipeline aimed at bioinformaticians. To fully realize this pipeline, we will need to automate various steps surrounding the core search over systolic array designs. We will automate, or at least provide templates for, the mapping of common DP-based biosequence analysis algorithms to uniform linear systems of recurrences. This problem, which we solved manually for RNA folding, is much more tractable for the handful of common recurrence shapes in bioinformatics than for arbitrary recurrences. We will also focus on automating low-level HDL code generation from our systolic array designs. Finally, we will explore the feasibility of mapping not only onto reconfigurable logic platforms but also onto other highly constrained parallel architectures, such as the SIMD threaded model of a GPU.

Ultimately, our goal is to put the power to exploit massive, fine-grained parallelism into the hands of the typical bioinformatics programmer, providing a boost to bioinformaticians in their ongoing race to keep up with advances in sequencing technology.

Acknowledgment

Our work on systolic array design automation and RNA folding acceleration has been supported by NIH award R42 HG003225 and by NSF awards ITR-427794 and CNS-0905368.

References

[1] J. Shendure and H. Ji, "Next-generation DNA sequencing," *Nature Biotechnology*, vol. 26, pp. 1135–45, 2008.

[2] National Human Genome Research Institute, "DNA sequencing costs," 2011, http://www.genome.gov/sequencingcosts/.

[3] R. Bellman, "The theory of dynamic programming," *Bulletin of the American Mathematical Society*, vol. 60, pp. 503–516, 1954.

[4] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–97, Mar. 1981.

[5] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman, "Algorithms for loop matchings," *SIAM Journal of Applied Mathematics*, vol. 35, pp. 68–82, 1978.

[6] M. Zuker, "Computer prediction of RNA structure," *Methods in Enzymology*, vol. 180, pp. 262–88, 1989.

[7] S. R. Eddy, "Profile hidden Markov models," *Bioinformatics*, vol. 14, pp. 755–63, 1998.

[8] E. P. Nawrocki, D. L. Kolbe, and S. R. Eddy, "Infernal 1.0: Inference of RNA alignments," *Bioinformatics*, vol. 25, pp. 1335–7, 2009.

[9] H. T. Kung and C. E. Leiserson, "Algorithms for vlsi processor arrays," in *Introduction to VLSI systems*, C. Mead and L. Conway, Eds. Addison-Wesley, 1980, ch. 8.3.

[10] R. Karp, R. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the ACM*, vol. 14, pp. 583–90, 1967.

[11] R. Lipton and D. Lopresti, "A systolic array for rapid string comparison," in *Proceedings of the 1985 Chapel Hill Conference on VLSI*, 1985, pp. 363–376.

[12] E. Chow, T. Hunkapillar, J. Peterson, and M. Waterman, "Biological information signal processor," in *Proceedings of the IEEE Applications-Specific Array Processors Conference*, 1991.

[13] J. Hirschberg, R. Hughey, K. Karplus, and D. Speck, "Kestrel: a programmable array for sequence analysis," in *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors*, 1996, pp. 25–34.

[14] Y. Dou, F. Xia, X. Zhou, and X. Yang, "Fine-grained parallel application specific computing for RNA secondary structure prediction on FPGA," in *26th International Conference on Computer Design*, 2008, pp. 240–247.

[15] E. Birney and R. Durbin, "Dynamite: a flexible code generating languages for dynamic programming methods used in sequence comparison," in *Proceedings of the 5th International Conference on Intelligent Systems in Molecular Biology*, 1997, pp. 56–64.

[16] G. Slater and E. Birney, "Automated generation of heuristics for biological sequence comparison," *BMC Bioinformatics*, vol. 6, p. 31, 2005.

[17] H. Le Verge, C. Mauras, and P. Quinton, "The ALPHA language and its use for the design of systolic arrays," *Journal of VLSI Signal Processing*, vol. 3, pp. 173–182, 1991.

[18] A. Mozipo, D. Massicote, P. Quinton, and T. Risset, "Automatic syntehsis of a parallel architecture for Kalman filtering using MMAlpha," in *Proceedings of the 1999 IEEE Canadian Conference on Electrical and Computer Engineering*, 1999.

[19] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich, "PARO: synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications," in *Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing*, 2008, pp. 287–93.

[20] A. Jacob, J. Buhler, and R. Chamberlain, "Design of throughput-optimized arrays from recurrence abstractions," in *Proceedings of the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2010, pp. 133–40.

[21] ——, "Rapid RNA folding: Analysis and acceleration of the Zuker recurrence," in *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010, pp. 87–94.

SESSION

INVITED SESSION - RUNTIME ADAPTIVE EMBEDDED SYSTEMS AND ARCHITECTURES

Chair(s)

PROF. ROMAN LYSECKY

INVITED TALKS

i-Core: A run-time adaptive processor for embedded multi-core systems

Jörg Henkel, Lars Bauer, Michael Hübner, and Artjom Grudnitsky
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{henkel, lars.bauer, michael.huebner, artjom.grudnitsky} @ kit.edu
Submitted as Invited Paper to ERSA 2011

Abstract

We present the i-Core (*Invasive Core*), an Application Specific Instruction Set Processor (ASIP) with a run-time adaptive instruction set. Its adaptivity is controlled by the run-time system with respect to application properties that may vary during run-time. A reconfigurable fabric hosts the adaptive part of the instruction set whereas the rest of the instruction set is fixed. We show how the i-Core is integrated into an embedded multi-core system and that it is particularly advantageous in multi-tasking scenarios, where it performs applications-specific as well as system-specific tasks.

1. Introduction and Motivation

Embedded processors are the key in rapidly growing application fields ranging from automotive to personal mobile communication, computation, and entertainment, to name just a few. In the early 1990s, the term ASIP has emerged denoting processors with an application-specific instruction set (Application Specific Instruction-set Processors). They are more efficient in one or more design criteria like 'performance per area' and 'performance per power' [1] compared to mainstream processors and eventually make today's embedded devices (which are often mobile) possible. Nowadays, the term ASIP comprises a far larger variety of embedded processors allowing for customization in various ways including a) instruction set extensions, b) parameterization and c) inclusion/exclusion of predefined blocks tailored to specific applications (like, for example, an MPEG-4 decoder) [1]. An overview for the benefits and challenges of ASIPs is given in [1-3].

A generic design flow of an embedded processor can be described as follows:

i) an application is analyzed/profiled
ii) an instruction set extension (containing so-called *Special Instructions*, SIs) is defined
iii) the instruction set extension is synthesized together with the core instruction set
iv) retargetable tools for compilation, instruction set simulation, and so on, are (often automatically) created and application characteristics are analyzed
v) the process might be iterated several times until design constraints comply

Automatically detecting and generating SIs from the application code (like in [4]) plays a major role for speeding-up an application and/or for power efficiency. Profiling and pattern matching methods [5, 6] are typically used along with libraries of reusable functions [7] to generate SIs.

However, fixing critical design decisions during design time may lead to embedded processors that can hardly react to an often non-predictive behavior of today's complex applications. This does not only result in reduced efficiency but it also leads to an unsatisfactory behavior when it comes to design criteria like 'performance' and 'power consumption'. A means to address this dilemma is reconfigurable computing [8-11] since its resources may be utilized in a time-multiplexed manner (i.e. reconfigured over time). A large body of research has been conducted in interfacing reconfigurable computing fabrics with standard processor cores (e.g. using an embedded FPGA [12-14]).

This paper presents the i-Core, a reconfigurable processor that provides a high degree of adaptivity at the level of instruction set architecture (through using reconfigurable SIs) and microarchitecture (e.g. reconfigurable caches and branch predictions). The potential performance advantages at both levels are exploited and combined which allows for a high degree of adaptivity that is especially beneficial in run-time varying multi-tasking scenarios.

Paper structure: Section 2 presents state-of-the-art related work for reconfigurable processors. An overview of our i-Core processor is given in Section 3 where we explain the way it is integrated into a heterogeneous multi-core system and the kind of adaptivity it provides with respect to SIs and the microarchitecture. Section 4 explains how SIs are modeled and how the programmer can express which SIs are demanded by the application to trigger their reconfiguration. Performance results of applications executing on the i-Core are given in Section 5 and conclusions are drawn in Section 6.

2. Related Work

Diverse approaches for reconfigurable processors were investigated particularly within the last decade [8-11]. The Molen Processor couples a reconfigurable coprocessor to a core processor via a dual-port register file and an arbiter for shared memory [15]. The application binary is extended to include instructions that trigger the reconfigurations and control the usage of the reconfigurable coprocessor. The OneChip project [16, 17] uses tightly-coupled Reconfigurable Functional Units (RFUs) to utilize reconfigurable computing in a processor. As their speedup is mainly obtained from streaming applications, they allow their RFUs to access the main memory, while the core processor (i.e. the non-reconfigurable part of the processor) continues executing [18]. Both approaches target a single-tasking environment and statically predetermine which SIs shall be reconfigured at ‘which time’ and to ‘which location’ on the reconfigurable fabric. This will lead to conflicts if multiple tasks compete for the reconfigurable fabric (not addresses by these approaches).

The Warp Processor [19] automatically detects computational kernels while the application executes. Then, custom logic for SIs is generated at run-time through on-chip micro-CAD tools and the binary of the executing program is patched to execute them. This potentially allows adapting to changing multi-tasking scenarios. However, the required online synthesis may incur a non-negligible overhead and therefore the authors concentrate on scenarios where one application is executing for a rather long time without significant variation of the execution pattern. In these scenarios, only one online synthesis is required (i.e. when the application starts executing) and thus the initial performance degradation accumulates over time. Adaptation to frequently changing requirements –as typically demanded in a multi-tasking system– is not addressed by this approach.

The Proteus Reconfigurable Processor [20] extends a core processor with a tightly-coupled reconfigurable fabric. It concentrates on Operating System (OS) support with respect to SI opcode management to allow different tasks to share the same SI implementations. Proteus’ reconfigurable fabric is divided into multiple Programmable Functional Units (PFUs) where each PFU may be reconfigured to contain one SI (unlike ReconOS [21], where the reconfigurable hardware is deployed to implement entire threads). However, when multiple tasks exhibit dissimilar processing characteristics, a task may not obtain a sufficient number of PFUs to execute all SIs in hardware. Therefore, some SIs will execute in software, resulting in steep performance degradation.

The RISPP processor [22, 23] uses the reconfigurable fabric in a more flexible way by introducing a new concept of SIs in conjunction with a run-time system to support them. Each SI exists in multiple implementation alternatives, reaching from a pure software implementation (i.e. without using the reconfigurable fabric) to various hardware implementations (providing different trade-offs between the amount of required hardware and the achieved performance). The main idea of this concept is to partition SIs into elementary reconfigurable data paths that are connected to implement an SI. A run-time system then dynamically chooses one alternative out of the provided options for SI implementations, depending on run-time application requirements. It focuses on single-tasking scenarios and does not aim to share the reconfigurable fabric among multiple tasks or among user tasks and OS tasks. KAHRISMA [24, 25] extends the concepts of RISPP by providing a fine-grained reconfigurable fabric along with a coarse-grained reconfigurable fabric that can then be used to implement SIs and to realize pipeline- or VLIW processors. Therefore, KAHRISMA supports simultaneous multi-tasking (one task per core), but it does not consider executing multiple tasks per core or adapting the microarchitecture (e.g. cache- or branch-prediction) of a core.

Altogether, only Proteus explicitly targets multi-tasking systems in the scope of reconfigurable processors that use a fine-grained reconfigurable fabric. The concept of PFUs does not provide the demanded flexibility to support multiple tasks efficiently though. RISPP and KAHRISMA provide a flexible SI concept but the challenge of sharing the reconfigurable fabric and the configuration of the microarchitecture among competing tasks is not addressed. The Warp processor provides the potentially highest flexibility, but it comes at the cost of online synthesis, which limits the scenarios in which this flexibility can be efficiently used. Hence, when studying state-of-the-art approaches, the following challenge remains: providing an adaptive reconfigurable processor that can share the reconfigurable fabric efficiently among multiple user tasks while providing an adaptive microarchitecture that can adapt to varying task requirements.

3. i-Core Overview

The i-Core is a reconfigurable processor that provides a run-time reconfigurable instruction set architecture (ISA) along with a run-time reconfigurable microarchitecture. The ISA consists of two parts, the so-called core ISA (cISA) and the Instruction Set Extension (ISE). The cISA is statically available (i.e. implemented with non-reconfigurable hardware) and the ISE represents the task-specific components of the ISA that are realized as reconfigurable Special Instructions (SIs). The i-Core uses a fine-grained reconfigurable fabric (i.e. an embedded FPGA, e.g. [12-14]) to provide

i) task-specific ISEs,
ii) OS-specific ISE, and
iii) an adaptive microarchitecture that – among others – allows for supporting and executing both kinds of ISEs efficiently by performing run-time reconfigurations

This approach exceeds the concept of state-of-the-art ASIPs, as it adds flexibility and additionally enables dynamic run-time adaptation towards the executing application to increase the performance.

The reconfigurable microarchitecture characterizes the concrete processor-internal realization of the i-Core for a given ISA. It refers to the internal process of instruction handling and it is developed with respect to a predefined ISA (SPARC-V8 [26] in our case). Summarizing, the ISA specifies the instruction set that can be used to program the processor (without specifying how the instructions are implemented) and the microarchitecture specifies its implementation and further ISA-independent components (e.g. caches and branch prediction).

Figure 1 shows how the i-Core is embedded into a heterogeneous loosely-coupled multi-core system which is partitioned into tiles that consist of processor cores and local shared memory. Each tile is connected to one router that is part of an on-chip network to connect the tiles to each other and to external memory. Within one tile, none, one, or multiple i-Core instances are located and connected to the tile-internal communication structure (similar to the other CPUs of the tile). The CPUs within the tile access the shared memory (an application-managed scratchpad) via the local bus. The i-Core uses a dedicated connection to the local memory, using two 128-bit ports to provide a high memory bandwidth to expedite the execution of SIs. When multiple i-Cores are situated within one tile, then the access to the local memory and the reconfigurable fabric is shared among them.

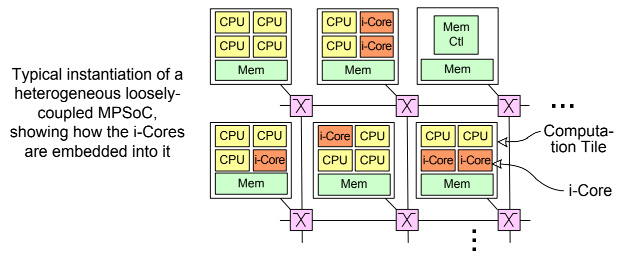

Typical instantiation of a heterogeneous loosely-coupled MPSoC, showing how the i-Cores are embedded into it
CPU
i-Core
Mem
Mem Ctl
Computa-tion Tile
i-Core

Figure 1: Integration of the i-Core into a heterogeneous multi-core system with on-chip interconnect network

Figure 2 provides an overview of the i-Core-internal adaptation options of the ISE and the microarchitecture. The ISE provides task-specific and system-specific SIs. Whereas the task-specific SIs are particularly targeted towards a certain application or application domain, the system-specific SIs support basic operating system (OS) functionalities that are required to execute tasks and to manage the multi-core system. The OS can (to some degree) be viewed as a set of tasks (e.g. task mapping, task scheduling, etc.) that can be accelerated by SIs. Typically, it is impossible to fulfill all ISE requests due to the limited size of the reconfigurable fabric. The SIs that are not implemented on the reconfigurable fabric at a certain point in time can be executed through an ‘unimplemented instruction’ trap as presented in more detail in [27].

SIs are prepared at compile time of the tasks and the calculations that are performed by an SI are fixed at run time. Instead, the *implementation* of a particular SI may change during run-time (details are explained in Section 4.1). These adaptations correspond to ISE-specific adaptations at the microarchitecture level. Figure 2 shows a reconfigurable fabric – in addition to the hardware of the processor pipeline – that is employed to realize an ISE. Depending on the executing tasks and their specific ISE requirements, the reconfigurable fabric is allocated or – as we call it – *invaded* to realize a certain subset of the requested ISEs (that is why we call it an *invasive Core*, i.e. i-Core). The concepts of *invasive computing* [28] are used to manage the competing requests of different tasks. In the scope of reconfigurable processors this means that each task specifies the SIs that it uses (i.e. ‘its requests’; details are presented in Section 4.2). Additionally, each task provides information which performance improvement (speedup) can be expected, depending on the size of the reconfigurable fabric that is assigned to it. A run-time system (part of the OS) then decides for the tasks that compete for the resources, which task obtains which share of the reconfigurable fabric (similar to the approach presented in [29] where the fabric of one task is partitioned among the SIs of that task).

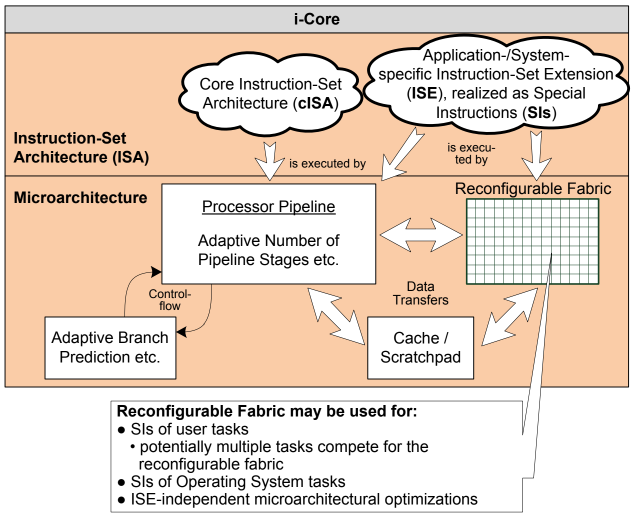

i-Core
Core Instruction-Set Architecture (cISA)
Application-/System-specific Instruction-Set Extension (ISE), realized as Special Instructions (SIs)
Instruction-Set Architecture (ISA)
is executed by
is executed by
Microarchitecture
Processor Pipeline
Adaptive Number of Pipeline Stages etc.
Reconfigurable Fabric
Control-flow
Data Transfers
Adaptive Branch Prediction etc.
Cache / Scratchpad
Reconfigurable Fabric may be used for:
• SIs of user tasks
• potentially multiple tasks compete for the reconfigurable fabric
• SIs of Operating System tasks
• ISE-independent microarchitectural optimizations

Figure 2: The Instruction-Set Architecture and Microarchitecture Adaptations of our i-Core

The highly adaptive nature of modern multi-tasking embedded systems intensifies the potential advantages that come along with an adaptive microarchitecture and instruction set architecture. The adaptations of the instruction set comprise flexible task-specific SIs that support at run-time adaptations in terms “performance per area” (details given in Section 4.1) depending on the number of tasks that execute on an i-Core at a specific time (and thus the amount of reconfigurable fabric that is available per task). In addition to the ISE, the i-Core also supports adapting its microarchitecture. The microarchitecture adaptations are independent upon the instruction set and they comprise:

- adaptive number of pipeline stages
- adaptive branch prediction
- adaptive cache/scratchpad configuration

Even though these optimizations are ISA-independent, they may significantly increase the performance of the executing task (or reduce the power consumption etc.). The number of pipeline stages is altered by combining neighbored stages and bypassing the pipeline registers between them. This reduces the maximal frequency of the processor but it may lead to power savings (reduced number of registers) and may be beneficial in terms of performance for applications where a complex control flow leads to many branch miss-predictions. Additionally, techniques like pipeline balancing [30] or pipeline gating [31] can be applied. Depending on a task’s requirements, the branch prediction scheme can also be changed dynamically, e.g. by providing different schemes and letting the task decide which one to use. Another example of performance-oriented adaptive design is *branch history length adaptation*, e.g. Juan et al. [32] explore dynamic history-length fitting and develop a method for dynamically selecting a history length that accommodates the current workload.

In addition, the cache can be changed in various ways. For instance, Albonesi [33] proposes to disable cache ways dynamically to reduce dynamic energy dissipation. Kaxiras et al. [34] reduce leakage power by invalidating and turning off the cache lines when they hold data that is not likely to be reused. The approaches of [35-37] use an adaptive strategy to adjust the cache line size dynamically to an application. In addition to these approaches, the microarchitecture of the i-Core exploits the availability of the fine-grained reconfigurable fabric to extend the size and associativity of the cache. For example, the size of the cache (e.g. number of cache lines) can be extended (using the logic cells of the reconfigurable fabric as fast memory), further parallel comparators can be realized to increase the associativity of the cache, or additional control bits can be assigned to each cache line for protocol purpose (e.g. error detection/correction schemes). Additionally, the memory of the cache can be reconfigured to be used as a task-managed scratchpad memory.

In summary, instruction set and microarchitecture adaptations target task-specific optimizations, for example, a particular task might benefit from a certain SI (part of the ISA) and a certain branch prediction (part of the microarchitecture). Additionally, a particular task might also benefit from different ISA/microarchitecture implementations at different phases of its execution (e.g. different computational kernels), i.e. the requirements of a sole task may change over time. Depending on the tasks that execute at a certain time and their requirements, the adaptations focus on:

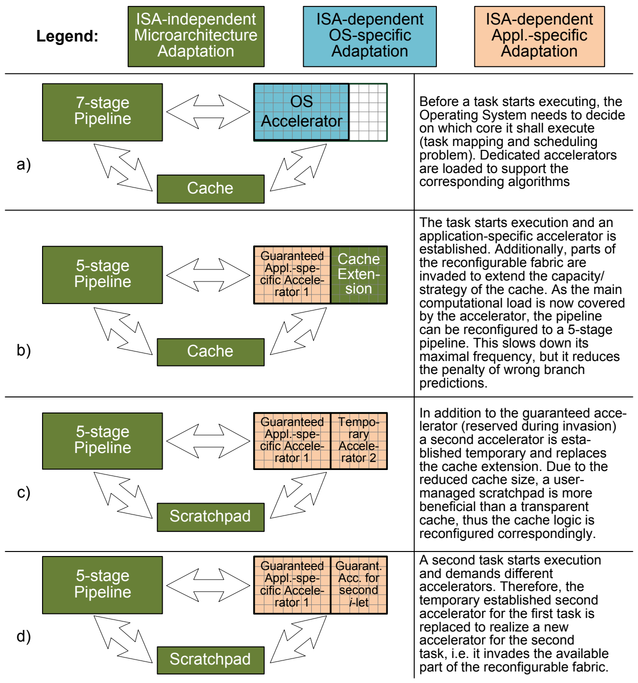
- some selected tasks (beneficial for those tasks at the cost of other tasks)
- operating system optimization (beneficial for all tasks)
- a trade-off between both

Determining this trade-off depends on the user-priorities of the executing tasks. This large degree of flexibility is an advantage in comparison to state-of-the-art adaptive processors (e.g. RISPP [22, 23] or KAHRISMA [24, 25]) as they focus on accelerating either the tasks or the operating systems (but not both) by improving either the instruction set or the microarchitecture (and gain, not both).

3.1. Partitioning the Reconfigurable Fabric among Special Instructions and Microarchitecture

The core ISA (cISA) is executed by means of a specific hardware at the microarchitecture level. Depending on the requirements of the executing task, the microarchitecture implementation of the cISA can be changed during runtime. For instance, a 5-stage pipeline implementation can be replaced by a faster 7-stage pipeline implementation as explained in Section 3.

Figure 3 illustrates an example for the different levels of adaptivity, using a task execution scenario in a sequence from a) to d). It shows how the execution pipeline, the cache, and the reconfigurable fabric may be *invaded* by different tasks (i.e. the resources are reconfigured towards the requirements of the task as explained in Section 3). Part a) of Figure 3 illustrates that the reconfigurable fabric can be used to accelerate OS functionality. This is especially beneficial, as the workload of the OS heavily depends on the behavior of the tasks, that is, 'when' and 'how many' system calls etc. will be executed. Therefore, providing static accelerators for the OS is not necessarily beneficial. Instead, the hardware may be reconfigured to accelerate other tasks in case the OS does not benefit from it at a certain time, as shown in part b) of the figure.


Legend:
ISA-independent Microarchitecture Adaptation
ISA-dependent OS-specific Adaptation
ISA-dependent Appl.-specific Adaptation
a)
7-stage Pipeline
OS Accelerator
Cache
b)
5-stage Pipeline
Guaranteed Appl.-specific Accelerator 1
Cache Extension
Cache
c)
5-stage Pipeline
Guaranteed Appl.-specific Accelerator 1
Temporary Accelerator 2
Scratchpad
d)
5-stage Pipeline
Guaranteed Appl.-specific Accelerator 1
Guarant. Acc. for second *i*-let
Scratchpad

Before a task starts executing, the Operating System needs to decide on which core it shall execute (task mapping and scheduling problem). Dedicated accelerators are loaded to support the corresponding algorithms

The task starts execution and an application-specific accelerator is established. Additionally, parts of the reconfigurable fabric are invaded to extend the capacity/ strategy of the cache. As the main computational load is now covered by the accelerator, the pipeline can be reconfigured to a 5-stage pipeline. This slows down its maximal frequency, but it reduces the penalty of wrong branch predictions.

In addition to the guaranteed accelerator (reserved during invasion) a second accelerator is established temporary and replaces the cache extension. Due to the reduced cache size, a user-managed scratchpad is more beneficial than a transparent cache, thus the cache logic is reconfigured correspondingly.

A second task starts execution and demands different accelerators. Therefore, the temporary established second accelerator for the first task is replaced to realize a new accelerator for the second task, i.e. it invades the available part of the reconfigurable fabric.

Figure 3: An example to demonstrate the adaptivity of the i-Cores, comprising ISA-independent adaptations as well as task-specific and OS-specific adaptations

Microarchitectural components like the cache and the processor pipeline can be adapted to different task requirements. Depending on the demanded time for performing these reconfigurations, the changes may be performed specific to the currently executing task or they may be performed for the set of tasks that execute on the i-Core. For instance, reconfiguring a 7-stage pipeline into a 5-stage pipeline only demands a few cycles and thus it can be performed as part of the task switch (in a preemptive multitasking system) when changing from one task to another. Instead, the reconfiguration time of the reconfigurable fabric is typically larger and changing the configuration as part of the task switch would increase the task switching time significantly. Therefore, to share the reconfigurable fabric among multiple tasks, it needs to be partitioned dynamically as indicated in parts c) to d) of Figure 3. Therefore, at different points in time, the temporary allocation ‘which part’ of the reconfigurable fabric accelerates ‘which task’ changes dynamically. Parts c) and d) show that a task can obtain a guaranteed share of the reconfigurable fabric, but it may use a larger share of it temporarily.

4. Modeling and Using Reconfigurable Special Instructions

4.1. Special Instruction Model

As motivated in Section 3, the reconfigurable fabric is shared among several tasks and thus, the ISE of a particular task has to cope with an at compile-time unknown size of the reconfigurable fabric. The approach of so-called *modular SIs* allows for providing different trade-offs between the amount of required hardware and the achieved performance by breaking SIs into elementary reconfigurable data paths (DPs) that are connected to implement an SI. It was originally developed in the scope of the RISPP project [22] and is meanwhile integrated and extended in other projects as well (e.g. KAHRIMSA [24]). The basic idea of modular SIs is illustrated in Figure 4 (description and an example follows) and is based on a hierarchical approach that – among others– allows to represent that

a) an SI may be realized by multiple SI implementations, typically differing in their hardware requirements and their performance (and/or other metrics), and
b) a DP is not dedicated to a specific SI, but it can be used as a part of different SIs instead.

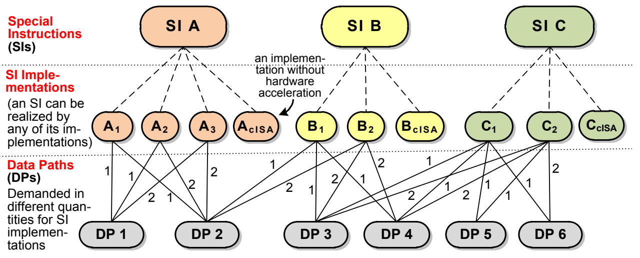

Special Instructions (SIs)
SI A
SI B
SI C
SI Implementations (an SI can be realized by any of its implementations)
an implementation without hardware acceleration
A1
A2
A3
AcISA
B1
B2
BcISA
C1
C2
CcISA
Data Paths (DPs) Demanded in different quantities for SI implementations
DP 1
DP 2
DP 3
DP 4
DP 5
DP 6

Figure 4: Hierarchical composition of SIs: multiple implementation alternatives exist per SI and demand data paths for realization; Figure based on [22]

Modular SIs are composed of DPs, where typically multiple DPs are combined to implement an SI. Figure 5 shows the Transform DP and the SAV (Sum of Absolute Values) DP to indicate their typical complexity. DPs are created and synthesized during compile time and they are reconfigured at run time. Technically, DPs are the smallest components that are reconfigured in modular SIs. An SI is composed of multiple DPs, as indicated in Figure 4 and illustrated with an example in Figure 5 and an SI implementation incorporates them in different quantities.

Figure 4 shows the hierarchy of SIs, SI Implementations, and DPs. At each point in time, a particular SI is implemented by one specific SI Implementation. It is noticeable that DPs can be used by different implementations of the same SIs and even by different SIs. This means that a DP can be shared among different SIs.

Each SI has one special Implementation that does not demand any DPs. This means that this SI implementation is not accelerated by hardware. If the SI shall execute but an insufficient amount of DPs needed to implement any of the other Implementations is available (i.e. reconfigured to the reconfigurable fabric), then the processor raises an ‘unimplemented instruction’ trap and the corresponding trap handler is used to implement the SI’s functionality [27]. The trap handler uses the core Instruction Set Architecture (cISA) of the processor. Therefore, this cISA implementation allows bridging the reconfiguration time (in the range of 1 to 10 ms), i.e. the time until the required DPs are reconfigured.

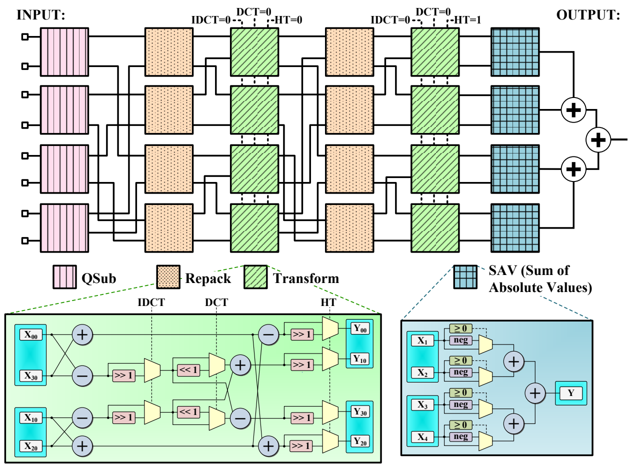


Figure 5: Example for the modular Special Instruction SATD (Sum of Absolute (Hadamard-) Transformed Differences); Figure based on [22, 27]

Figure 5 shows an example for a modular SI that is composed of four different types of DPs and that implements the Sum of Absolute Hadamard-Transformed Differences (SATD) functionality, as it is used in the Motion Estimation process of an H.264 video encoder. The data-flow graph shown in Figure 5 describes the general SI structure and each implementation of this SI has to specify how many *DP instances* of the four utilized *DP types* (i.e. QSub, Repack, Transform, and SAV) shall be used for implementation. Depending on the provided amount of DPs, the SI can execute in a more or less parallel way. When more DPs are available, then a faster execution is possible (that corresponds to a different implementation of same SI). Dynamically changing between these different performance-levels of an SI implementation allows reacting on changing application requirements or changing availability of the reconfigurable fabric.

4.2. Programming Interface for using the reconfigurable fabric

A task uses DPs on the reconfigurable fabric without the knowledge of low-level details such as partitioning the fabric for different SIs, determining the sequence of ‘which’ DPs are loaded ‘where’ on the fabric, etc. These steps are handled by the run-time system of the i-Core. Nevertheless, the programmer needs to trigger these steps by issuing system calls (presented in this section). The code fragment in Figure 6 (a high-level excerpt from the main loop of a H.264 video encoder) illustrates the programming model of the i-Core.

Microarchitectural features are adapted using the *set_i_Core_parameter* system call. It ensures that any preconditions for an i-Core adaptation are met (e.g. emptying the pipeline before changing pipeline length, invalidating cache-lines before modifying cache-management parameters, etc.) and it then performs the adaptations. For example, in Line 3 of Figure 6, the i-Core pipeline length is set to 5 stages and branch prediction is switched to a (2, 2) correlating branch predictor.

To request a share of the reconfigurable fabric, the task issues the *invade* system call (Line 5). *Invade* selects a share of a resource and grants it to the task. There, the resource is the entire reconfigurable fabric of the i-Core, a part of which is assigned to the task. The size of the assigned fabric depends on the speedup that the application is expected to exhibit. Generally, the more fabric is available to the task, the higher the speedup, but the expected speedup for a given amount of fabric is task-specific. This ‘speedup per size of assigned fabric’ relationship is explored during offline profiling and passed as the *trade_off_curve* parameter to the *invade* system call. During execution of *invade*, the run-time system will use the trade-off curve to decide which share of the fabric will be granted to the task. In the worst case, the application will receive no fabric at all (due to e.g. all fabric being occupied by higher priority tasks), then all SIs need their core ISA implementation (see Section 4.1) for execution The share of the fabric assigned to an application corresponds to the *my_fabric* return value in Line 5. Requesting a share of the fabric is typically done before the actual computational kernels start.

Next, the implementations of the SIs that will be used during the next kernel execution (the ‘while’ loop in the code example, lines 6-15) must be determined. The application programmer does not need to know which SI implementations are available at run time, as long as it is guaranteed that the SI functionality is performed (the SI implementation determines the performance of the SI, but not its functionality). The application informs the run-time system which SIs it will use during the next kernel execution and the run-time system selects implementations for these SIs. This is accomplished by means of the *invade* system call again (Line 7). Here, the invaded resource is the application’s own share of the reconfigurable fabric acquired earlier, which needs to be partitioned such that SI implementations for the requested SIs (SATD and SAD in the code example) fit onto it [29]. The programmer may also explicitly request specific DPs that shall be loaded into the fabric and the run-time system will consider these requests when selecting SI implementations for the task. This manual intervention may be used if the SI requirements of a kernel are rather static, i.e. the run-time system does not need to provide adaptivity for its implementation. Additionally, it can be used to limit the search for SI implementations and thus reduce the overhead of the run-time system.

```
1.  H264_encoder() {
2.      // Set i-Core microarchitecture parameters
3.      set_i_Core_parameter(pipeline_length=5,
            branch_prediction=2_2_correlation_predictor);
4.      // Invade a share of the reconfigurable fabric
5.      my_fabric=invade(resource=reconf_fabric,
                performance=trade_off_curve);
6.      while (frame=videoInput.getNextFrame()) {
7.          SI_implementations=invade(resource=my_fabric,
                SI={SAD, trade_off_curve[SAD],
                    execution_prediction[SAD_ME]},
                SI={SATD, trade-off_curve[SATD],
                    execution_prediction[SATD_ME]} );
8.          infect(resource=my_fabric, SI_implementations);
9.          motion_estimation(frame, ...);
10.         ...
11.         SI_implementations=invade(resource=my_fabric, ...);
12.         infect(resource=my_fabric, SI_implementations);
13.         encoding_engine(frame, ...);
14.         ...
15.     }
16. }
```

Figure 6: Pseudo code example for invading the reconfigurable fabric and the microarchitecture

After the run-time system has decided which SI implementations to use, the application can start loading the required DPs by issuing the *infect* system call (Line 8). DPs are loaded in parallel to the task execution, allowing the application to continue processing without waiting for the DPs to finish loading (which is a non-negligible amount of time; it is in the order of milliseconds). This implies that the *motion_estimation* function in the code example (Line 9) will start executing before the DPs have finished loading. Reconfiguring DPs in parallel to task execution provides a speedup for the following reason: if an SI is executed, but not all DPs for the desired implementation are available, the i-Core will use a slower implementation for the same SI that requires only the already loaded DPs (see Section 4.1). When additional DPs are loaded then faster SI implementations become available and they are used automatically.

After completing execution of a particular kernel (e.g. motion_estimation in Line 9), an entirely different set of SIs may be executed on the same share of the fabric attained by the task during its initial *invade* call. The fabric must, however, be prepared for execution of the new SIs (*invade* and *infect*, lines 11-13).

5. Results

In this section, a first evaluation of the i-Core is given, focusing on the application-specific ISE extensions for different tasks. The SIs for speeding up the tasks were developed manually to demonstrate the feasibility and the performance benefits of the i-Core. Figure 7 shows the speedup of four different tasks when accelerated by DPs on the reconfigurable fabric in comparison to executing the tasks without SIs. The obtained speedup depends on the size of the reconfigurable fabric that is expressed by the number of so-called Data Path Containers (DPCs), i.e. regions of the reconfigurable fabric into which a DPs can be reconfigured. For the results in Figure 7, each task uses the available number of DPCs on its own, i.e. the reconfigurable fabric is not shared among multiple tasks. Tasks like the CRC calculation require only few DPs for acceleration. With just one available DPC, a speedup of 2.51x is obtained for CRC. Further DPCs do not lead to further performance benefits for this task. The JPEG decoder approaches its peak performance after 5 DPCs. For more than 5 DPCs (3.49x speedup) only minor additional performance improvements are achieved (up to 3.81x). The image-processing library SUSAN and the H.264 video encoder achieve noticeable performance improvements of more than 18x each. Both tasks consist of multiple kernels that execute repeatedly after each other, i.e. the DPCs are consistently reconfigured to fulfill the task's requirements.

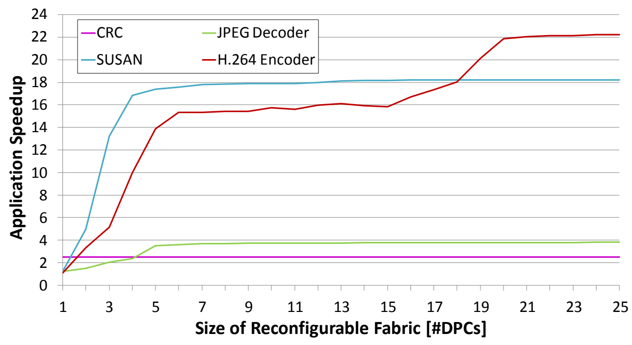


Figure 7: Speedup of different tasks (in comparison to execution without SIs) when executing on the i-Core in single-tasking mode (i.e. the entire reconfigurable fabric is available for the task)

Figure 8 shows the speedup of the same tasks as used in Figure 7, but here all tasks are executed at the same time, i.e. they need to share the available reconfigurable fabric (shown as the horizontal axis). Consequently, the performance improvement for a given number of DPCs is lower than the one shown in Figure 7, as not all available DPCs are assigned to one task. Altogether, five tasks execute, as two instances of the H.264 video encoder are executed at the same time. The figure shows that the characteristics of performance improvement is similar to the case where all tasks execute on their own, i.e. when they can utilize the entire reconfigurable fabric rather than sharing it. This demonstrates that it is possible and beneficial to share the reconfigurable fabric among the tasks.

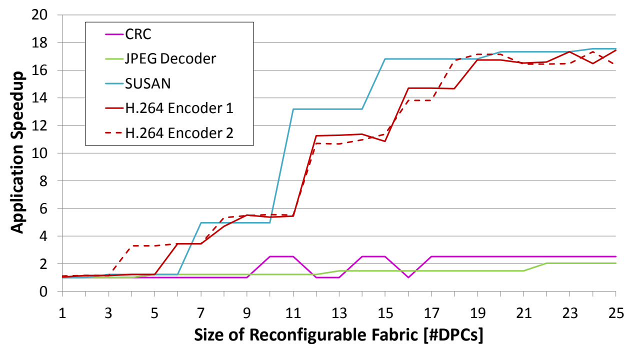


Figure 8: Speedup of different tasks (in comparison to execution without SIs) when executing on the i-Core in multi-tasking mode (i.e. the reconfigurable fabric is shared among all tasks)

6. Conclusion

This paper presented the i-Core concept, a reconfigurable processor that provides a very high adaptivity by utilizing a reconfigurable fabric (to implement Special Instruction) and a reconfigurable microarchitecture. The combination of an adaptive instruction set architecture and microarchitecture allows optimizing performance-wise relevant characteristics of the i-Core to task-specific requirements, which makes the i-Core especially beneficial in multi-tasking scenarios, where different tasks compete for the available resources. We evaluated the i-Core and demonstrated its conceptual advantages when several of these tasks execute together in a multi-tasking environment.

7. Acknowledgement

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89)

References

[1] J. Henkel, “Closing the SoC design gap”, *Computer*, vol. 36, no. 9, pp. 119–121, September 2003.

[2] K. Keutzer, S. Malik, and A. R. Newton, “From ASIC to ASIP: The next design discontinuity”, in *International Conference on Computer Design (ICCD)*. IEEE Computer Society, September 2002, pp. 84–90.

[3] J. Henkel and S. Parameswaran, *Designing Embedded Processors: A Low Power Perspective*. Springer Publishing Company, Incorporated, 2007.

[4] N. Clark, H. Zhong, and S. Mahlke, “Processor acceleration through automated instruction set customization”, in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2003, pp. 129–140.

[5] K. Atasu, L. Pozzi, and P. Ienne, “Automatic application-specific instruction-set extensions under microarchitectural constraints”, in *Proceedings of the 40th annual Conference on Design Automation (DAC)*, June 2003, pp. 256–261.

[6] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, “A scalable application-specific processor synthesis methodology”, in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, November 2003, pp. 283–290.

[7] N. Cheung, J. Henkel, and S. Parameswaran, “Rapid configuration and instruction selection for an ASIP: a case study”, in *IEEE/ACM Proceedings of Design Automation and Test in Europe (DATE)*, March 2003, pp. 802–807.

[8] K. Compton and S. Hauck, “Reconfigurable computing: a survey of systems and software”, *ACM Computing Surveys (CSUR)*, vol. 34, no. 2, pp. 171–210, June 2002.

[9] F. Barat and R. Lauwereins, “Reconfigurable instruction set processors: A survey”, in *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP)*, June 2000, pp. 168–173.

[10] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*. Springer Publishing Company, Incorporated, 2007.

[11] H. P. Huynh and T. Mitra, “Runtime adaptive extensible embedded processors – a survey”, in *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, July 2009, pp. 215–225.

[12] T. v. Sydow, B. Neumann, H. Blume, and T. G. Noll, “Quantitative analysis of embedded FPGA-architectures for arithmetic”, in *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, September 2006, pp. 125–131.

[13] B. Neumann, T. von Sydow, H. Blume, and T. G. Noll, “Design flow for embedded FPGAs based on a flexible architecture template”, in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, 2008, pp. 56–61.

[14] M. Huebner, P. Figuli, R. Girardey *et al.*, “A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture”, in *Proc. of Reconfigurable Architectures Workshop (RAW)*, May 2011.

[15] S. Vassiliadis, S. Wong, G. Gaydadjiev *et al.*, “The MOLEN polymorphic processor”, *IEEE Transactions on Computers (TC)*, vol. 53, no. 11, pp. 1363–1375, November 2004.

[16] R. Wittig and P. Chow, “OneChip: an FPGA processor with reconfigurable logic”, in *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996, pp. 126–135.

[17] J. A. Jacob and P. Chow, “Memory interfacing and instruction specification for reconfigurable processors”, in *Proceedings of the ACM/SIGDA 7th international symposium on Field Programmable Gate Arrays (FPGA)*, February 1999, pp. 145–154.

[18] J. E. Carrillo and P. Chow, “The effect of reconfigurable units in superscalar processors”, in *Proceedings of the international symposium on Field Programmable Gate Arrays (FPGA)*, February 2001, pp. 141–150.

[19] R. Lysecky, G. Stitt, and F. Vahid, “Warp processors”, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, no. 3, pp. 659–681, June 2006.

[20] M. Dales, “Managing a reconfigurable processor in a general purpose workstation environment”, in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2003, pp. 980–985.

[21] E. Lübbers and M. Platzner, “ReconOS: An RTOS supporting hard- and software threads”, in *International Conference on Field Programmable Logic and Applications (FPL)*, August 2007, pp. 441–446.

[22] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, “RISPP: Rotating Instruction Set Processing Platform”, in *Proceedings of the 44th annual Conference on Design Automation (DAC)*, June 2007, pp. 791–796.

[23] L. Bauer, M. Shafique, and J. Henkel, “Efficient resource utilization for an extensible processor through dynamic instruction set adaptation”, *IEEE Transactions on Very Large Scale Integration Systems (TVLSI), Special Section on Application-Specific Processors*, vol. 16, no. 10, pp. 1295–1308, October 2008.

[24] R. König, L. Bauer, T. Stripf *et al.*, “KAHRISMA: A novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture”, in *Proceedings of the 13th conference on Design, Automation and Test in Europe (DATE)*, March 2010, pp. 819–824.

[25] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel, “mRTS: Run-time system for reconfigurable processors with multi-grained instruction-set extensions”, in *Proceedings of the 14th conference on Design, Automation and Test in Europe (DATE)*, March 2011, pp. 1554–1559.

[26] SPARC International, Inc., “The SPARC architecture manual, version 8”, http://www.sparc.org/specificationsDocuments.html#V8, http://gaisler.com/doc/sparcv8.pdf.

[27] L. Bauer, M. Shafique, and J. Henkel, “A computation- and communication- infrastructure for modular special instructions in a dynamically reconfigurable processor”, in *18th International Conference on Field Programmable Logic and Applications (FPL)*, September 2008, pp. 203–208.

[28] J. Teich, J. Henkel, A. Herkersdorf *et al.*, “Invasive computing: An overview”, in *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds. Springer, Berlin, Heidelberg, 2011, pp. 241–268.

[29] L. Bauer, M. Shafique, and J. Henkel, “Run-time instruction set selection in a transmutable embedded processor”, in *Proceedings of the 45th annual Conference on Design Automation (DAC)*, June 2008, pp. 56–61.

[30] R. I. Bahar and S. Manne, “Power and energy reduction via pipeline balancing”, in *Proceedings of the International Symp. on Computer architecture (ISCA)*, 2001, pp. 218–229.

[31] S. Ghiasi, J. Casmira, and D. Grunwald, “Using IPC variation in workloads with externally specified rates to reduce power consumption”, in *Workshop on Complexity Effective Design*, 2000.

[32] T. Juan, S. Sanjeevan, and J. J. Navarro, “Dynamic history-length fitting: a third level of adaptivity for branch prediction”, in *Proceedings of the international symposium on Computer architecture (ISCA)*, 1998, pp. 155–166.

[33] D. H. Albonesi, “Selective cache ways: on-demand cache resource allocation”, in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, 1999, pp. 248–259.

[34] S. Kaxiras, Z. Hu, and M. Martonosi, “Cache decay: exploiting generational behavior to reduce cache leakage power”, in *Proceedings of the 28th annual international symposium on Computer architecture (ISCA)*, 2001, pp. 240–251.

[35] A. V. Veidenbaum, W. Tang, R. Gupta *et al.*, “Adapting cache line size to application behavior”, in *Proceedings of the intl. conference on Supercomputing (ICS)*, 1999, pp. 145–154.

[36] F. Nowak, R. Buchty, and W. Karl, “A run-time reconfigurable cache architecture”, in *International Conference on Parallel Computing: Architectures, Algorithms and Applications*, 2007, pp. 757–766.

[37] J. Tao, M. Kunze, F. Nowak *et al.*, “Performance advantage of reconfigurable cache design on multicore processor systems”, *International Journal of Parallel Programming*, vol. 36, no. 3, pp. 347–360, 2008.

Advanced Profiling of Applications for Heterogeneous Multi-Core Platforms

Koen Bertels, S. Arash Ostadzadeh, Roel Meeuws
Computer Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, Delft, the Netherlands
Email: {K.L.M.Bertels,S.A.Ostadzadeh,R.J.Meeuws}@tudelft.nl

Abstract—**The increased complexity of programming on multiprocessors platforms requires more insight into program behavior, for which programmers need increasingly sophisticated methods for profiling, instrumentation, measurement, analysis, and modeling of applications. Particularly, tools to thoroughly investigate the memory access behavior of applications have become crucial due to the widening gap between the memory bandwidth/latency compared to the processing performance. To address these challenges, we developed the *Q²* profiling framework in the context of the Delft Workbench (DWB), which is a semi automatic tool platform for integrated hardware/software co-design, targeting heterogeneous computing systems containing reconfigurable components. The profiling framework consists of two parts, a static part which extracts information related to memory accesses during the execution of an application. We present an advanced memory access profiling toolset that provides a detailed overview of the runtime behavior of the memory access pattern of an application. This information can be used for partitioning and mapping purposes. The second part involves a statistical model that allows to make predictions early in the design phase regarding memory and hardware usage based on software metrics. We examine in detail a real application from the image processing domain to validate and specify all the potentials of the *Q²* profiling framework.**

I. Introduction

As computer manufacturers move increasingly beyond the traditional single-core platforms, system developers are confronted with an increasing number of complex architectures. There has already been great proliferation of multi-core architectures and reconfigurable devices. Currently, the trend in research is to utilize an increasing number of cores (many-core) and to mix different types of processing elements. These elements include GPPs, reconfigurable devices, GPUs, and ASICs, just to name a few. Such architectures not only require new tool-chains, libraries, and interconnects, among others, but also demand more insight into program behavior in order to take advantage of the characteristics of these heterogeneous systems. As a result, it is increasingly important to support developers in their analysis and understanding of different platforms and application behaviors. Particularly, tools to thoroughly investigate the memory access behavior of applications become crucial due to the widening gap between the memory bandwidth/latency and the processing performance. Furthermore, these heterogeneous architectures involve development of hardware blocks that can take much time to develop. Even in the case of automatic generation of the hardware, the actual synthesis of the design can take a long time. As a consequence, there is a clear need to have early and fast predictions of the hardware cost of the different parts of an application.

These two challenges are the main concern of the Q^2 profiling framework developed in the context of the Delft Workbench (DWB) [1]. DWB is a semi-automatic tool platform for integrated hardware/software co-design, targeting heterogeneous computing systems containing reconfigurable components. The profiling framework consists of two parts: a static part, which can provide estimates of some critical metrics in a small amount of time, and a dynamic part, which provides accurate measurements of some of those metrics.

The aim of this paper is to demonstrate the need for and the usage of such profiling tools. This will be done using a case study of a well known image processing application. The main contributions of this paper are the following:

- the introduction of the Q^2 Profiling Framework,
- a case study of the Q^2 framework on the Canny Edge Detection application.

The paper is structured as follows. First, Section II evaluates the related research and establishes the niche for our framework. Then, in Section III the direct research context of Q^2 is presented. The main exposition of the Q^2 framework can be found in Section IV. Subsequently, we present the case study of the Canny Edge Detection application in Section VII. Finally, Section VIII concludes the paper.

II. Related Work

As heterogeneous computing systems become increasingly more complex and demanding, utility tools turn out to be essential in assisting application developers to analyze and optimize the performance of applications. Developers require these tools to improve existing applications or drive the development of new applications for heterogeneous reconfigurable systems. Profiling tools focus on finding critical code regions and provide hints for optimizations. Critical code regions are conventionally associated with computational hot-spots or communication bottlenecks of an application. Potential optimizations for such code regions include: code revisions (memory usage reduction, FPGA area optimization, etc.), functional merging/splitting, or HW/SW partitioning; which all first need the identification of the critical region(s).

Dynamic profilers, in contrast with *static profilers*, examine an application during execution to provide information about the run-time behavior of the application. Many dynamic profiling tools have been developed so far. General profilers, such as *gprof* [2], normally provide profiling information at function-level granularity. Furthermore, they do not distinguish between computation and communication times. On the other hand, there are some profilers that focus on more detailed levels, such as, statement-level, block-level, or loop-level [3]. Some profilers, such as Intel's *vTune* [4] and AMD's *CodeAnalyst* [5], integrate hardware event monitoring in addition to software-based sampling or instrumentation techniques. There also exist profiling tools, which are specific for particular architectures. As an example, SpixTools [6] is a collection of programs that allow profiling of applications at different levels of granularity for the Sun Microsystems SPARC architectures. *QPT* [7], is dynamic profiler and tracing system that rewrites the executable file of an application by inserting extra code to record the execution frequency or sequence of each basic block. The execution cost of functions in the program can be extracted from this information. Unlike *gprof*, *QPT* records the exact execution frequency, not a statistical sample. When tracing a program, *QPT* produces a trace regeneration program that reads the highly compressed trace file and regenerates a full program trace.

Apart from general profilers, which mostly aim to provide execution time related information, there is a different class of profilers that focus on resources, such as the memory system, rather than on computation. MemSpy [8] instruments applications with Tango [9], an execution-driven simulator, by putting calls to the memory simulator for each memory reference associated with dynamically allocated memory or explicitly-identified address ranges. The CPROF [10] system is a cache performance profiler that annotates source code to identify the source lines and data structures that cause frequent cache misses. The Memory Trace Visualizer (MTV) [11] is a tool that provides interactive visualization and analysis of the sequence of memory operations performed by a program as it runs. It uses visual representations of abstract data structures, a simulated cache, and animating memory operations. *ProfileMe* [12] takes samples of instructions as they move through an out-of-order issue pipeline and provide some statistical reports, such as, cache miss rates. *Cacheprof* [13] is a memory simulator that annotates each memory access instruction and links a cache simulator into the resulting executable. During the execution, all data references are intercepted and sent to the simulator. It is able to report on the number of memory references and the number of misses for each line of the source code.

When mapping candidate regions onto reconfigurable components, detailed evaluation of the required resources, power consumption, and speed-up are also essential. Over the years, many hardware performance estimation schemes have become available. Part of these schemes drive low-level design processes [14], [15]; for example, in the placement and route phases [16]. However, in the early stages of design, normally, there are no low level description available and other more high-level estimation schemes are required. Therefore, others, including our estimation scheme, operate on a HLL description such as C [17], [18], [19]. These approaches allow for early estimation of resources, and as such can help designers to tailor their code for heterogeneous platforms at an early stage. However, most of the high-level methods focus either on a specific application domain [18] or on a specific kind of design [19], [20], while other high-level methods are only applicable to certain types of platforms [17], [14]. For example, [18] is tailored for the SA-C language, which is a restricted C-dialect for the Multimedia domain. Furthermore, this approach was embedded in the SA-C compiler and, as such, cannot be simply assumed to be valid for other compilers or platforms. These issues reduce the applicability of this approach for more general use.

Most works base their claims on the quality of their models in terms of error on validation sets containing between four [18] to twelve [19] kernels. *However, if the target is to report errors that are not biased to a small data set, a larger set of validation data, such as the one used in this paper, becomes vital.* Let's suppose, for example, one validates a model using a set of only 12 kernels. It is very unlikely that these kernels can represent the whole spectrum of possible kernels. For one, with such a limited set of kernels the possibility of cherry-picking your validation set increases. Secondly, an anomalous kernel that is not well modeled by a certain model can adversely affect the validation error when one uses a small set of kernels.

III. The Research Context

The Q^2 profiling framework is developed in the context of the Delft Workbench (DWB) [1]. The DWB is a semi-automatic tool platform for integrated hardware/software co-design, targeting heterogeneous computing systems containing reconfigurable components. It targets the Molen machine organization [21], an architectural template for heterogeneous reconfigurable platforms developed at the Delft University of Technology. This architectural template adheres to the Molen programming paradigm [22]. This paradigm is a sequential consistency paradigm for programming reconfigurable machines.

The DWB addresses the entire design cycle from profiling and partitioning to synthesis and compilation of an application and it focuses on four main steps within the entire heterogeneous system design, namely:

- *the code profiling and the cost modeling* [23], [24], [25];
- *the graph transformations and optimizations* [26], [27], [28];
- *the retargetable compiler* [29]; and
- *the VHDL generation*[30].

For a given application, *code profiling and cost modeling* identify which parts of the application are good candidates for hardware implementation. This decision takes into consideration the available hardware resources and the speed-up provided by the hardware implementation of the application, or parts of it, versus a software implementation. *Graph transformations and optimizations* analyze the candidate parts of the application for hardware implementation to find out if the code segments can be clustered/partitioned according to various targets as, for example, hardware resource sharing. Next, an *optimization* phase is performed to spot parallelization opportunities.

After making the decision of which parts of the code segments to implement in hardware, the code is annotated. Subsequently, the *retargetable compiler* generates the new object code, which contains the call to the reconfigurable hardware for the selected segments of code. Finally, the identified instructions (code segments) pass through a *VHDL generation* phase, which generates hardware description of the instructions.

IV. The Q^2 Profiling Framework

The focus of this work is on code profiling. Figure 1 depicts in detail the Q^2 profiling framework, which is part of the DWB platform. We distinguish between static and dynamic profiling paths. Static profiling can provide estimates in a small amount of time, whereas dynamic profiling provides accurate measurements of several aspects, such as, execution time estimates and memory access intensity of individual functions in an application. The static profiling path first extracts code characteristics from the application source code. These characteristics, or software complexity metrics, are then used by the *Quipu* model to make fast and early predictions of components with specific implementation details like FPGA area or speed-up. Based on the code characteristics and *Quipu* estimates, a conjecture is made on which kernels are interesting for further analysis regarding hardware implementation. The dynamic profiling path focuses on run-time behavior of an application and, therefore, is not as fast as the static profiling. Furthermore, the dynamic profiling requires the application binaries and representative input data in order to provide relevant measurements. This, in itself, can raise some concerns, if the application acts quite distinctively in different circumstances. However, that rarely happens in practice. The common *gprof* [2] profiler is utilized to identify application hot-spots in terms of execution cost and to recognize frequently executed functions. *gprof* also provides execution timing estimates, which can only be useful for applications running for relatively large period of time. On the other hand, *MAIP* provides accurate measurements for the contribution percentage of individual functions with respect to the whole execution cost of the application. It also distinguishes between memory access related and computation related operations. The *QUAD* [23] core module provides a comprehensive quantitative analysis of memory access behavior of an application with the primary goal of detecting actual data dependencies at the function-level. The tracing modules implemented in the *QUAD* core is utilized in the *cQUAD* tool to reveal the data communication pattern of a pair of cooperating functions in an application. The *xQUAD* [31] tool provides detailed, fine-grained memory access information for each function in the application. The information includes runtime memory usage statistics regarding individual data objects defined in an application source code. The *tQUAD* [24] tool reveals the memory bandwidth usage of each function in terms of relative execution timings.

V. The *Quipu* Modeling Approach

The static profiling part in this paper consists of the *Quipu* modeling approach, which we have developed and presented in [32], [33], and [34]. The approach is generic and not limited to

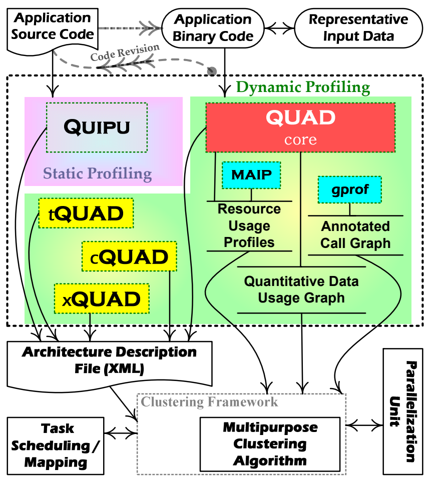


Fig. 1. The Q^2 Profiling framework within the DWB tool platform.

any particular platform or tool-chain by allowing the generated models to be recalibrated for different tools and platforms, contrary to the majority of the existing techniques. Furthermore, a major strength of *Quipu* models is their linear nature. Although, the statistical techniques to create the models may be very time-consuming, the resulting prediction model requires only a number of multiplications in addition to parsing of the source code. This allows for integration of *Quipu* models in highly iterative design processes where estimates are recalculated many times in a short time period. Additionally, as *Quipu* models are based on measurements from C code, very early predictions become possible, allowing designers to take important decisions on hardware mapping at an earlier stage.

The *Quipu* models that we utilize in this paper are generated for a combination of the DWARV C-to-VHDL compiler [35] and the Xilinx ISE Synthesizer for the Virtex 5 FX 130T FPGA. Notwithstanding, *Quipu* can calibrate models for other combinations as well, such as, an Altera Stratix IV FPGA combined with the C-to-Verilog compiler from the Haifa University [36], [37]. The output data of a combination of tools and platform is used to calibrate a specific model instance. Because the set of calibration data can vary depending on certain tool options, the user may want to consider generating models for each option value. For example, area measurements will be much lower when optimizing for area compared to when optimizing for speed.

In the following, we first define the models and the criteria. Subsequently, we discuss the key components of the *Quipu* approach. Last but not least, the newly developed techniques that constitute our approach are discussed in detail.

A. The Models and the Criteria

It is essential to quantify the characteristic aspects of the software description at hand, when we consider the modeling of hardware from software descriptions. In [33] and [34], we have introduced Software Complexity Metrics (SCMs) as a way to address this. SCMs are indicators of specific characteristics of the software code. Examples of SCMs are the number of operators, the number of loops, or the cyclomatic complexity, but also more complex metrics involving data-flow analysis and the attribution of operations to functional units. Currently, we use a set of 58 SCMs as a base for our model. We refer the reader to our previous work, for more information on many of the implemented SCMs. [33], [32]

To characterize hardware performance in terms of area, interconnect, power, and other parameters, we measure and predict 49 different hardware performance indicators. For instance, the number of slices, the number of wires, the number of LUTs, and the number of clock wires. Most of these parameters are related to interconnect and provide specific information on wires, nets, route-throughs, or switch-boxes, further subdivided for logic, power, or clock resources.

Given these criteria, we look for a model that describes the relation between hardware and software:

$$y_{HW} = g(\bar{\mathbf{x}}_{\mathbf{SCM}}) + \epsilon. \tag{1}$$

This is the theoretical optimal model relating some hardware metric y_{HW} to a vector of SCMs $\bar{\mathbf{x}}_{\mathbf{SCM}}$ with the ideal relation $g(\cdot)$ and some error ϵ inherent to the problem at hand. In practice, an ideal model cannot be found. Instead, any modeling scheme is an approximation to some level. Therefore, we model the relation $g(\cdot)$ with an approximated relation $\hat{g}(\cdot)$. This results in the introduction of some estimation error $\hat{\epsilon}$ inherent to our approximation scheme:

$$\hat{y}_{HW} = \hat{g}(\bar{\mathbf{x}}_{\mathbf{SCM}}) + \hat{\epsilon}. \tag{2}$$

The approximation $\hat{g}(\cdot)$ can be, for example, an ad-hoc model, a Linear Regression Model (LRM), or an Artificial Neural Network (ANN). In case of LR techniques, $\hat{g}(\cdot)$ is a linear equation:

$$\hat{y}_{HW} = \hat{\mathbf{a}}\bar{\mathbf{x}}_{\mathbf{SCM}} + \hat{b} + \hat{\epsilon}, \tag{3}$$

where $\hat{a}$ is a vector of estimated coefficients $\hat{a}_i$ corresponding to element x_i of the vector of SCMs $\bar{\mathbf{x}}_{\mathbf{SCM}}$ obtained for some kernel that corresponds to the estimated hardware metric $\hat{y}$, and $\hat{b}$ is the offset of the linear model. Note, that these variables are stochastic variables. *This means that reporting a simple percentage error is not enough. As a result, the characterization of the error distribution must be addressed as well.*

B. The Tools and the Kernel Library

Quipu consists of a set of tools and a kernel library, as depicted in Fig. 2. In the modeling flow, *Quipu* extracts SCMs and hardware performance indicators from a kernel library. *This is a library of 178 kernels from a wide variety of application domains, contrary to many existing techniques, which use libraries of tens of kernels at most.* This allows us to build models that are generally applicable. It is also possible to build domain-specific models by using, for example, only the 57 Cryptography-related kernels out of the 178 kernels considered. An overview of the kernels in this library is provided in Table I.

Regarding the tools in the *Quipu* approach, we have:

Domain	Kernels	Size[a]	Bit-Based	Streaming	Control
Compression	12	47.6 (14-95)	x		x
Cryptography	57	192.2 (15-1107)	x	x	some
DSP	12	32.3 (10-110)	x	x	some
ECC	13	74.8 (10-496)	x	x	x
Mathematics	29	33.9 (5-100)			
Multimedia	42	81.8 (6-1107)	some	x	x
General	13	72.9 (22-163)			x
Total	178	102.6 (5-1107)			

[a]avg. size in number of statements (range).

TABLE I
OVERVIEW OF THE KERNEL LIBRARY, THE NUMBER OF KERNELS AND THEIR MAIN ALGORITHMIC CHARACTERISTICS IN EACH APPLICATION DOMAIN.

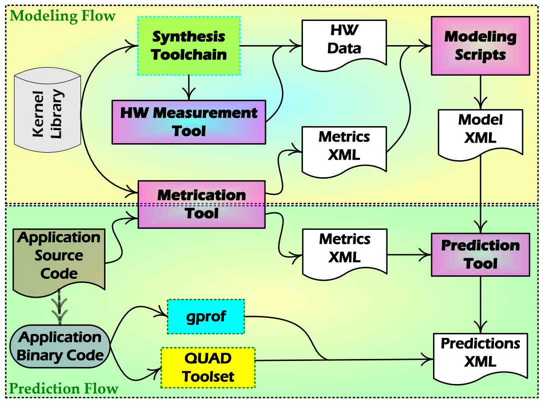


Fig. 2. An overview of the *Quipu* modeling approach, its tools (the boxes with thick border), and the accompanying tools (the boxes with dashed borders).

1) The Metrication tool: The 58 different SCMs can be extracted using the *Quipu Metrication tool*, which produces an XML file containing SCM measurements for each kernel. This tool is written as an engine in the CoSy compiler system [38] developed by ACE Associated Compiler Experts. This comprehensive compiler contains a large set of optimizations and is easily extensible by writing engines that can be plugged into the framework. Using the existing engines of the CoSy framework, different optimizations can be performed before measuring the SCMs. This flexibility allows to choose specifically a set of optimizations that fit a target tool-chain, which in turn adds to the generic character of the *Quipu* modeling approach. Apart from being an essential part of the *Quipu* modeling framework, the metrication tool can also be useful within an Software Measurement framework that helps drive management of software development processes.

2) The Hardware Measurement tool: The *Quipu hardware Measurement tool* measures 49 different hardware parameters, such as area and interconnect, from synthesized hardware targeted at Xilinx FPGAs. The tool keeps track of all nets and components in the design by processing the XDL file generated by the Xilinx synthesis tool-chain. It provides detailed interconnect measurements, such as the number of clock wires or the number of power nets. For Altera FPGAs, such a tool is not needed, as the report files generate detailed numbers on the interconnect usage automatically.

3) The Modeling Scripts and the Prediction tool: The gathered SCMs and hardware measurements are run through a set of modeling scripts that automatically evaluate different modeling techniques. The output model XML file can be used in the prediction flow, where, based on SCM inputs, the *Quipu prediction tool* provides estimates of any required hardware aspects. All intermediate files in the prediction flow are saved in XML format for an easy integration. For example, the results of execution and memory profiling tools might be integrated as depicted in Fig. 2.

C. *Utilization of* Quipu *models*

The *Quipu* models can be used in several contexts. In the first place, developers can use the predictions for function kernels they are working on in order to find problems with potential placement on hardware. If a kernel is predicted to have a large amount of slices on a particular platform, the developer might move to split the function in several parts, or address the root cause for the large area (e.g. a large local array). Secondly, a tailored *Quipu* model may drive the optimization pass in a hardware compiler by predicting the hardware size of the contained basic blocks at different unroll factors. In this way, the hardware compiler can automatically choose a beneficial unroll factor. Thirdly, a high-level HW-SW partitioning algorithm could use a *Quipu* model to evaluate many different partitionings at an early stage of the development.

VI. QUAD Dynamic Memory Profiling Toolset

Nowadays, Dynamic Binary Instrumentation (DBI) is gaining popularity among the available methods for intercepting memory accesses. DBI can be used to develop dynamic profilers. The dynamic part of the Q^2 profiling framework falls under this category of profilers. The tools are implemented as Dynamic Binary Analysis (DBA) tools using the Pin [39] DBI platform. In this section, we first present a brief overview of the Pin DBI platform and then describe the individual tools developed in the dynamic profiling framework of the DWB and discuss their potentials.

A. *Pin Dynamic Binary Instrumentation*

Run-time instrumentation is a technique for injecting extra user defined code into an application during its execution to study the behavior of the application. The primary advantage of this kind of instrumentation is that, in order to profile an application, we only need the binary executable code of the application. Furthermore, Pin adopts the dynamic compilation technique that uses a Just-In-Time (JIT) compiler to (re)compile and instrument the application code on the fly. This capability provides the benefits of portability, transparency and efficiency to the end user. In summary, Pin supplies a fast instrumentation system, which is able to work with unmodified executables in addition to the preservation of the application's original uninstrumented behavior.

Pin provides a DBI platform for building a variety of DBA tools for multiple architectures, namely, the IA32, 64bit x86, Itanium®, and ARM architectures. A primary key feature of the Pin DBI platform is instrumentation transparency. It means that Pin preserves the original behavior of the application. As a result, the application uses the same addresses (both instruction and data) and the same values (both register and memory) as it would do in an uninstrumented execution. Furthermore, Pin does not modify the application stack, as some applications may deliberately reference memory addresses beyond the top of the stack.

Generally, two types of routines are defined in Pin-based tools, namely, *instrumentation* routines and *analysis* routines. Instrumentation routines determine where, in the application code, to place calls to analysis routines. Analysis routines are customizable by the user and they are called while the program executes. The arguments to analysis routines can be, for example, the instruction pointer, the effective memory address of the instruction, the memory or stack value, the address of branch instruction, the system calls values, and others. The actual instrumentation is performed by the JIT compiler. Pin intercepts the very first instruction of the application and recompiles the executable generating *basic blocks* code starting at that instruction, and instrumenting the code according to the specified instrumentation type. The generated code sequence is almost identical to the original one, except that it runs under the control of Pin. When a branch exits a basic block, Pin generates more basic blocks code for the branch target and it continues the execution. The JIT generated code and its instrumentation are saved in a code cache for future execution of the same sequence of instructions to improve performance.

Instrumentation with Pin can be done at different levels of granularity. The finest level is instrumentation at the *instruction level*, i.e. instrumenting the application one instruction at a time. It is also possible to instrument code at the *trace level*[1], at the *procedure level*, and at the entire *image level*.

The execution of an instrumented application usually shows a considerable slowdown. This depends on the nature of the instrumented application, as well as on the overhead caused by the analysis routines in the tool. It appears that most of the slowdown is caused by the execution of the code, rather than on-the-fly code compilation (which includes the insertion of the instrumentation code). In Pin, some performance improvements are done during the compilation phase of the application. Improvements are on register reallocation, inlining, liveness analysis, and instruction scheduling. This results in an instrumented code, which run very fast compared to other DBI platforms.

B. *The QUAD Tools*

The *QUAD* toolset consists of several related tools developed to demonstrate a comprehensive overview of the memory access behavior of an application, as well as, to provide fine-grained detailed memory access related statistics. The *QUAD* core module has been designed and implemented as the primary component to provide useful quantitative information about

[1]A trace is defined as a straight-line sequence of instructions executed sequentially. Pin guarantees that traces only enter at the top, but may have multiple exits.

the data dependence between any pair of cooperating functions in an application. Data dependence is estimated in the sense of producer/consumer bindings. More precisely, *QUAD* core reports which function is consuming the data produced by another function. The exact amount of data communication and the number of Unique Memory Addresses (UnMA) used in the communication process are also calculated. The *QUAD* core contains a fast and efficient Memory Access Tracing (MAT) module, which detects and traces all the memory references made during an application execution. It acts as a kind of shadow memory for the whole memory access space. Considering the fact that *MAT* structures the shadow memory in such a way to make tracing as fast as possible, the size of the space overhead can be very huge. The tracing mechanism is also utilized by *cQUAD* tool to monitor in detail each and every data transfer event occurring between a pair of communicating functions. This data communication can be viewed as a dedicated virtual channel for transferring data items from the producer side to the consumer end. The Data Communication Channel Pattern Detector (DCCPD) module thoroughly analyzes the extracted raw profile data to compute several critical metrics that can classify and describe the pattern of the communication between the two functions. These metrics include the interleaving balance factor, spatial/temporal localities and data communication pattern complexity. In general, these information is required to reveal the coupling intensity and regularity between the communicating functions in an application. It can be very useful in understanding the behavior of the functions with regard to their data dependencies and requirements, as well as, in mapping and scheduling potential parts of the application onto reconfigurable devices.

The *xQUAD* extension to the *QUAD* toolset augments the memory access analysis of the application by providing a very detailed, fine-grained intra-function information. The information is provided based on the application source code data object granularity. In this respect, the programmers can utilize the information to fine-tune the application behavior regarding it memory access references. The main motivation for this extension is that a coarse view of the intra-function data access makes it difficult for the application developers to attribute the extracted information to particular user-defined data objects in an application at the source code level. Hence, fine-grained code revision/optimization becomes a burdensome task. The Pin DBI framework does not provide API functions for retrieving data objects information. Therefore, source-level information about data objects should be extracted directly from the binary file(s). The *DWARF* module is utilized to extract low-level source code representations, such as, information about source code types, function and object names, and line numbers. *tQUAD* is another component in the *QUAD* toolset. It extracts the timing information of the functions, as well as, their memory bandwidth usages during the application execution. The information extracted by *tQUAD* can lead to the recognition of the main *execution phases* within the application, which in the end can be used to identify the related functions in each phase. Each execution phase ordinarily corresponds to a specific and well-defined (sub)task within the application process, which is based on the high-level algorithm of the application source code. The logical steps of the algorithm are clear for the programmers. However, the information provided by *tQUAD* can help users to have a more solid vision of what are the steps the application actually takes to perform its task. In addition, this information can also be regarded as valuable hints for the application developers to understand the (re)usability scope of the functions defined in the application, and probably serve as further optimization directives. As an example, a general function can be used in different phases of the application, which makes it a good candidate for replacement with several specific-purpose tailored functions.

Figure 3 illustrates the architectural overview of the QUAD toolset along with the Pin DBI components. At the highest level in the Pin software layer, there is a Virtual Machine (VM), a code cache, and an instrumentation API. The VM consists of a JIT compiler, an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute the application. The JIT compiles and instruments the application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering (leaving) the VM from (to) the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly. It is used for system calls that require special handling from the VM. Since Pin does not reside in the kernel of the operating system, it can only capture user-level code.

The main component inside the *QUAD* core is the MAT module, which is responsible for building and maintaining the dynamic trie [40] data structure to provide relevant memory access tracing information. As Figure 3 shows, three binary objects are present when an instrumented application is running: the to-be-profiled application, Pin, and the *QUAD* toolset. Pin is the engine that instruments the application. The *QUAD* toolset contains instrumentation and analysis routines and it is linked with a library that allows *QUAD* to communicate with Pin.

To the best of our knowledge, *QUAD* is the first profiling toolset that focuses on examining the run-time memory access behavior of an application and on providing some valuable quantitative information. Even though *QUAD* toolset can be employed to spot coarse-grained parallelism opportunities in an application, it practically provides a more general-purpose facility that can be utilized in various heterogeneous reconfigurable systems optimizations by estimating effective memory access related parameters.

MAIP. The Memory Access Intensity Profiler (*MAIP*) is developed as a standalone Pin-based DBA profiler. *MAIP* aims to provide some unprocessed basic data for each function in an application with the main objective of revealing the intensity of memory access operations. Furthermore, *MAIP* can act as an enhanced alternative to the traditional *gprof* profiler, allowing accurate measurements for applications with a small running time. The problem with *gprof* is that the derived run-time figures are based on a sampling process. As a result, besides being subject to statistical inaccuracy, if a function runs for only a

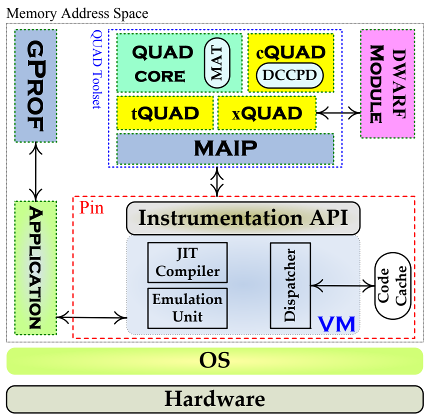


Fig. 3. Architectural overview of the dynamic part of the Q^2 profiling framework.

small amount of time, there is a pretty good chance that the profiler actually overlooks that function in action. Only if the total run-time of an application is large enough, a small run-time value for a function discloses that the function uses an insignificant fraction of the application's whole execution time. Otherwise, no valuable information can be gained from the *gprof* analysis at all. The extracted raw data by *MAIP* can be further processed to get some valuable information, such as, the ratio of memory access instructions to the total executed instructions in a program.

The basic data that are measured by the profiler are classified into three main classes: instructions, operands, and bytes. Each class is subsequently divided into Read/Write and Stack/Non-stack sub categories. A variety of parameters can be computed to measure the intensity of the memory access operations within an application by using a precise memory access tracking module, which intercepts and inspects every memory access instruction for detailed information. Specifically, *MAIP* extracts the following data for each function:

- *The total number of instructions executed within each function call.*
  If there is more than one call instance for a particular function, the aggregate value of this parameter is used in subsequent processing to have a more accurate estimation.
- *The total number of memory access instructions executed within each function call.*
  As before, if there is more than one call instance for a particular function, the aggregate value of this parameter is used in subsequent processing.
- *Memory Access Ratio (MAR)*
  This is the percentual ratio of the total number of memory access instructions to the total number of instructions.
- *NLOC-MAR*
  This is similar to the previous parameter. However, here we only consider memory access instructions within the heap and global data regions. This parameter tends to provide a more accurate estimation of the MAR, when the cost of local memory access is considered to be low compared to the expensive external memory access.
- *Memory Operand Ratio (MOR)*
  This is the percentage ratio of the total number of memory access operands to the total number of operands.
- *NLOC MOR*
  This is similar to the previous parameter. However, here we only consider the operands of the memory access instructions referencing the heap and the global data regions.
- *Stk Ratio*
  This is the percentage ratio of the total number of memory access instructions referencing the stack region to the total number of memory access instructions.
- *Flow Ratio*
  This is the total number of bytes read minus the total number of bytes written divided by the total number of bytes accessed. An extreme value of -1 means a write-only function, +1 represents the counterpart read-only function, and 0 represents a balanced R/W inert function.
- *NLOC-Flow Ratio*
  This is similar to the previous parameter. However, here we only consider memory accesses referencing the heap and the global data regions.
- *Byte-wise-Stk ratio*
  This is the percentage ratio of the total number of bytes accessed by the memory access instructions referencing the stack region to the total number of bytes accessed.
- *Bytes/Acc. ratio*
  This indicates the average value of the number of bytes accessed within each memory access instruction.

In the interpretation of these parameters, it should be generally noted that in some architectures, a single memory operand can be both read and written, for instance incl (%eax) on IA-32. In this case, we instrument it once for read and once for write. The same holds for combined R/W instructions.

VII. Case Study

In this section, we present a detailed analysis of an image processing application, i.e. Canny Edge Detection [41], to demonstrate the potentials of the advanced profiling techniques developed within the DWB platform. It serves as the direct approach to validate the usefulness, efficiency and applicability of the tools presented in this research work. *The main objective of the case study is to have an early yet comprehensive and thorough understanding of the application behavior, in particular of its memory access behavior and requirements.* Although the focus of the analysis is on the run-time attributes, we also examine the application source code to extract some valuable information. The result of this detailed profiling is primarily utilized to spot bottlenecks and deficiencies, particularly related to the application memory usage. It will provide hints for application developers to revise/optimize the application source code in order to gain better performance when running the application on a particular heterogeneous reconfigurable architecture. Throughout the experimental analysis, we used the same strategy and conducted several phases of source code optimizations as a means of verification of the profiling data. The extracted information can be further utilized in critical decisions during the design space exploration in heterogeneous multiprocessor systems, such as, HW/SW task partitioning, mapping and scheduling.

In this case study, we specifically aim to demonstrate the following qualities of the Q^2 framework:

- the visualization of the data communication between different kernels in the application,
- the prediction of hardware resource consumption for these kernels,
- the value of the generated data in identifying candidates for migration to reconfigurable components.

In the following, we will first discuss the Canny Edge Detection method in Section VII-A. Then, in Section VII-B, we will present our experimental setup. Finally, the results of the different steps in our case study are discussed in Section VII-C.

A. Canny Edge Detection

Canny [41] is a well-known edge detection algorithm. The method aims to achieve three main goals:

- *good detection* - this translates to the detection of as many of the real edges as possible, while also not falsely detecting non-existing edges as much as possible.
- *good localization* - this denotes the fact that the detected edges are as close as possible to the actual edges, i.e. the distance between the edge pixels as found by the detector and the actual edge is to be minimal.
- *unique detection of edges* - this means that real edges should be detected only once. This aspect of the detection method was added because the previous criteria do not imply that edges are only identified once.

Based on these criteria, the Canny edge detector first smoothes the image to eliminate any noise. It then finds the image gradient to highlight regions with high spatial derivatives. The algorithm then tracks along these regions and suppresses any pixel that is not at the maximum (non-maximal suppression). The gradient array is then further reduced by hysteresis, which tracks along the remaining pixels that have not been suppressed so far. Hysteresis uses two thresholds to accomplish this task. If the magnitude is below the first threshold, it is set to zero (made a non-edge). If the magnitude is above the high threshold, it is made an edge. In case the magnitude is between the two thresholds, then it is set to zero, unless there is a path from this pixel to a pixel with a gradient above the second threshold.

For our experiments, we have used the implementation provided by the Computer Vision Laboratory at the University of South Florida [42]. The application has the following steps:

- **Step 1**. *Filtering out any noise in the original image.* The Gaussian filter is used exclusively due to its simplicity. Once a suitable mask has been calculated, the Gaussian smoothing can be performed using standard convolution methods. A convolution mask is usually much smaller than the actual image. As a result, the mask is slid over the image, manipulating a square of pixels at a time. lower is the detector's sensitivity to noise. The localization error in the detected edges also increases slightly as the Gaussian width is increased. The width of the Gaussian mask used in the implementation is determined based on the standard deviation of the Gaussian smoothing filter that should be input by the user.
- **Step 2**. *Finding the edge strength.* This is done by taking the gradient of the image. The Sobel operator performs a 2D spatial gradient measurement on an image. Then, the approximate absolute gradient magnitude (edge strength) at each point is found. The Sobel operator uses a pair of 3×3 convolution masks, one estimating the gradient in the x-direction and the other estimating the gradient in the y-direction.
- **Step 3**. *Applying non-maximal suppression.* After finding the edge directions using the gradient values, non-maximal suppression is used to trace along the edges and suppress any pixel value that is not considered to be an edge. This will result in a thin line in the output image.
- **Step 4**. *Performing hysteresis.* Hysteresis is used to eliminate the breaking up of an edge contour caused by the edge pixels fluctuating above and below a threshold. Thresholding with hysteresis requires a low and a high threshold. Making the assumption that important edges should be along continuous curves in the image allows to follow a faint section of a given line and to discard a few noisy pixels that do not constitute a line but have produced large gradients. The low and high threshold values for hysteresis should also be specified as input parameters by a user.

B. Experimental Setup

All the experiments were performed on two different platforms. The general profiling of the CED application with *gprof* was done on an Intel 32-bit Core2 Duo E8500 @3.16 GHz with 4GB of RAM, running Linux kernel v2.6.34.7-0.7-pae. The application source code was compiled with *gcc* v4.5.0 and without any compiler optimization. We also utilized *gprof* on the embedded PowerPC 440 @400 MHz with 512 MB DRAM, which is integrated in a Xilinx ML510, Virtex 5 FX 130T with 1.3 MB BRAM FPGA board. In order to profile the application using the *QUAD* toolset, Pin DBI framework is needed which does not support PowerPC architecture. As a result, the *QUAD* profiling information on Intel x86 can demonstrate some level of inaccuracy for the architecture-specific data when targeting a different architecture. However, the overall image of the application should stay similar. The *Quipu* predictions target the Virtex 5 platform. For other FPGA devices, conversion formulas should be used based on the authentic published data-sheets.

C. Experimental Analysis

We have investigated the Canny Edge Detector (CED) application in several phases and report here the results of the different analysis steps that were taken. First, we present some conventional profiling analysis. Afterwards, we present the results of using the Q^2 profiling framework. Finally, we make some conjectures on how to adjust the program based on these data.

TABLE II
gprof flat profile for the *CED* application on the Intel x86 architecture.

Kernel	%time	self seconds	calls	self ms/call	total ms/call
gaussian_smooth	70.11	0.0985	1	98.50	98.50
non_max_supp	12.46	0.0175	1	17.50	17.50
magnitude_x_y	6.41	0.0090	1	9.00	9.00
apply_hysteresis	4.63	0.0065	1	6.50	11.50
follow_edges	3.56	0.0050	1433	0.00	0.00
derrivative_x_y	2.85	0.0040	1	4.00	4.00
canny	0.00	0.0000	1	0.00	140.50
make_gaussian_kernel	0.00	0.0000	1	0.00	0.00
read_pgm_image	0.00	0.0000	1	0.00	0.00
write_pgm_image	0.00	0.0000	1	0.00	0.00

% time is the percentage of the total execution time of the program used by the function; ***self seconds*** is the number of seconds accounted for by the function alone; ***calls*** is the number of times a function is invoked; ***self ms/call*** is the average number of milliseconds spent in the function per call; ***total ms/call*** is the average number of milliseconds spent in the function and its descendants per call.

TABLE III
gprof flat profile for the *CED* application on the embedded PPC.

Kernel	%time	self seconds	calls	self s/call	total s/call
gaussian_smooth	69.09	5.79	1	5.79	5.79
non_max_supp	19.81	1.66	1	1.66	1.66
magnitude_x_y	5.61	0.47	1	0.47	0.47
derrivative_x_y	3.70	0.31	1	0.31	0.31
apply_hysteresis	0.95	0.08	1	0.08	0.14
follow_edges	0.72	0.06	1433	0.00	0.00
canny	0.00	0.00	1	0.00	8.37
make_gaussian_kernel	0.00	0.00	1	0.00	0.00
read_pgm_image	0.00	0.00	1	0.00	0.00
write_pgm_image	0.00	0.00	1	0.00	0.00

Conventional profiling. We utilized *gprof* in order to have an approximate general understanding of the original CED application. The CED implementation consists of three source files with, in total, 15 functions. As mentioned earlier, three input parameters are required for the application to run. For all the experiments, we used a sample gray-scale image with a resolution of 800×600 and 8 bits per pixel in the *PGM* format. The input parameter for the standard deviation of the Gaussian blur kernel is set to 2.0. The values of the low and the high thresholds for performing hysteresis are both set to 0.5.

We used *gprof* for a typical run of the CED application without any compiler optimizations. The reported numbers demonstrate a considerable amount of error and do not reveal much valuable information. This is due to the fact that the application runs for a quite short time (approximately 150 milliseconds). Considering the sampling period which is 10 milliseconds, functions do not get much chances of being examined by the profiler. As mentioned previously, this is the main problem with *gprof*. To alleviate the problem, we run the application 20 times and recorded the average values. Table II summarizes these results. As seen in Table II, all the functions are called once with the exception of *follow_edges*, which is a recursive function. The number of times *follow_edges* is called depends on the image itself and on the values of input parameters. The primary share of the total execution time is attributed to *gaussian_smooth*. It is also interesting to note that there is no self contribution for *canny*, while the total contribution of this function and its descendants is around 140 milliseconds, which is equal to the sum of contributions from *gaussian_smooth*, *non_max_supp*, *magnitude_x_y*, *apply_hysteresis*, and *derrivative_x_y*. It indirectly implies that *canny* is the main function doing all the processing by just calling individual functions to carry out different phases in the edge detection algorithm described before. The time taken for reading the input image file and writing the output data is negligible.

Table III presents the *gprof* profiling results on the embedded PowerPC. As seen in the table, the total execution time of the application is increased by approximately 60x due to the decrease in the processing speed and simulated floating point arithmetic. Hence, the length of the application execution is large enough to get somehow accurate data with the sampling technique. Some changes are evident in the contribution percentages and the ordering of the functions. *follow_edges* and *apply_hysteresis* each now contributes less than 1 percent of the whole execution time.

A brief inspection of the application reveals the links between the main conceptual steps described in the CED algorithm and the corresponding functions defined in the source code. The top kernel, *gaussian_smooth*, utilizes the Gaussian filter to blur the input image. The filter itself is created in *make_gaussian_kernel*, which only allocates and fills a one-dimensional floating array. The size of the array is dependent on the input parameter *sigma* (standard deviation of the filter). However, for a predefined sigma value, there is a possibility of hard-wiring the individual calculated values. It is clear that the first step of the algorithm is the most time consuming task. The blurring procedure is performed on each pixel in the input image. *derrivative_x_y* computes the first derivatives (gradient) of the image along both the x and y directions. Subsequently, *magnitude_x_y* calculates the magnitude of the gradient. These functions together relate to the second step of the CED algorithm. Step 3 of the algorithm is implemented by *non_max_supp* which applies non-maximal suppression to the magnitude of the gradient image. Finally, *apply_hysteresis* implements the last step in the CED algorithm. Basically, the function finds edges that are above a high threshold or connected to a high pixel by a path of pixels greater than the low threshold. This is done by first initializing the edge map with all the possible edges that the non-maximal suppression suggested, except for the borders. Then, when a pixel is located above the high threshold, the function calls the recursive function *follow_edges* to continue tracking the edge along all paths.

In order to have accurate contribution estimates and also an overview of some memory access related statistics, we used *MAIP* to profile the application. Table IV presents a summary of the results. The most accurate values for the execution contribution of each function is calculated with *MAIP*, as it accounts for each single instruction within a function, contrary

TABLE IV
MAIP Flat profile for the *CED* application.

Kernel	% time	MAR	NLOC-MAR	MOR	NLOC-MOR	Stk Ratio	Flow Ratio	NLOC-Flow Ratio	Bytes/Acc.
gaussian_smooth	69.89	29.05	11.75	9.56	3.74	61.54	0.7376	0.9213	3.3090
non_max_supp	14.22	51.89	10.36	20.58	3.84	89.79	0.3635	0.8712	3.4318
magnitude_x_y	5.62	50.00	9.37	17.98	3.37	87.51	0.4168	0.3333	3.0007
apply_hysteresis	4.97	23.21	21.71	8.26	7.54	27.10	0.3288	0.4026	1.2998
derrivative_x_y	2.99	64.74	35.22	26.70	13.32	66.79	0.5557	0.3334	3.0028
follow_edges	2.24	48.33	7.79	18.25	4.41	94.85	-0.0030	0.8225	3.4357
read_pgm_image	0.03	46.81	22.56	18.78	9.39	59.02	0.5881	0.7547	3.4548
write_pgm_image	0.00	60.61	28.72	24.32	15.41	43.57	0.1258	0.1652	3.8105
make_gaussian_kernel	0.00	45.14	10.72	15.34	5.20	80.06	0.2529	0.8504	4.7549
canny	0.00	52.26	12.22	20.23	7.34	75.84	0.1889	0.6508	3.9658

MAR is the percentage ratio of the memory access operations to the total instructions executed in the application; ***NLOC-MAR*** is the same as ***MAR*** except that only references to the non-local region are considered; ***MOR*** is the percentage ratio of the memory access operands to the total number of operands; ***NLOC-MOR*** is the same as ***MOR*** except that only references to the non-local region are considered; ***Stk Ratio*** is the percentage ratio of the memory access instructions within the local region to the total memory access instructions; ***Flow Ratio*** is an indication of a function being more memory reader or writer. -1 means that the function only writes to the memory and +1 means that the function only read from memory; ***NLOC-Flow Ratio*** is the same as ***Flow Ratio*** except that only references to the non-local region are considered;

to the sampling technique used in *gprof*. As seen in Table IV, the values more or less conform to the numbers that were calculated previously. *gaussian_smooth* is not only the top contributor for the CED application, but also it demonstrates a relatively low percentage of memory access related workload compared to the computational burden. The *NLOC-MAR* and *Stk ratio* columns in Table IV can reveal valuable hints regarding the tendency of the function to go for local or non-local memory accesses. This is an important attribute if we opt to maintain the dynamically allocated memory on external sources (off-chip memory) in reconfigurable devices. For example, we would rather map *gaussian_smooth* or *non_max_supp* on FPGA than *apply_hysteresis* as a higher portion of memory accesses are intended for external memory regarding the latter function. In the case of the Molen platform, however, this is not relevant, as Molen does not currently support off-chip memory. As such, all the memory space required for the execution of the application should be allocated and managed on-chip. In this respect, a memory management module particularly takes care of all the dynamic memory allocations in the application.

Quipu Profiling. In the continuing work with the CED application, we want to be able to map different kernels to hardware. An important caveat for mapping kernels to hardware is that within the DWB some restrictions apply to the kernels that will be mapped to hardware. For this reason, we have modified the CED application where necessary, so that the intensive kernels could be mapped to hardware. There were two main issues that were solved:

- **Memory allocation**
  The DWARV compiler currently does not support allocating memory blocks at runtime from hardware. Therefore, all memory allocations were moved from the kernels into function stubs that allocate the required memory blocks first and then call the real kernels.
- **(Recursive) function calls**
  Furthermore, at this time function calls are not supported in DWARV. For most cases, manually inlining the code solved the problem. However, there was one instance of recursion in the *follow_edges* function, which inhibited simple inlining in the *apply_hysteresis* function. Therefore, the recursive function call was moved to an appropriate function stub.

Kernel	Area[a]			MAIP %time	Speed-up[b]	
	Slices	% of area	cum.[c] % of area		single kernel	cum.
hw_gaussian_smooth	2265	11.1%	11.1%	70.59%	3.40×	3.40×
hw_derrivative_x_y	2720	13.3%	24.3%	2.49%	1.03×	3.71×
hw_magnitude_x_y	1143	5.6%	29.9%	5.14%	1.05×	4.59×
non_max_supp	5599	27.3%	57.3%	14.36%	1.17×	13.48×
hw_apply_hysteresis	36617	178.8%	236.1%	2.68%	1.03×	21.10×

[a]Area predicted by a *Quipu* prediction model for the Virtex 5 FX 130T.
[b]Theoretical application speed-up, assuming 0s execution time for each kernel.
[c]cumulative

TABLE V
Area predictions and theoretical speed-ups for the kernels in the CED application.

Of course, these changes required new profiling results. In Table V, the top 5 kernels are listed with their associated new time contributions, as reported by MAIP.

Evidently, in order to partition the application over hardware and software components, the evaluation of the computational and memory hot-spots is not sufficient. As there is only a limited amount of reconfigurable hardware area available, an investigation of the size of potential hardware designs is warranted. For this purpose, we used a *Quipu* prediction model for the Virtex 5 FX 130T FPGA. The results of the area prediction of the top 5 contributing kernels are presented in Table V. The table lists the actual number of slices, as well as the percentual area with respect to the target FPGA. The kernels in the table are in the order of execution in the CED application. As such, we can evaluate subsequent merging options by providing cumulative area figures as well. Note, that the *apply_hysteresis* kernel is exceedingly resource-intensive, taking up 178.8% of the target FPGA area. This large area requirement can be traced back to a local array of 32K 32-bit integers. The used *Quipu* model targets the DWARV C2VHDL compiler, which converts such local arrays to registers, resulting in a large number of slices. When considering to merge several kernels together, the predictions suggest the first four kernels will easily fit together on the target FPGA.

In order to find a candidate set of functions to be moved to the FPGA hardware, we need an idea of the speed-up that might be achieved. Therefore, in addition to the area predictions, the theoretical application speed-ups for each kernel are also reported in Table V. These speed-ups are calculated using Amdahl's law assuming an unlimited speed-up for the kernel(s) in question, as follows:

$$\lim_{p\to\infty} \frac{p}{1-f(p-1)} = \frac{1}{f} = \frac{1}{1-s} \tag{4}$$

where p stands for the speed-up factor that is achieved in the accelerated part of the application, f stands for the percentual contribution of the remaining sequential part of the application, and s stands for the percentual contribution of the accelerated part of the application before acceleration. Table V lists both the speed-up when one kernel is accelerated and the cumulative speed-up, where each subsequent kernel is accelerated together with the previously accelerated kernels. Observe that as large parts of the application are accelerated, the contribution of the remaining kernels becomes more significant. For example, *apply_hysteresis* has a contribution of 2.68%, but with much of the application already accelerated, the difference in theoretical speed-up is 56.8%.

As we have mentioned before, merging the first four kernels in Table V would yield a hardware block that would fit in the target FPGA. The maximum speed-up of the application using that block would be 13.48×. Of course, the efficiency of accelerating this block will never be 100%. And the actual speed-up will be lower. The designer would have to decide whether to go for the larger kernel or to instantiate several blocks of smaller size in parallel. This last option can especially be interesting when edge detection is performed on a video stream.

QUAD profiling. In order to have a clear insight into the CED application behavior regarding the data communication between different functions, we utilized the *QUAD* core to profile the application. The Quantitative Data Usage (QDU) graph of the CED is depicted in Figure 4. The graph makes it possible to visually follow the journey of data objects through the sequences of function calls. In this way, we can trace what is actually happening to the input image as it moves along different phases of the CED algorithm. The detailed information presented in the graph also helps to understand what are the memory requirements of each function to accomplish its task. Furthermore, it plainly identifies the actual data dependencies between functions. The critical data path is highlighted in the graph to distinguish it from all the subordinate data communications between functions. *QUAD* starts instrumenting the application as soon as the main function gains control. In other words, the data communication associated with the operating system calls and libraries are somehow overlooked. Although this normally does not contain much valuable information about the critical data path of the application, it may cause missing starting or ending points for communication edges, i.e. the producer or consumer of the data. Currently, for every byte with an unidentified producer a manual investigation has to be conducted to verify the source of data. In the CED application, the main part of the input image file (800×600 bytes) has been tracked down to be part of the *unknown producer* entity. We revised the graph to replace this with a dummy *Image* node. A small part of the input image (4864 bytes) is recognized to be attributed to *read_pgm_image*. Since the file reading process is considered to be I/O, the operating system will take care of it, and how the process is implemented is completely platform dependent. Hence, slight inconsistencies appear in numbers reported by *QUAD* for such cases. Nevertheless, this is not affecting the overall picture of the data communication in the CED application.

make_gaussian_kernel is responsible for creating the Gaussian filter array. Based on the standard deviation used in our case study, an array of eleven elements is allocated. This is verified by the 44 UnMA reported on the edge from *make_gaussian_kernel* to the *hw_gaussian_smooth*. It should be noted that the array of eleven floating point values is accessed repeatedly throughout the smoothing process. As a result, huge data communication is reported between the two functions (about 40MB). *hw_gaussian_smooth* produces a temporary image data object filled with calculated intermediate floating point values. This is clearly reported by a self edge of 800×600×4 bytes UnMA in the graph. *hw_derrivative_x_y* uses the smoothed image data object as an input. The smoothed image used in *hw_derrivative_x_y* is of the short integer type, which is appearing as a corresponding edge of 960000 UnMA between the *hw_gaussian_smooth* and *hw_derrivative_x_y*. It can also be derived that the smoothed image data object is accessed four times in total (3840000 bytes), two times for calculating the derivative in the X direction and two times for the Y direction.

The calculated derivatives are stored separately in two arrays of the short integer type. They serve as inputs to *hw_magnitude_x_y*, which will compute edge strength values based on the gradient of the image. The result will be saved in an array called *magnitude*. It is also clear from the graph that the input arrays are scanned only once to compute the magnitudes. The third step of the CED algorithm needs the computed *magnitude* array and also the derivative arrays. This can be verified by the graph edges connecting *hw_derrivative_x_y* and *hw_magnitude_x_y* to the *non_max_supp*. For each pixel, the values of neighboring pixels in some directions are also examined. This results in a large number reported for memory accesses (approximately 4.5 MB). The resulting binary image, also known as *thin edges*, is output into an array with the same size of the input image. Regarding the final step, *hw_apply_hysteresis* (excluding the last recursive part to trace along the identified edges) uses the *magnitude* array and the output array from non-maximal suppression. As shown in the extracted data, the usage of *magnitude* is conditioned to only those pixels which indicate a possible edge. This is dependent on

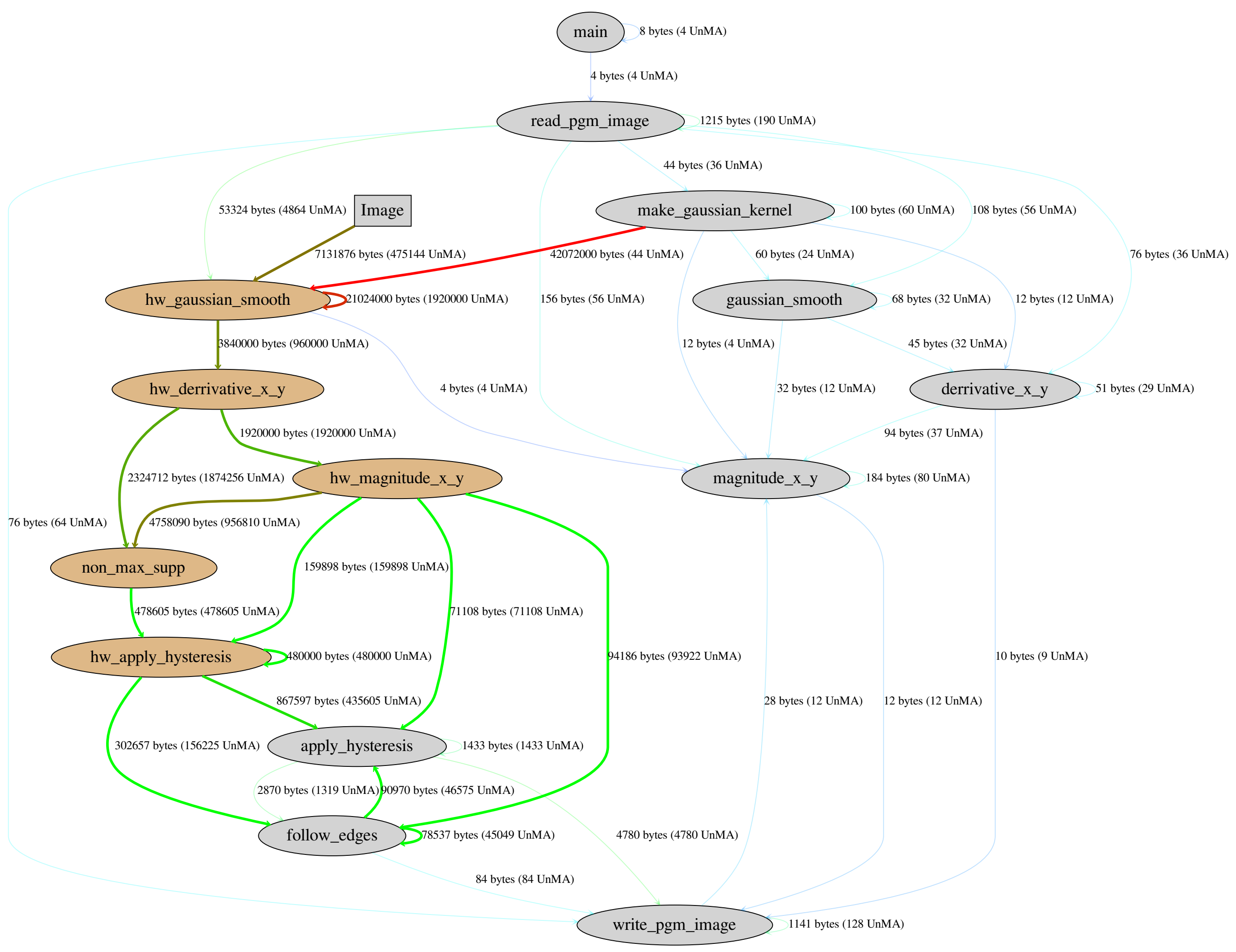


Fig. 4. Quantitative Data Usage graph for the hardware version of the Canny application.

the input image. In our case, about one sixth of the total pixels are identified as possible edge pixels. The self edge of 480000 bytes in *hw_apply_hysteresis* is attributed to the initialization of the *edge map* array. The array is subsequently processed in *hw_apply_hysteresis* for the computation of the histogram of magnitude values. *apply_hysteresis* finalizes the *edge map* by using the output of *hw_apply_hysteresis* and the *magnitude* array. To accomplish this task, it in turn utilizes *follow_edges*. All the corresponding data communications to perform the hysteresis process is identified and presented in Figure 4. *write_pgm_image* is responsible for writing the *edge map* array to the output file. As explained before, no major consumer of the array is identified by *QUAD* due to the I/O process.

VIII. Conclusions

The primary obstacle for improving the overall performance of computing systems arises from the communication bottleneck between processing elements and memory subsystem. This bottleneck is even more evident with the introduction of heterogeneous architectures containing reconfigurable fabrics; where, in addition to the memory bottlenecks, there are resource constraints that have to be taken into account. We have demonstrated in this paper that advanced profiling tools such as present in our Q^2 profiling framework can alleviate this problem. We plan to investigate the utilization of this framework more in-depth in the near future.

Acknowledgment

This research is partially supported by Artemisia iFEST project (grant 100203), Artemisia SMECY (grant 100230), and FP7 Reflect (grant 248976).

References

[1] K. Bertels and et al., “Developing applications for polymorphic processors: The delft workbench,” Delft University of Technology, Tech. Rep., January 2006.
[2] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof: A call graph execution profiler,” *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, 1982.
[3] D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, and G. Stitt, “Profiling tools for hardware/software partitioning of embedded applications,” in *LCTES*, 2003, pp. 189–198.
[4] “Intel’s vTune,” http://software.intel.com/en-us/intel-vtune.
[5] “AMD CodeAnalyst,” http://developer.amd.com/cpu/codeanalyst.

[6] R. F. Cmelik, "Spixtools: Introduction and user's manual," Sun Microsystems Inc., Mountain View, CA, USA, Tech. Rep., 1993.

[7] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Trans. Program. Lang. Syst.*, vol. 16, pp. 1319–1360, July 1994.

[8] M. Martonosi, A. Gupta, and T. Anderson, "Memspy: analyzing memory system bottlenecks in programs," *SIGMETRICS Perform. Eval. Rev.*, vol. 20, no. 1, pp. 1–12, 1992.

[9] H. Davis and S. R. Goldschmidt, "Tango: A multiprocessor simulation and tracing system," Stanford University, Stanford, CA, USA, Tech. Rep., 1990.

[10] A. R. Lebeck and D. A. Wood, "Cache profiling and the spec benchmarks: A case study," *IEEE Computer*, vol. 27, pp. 15–26, 1994.

[11] A. I. Choudhury, K. C. Potter, and S. G. Parker, "Interactive visualization for memory reference traces," *Computer Graphics Forum*, vol. 27, no. 3, pp. 815–822, May 2008.

[12] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "Profileme: hardware support for instruction-level profiling on out-of-order processors," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 30, 1997, pp. 292–302.

[13] "Cacheprof," http://www.cacheprof.org.

[14] P. Schumacher and P. Jha, "Fast and accurate resource estimation of rtl-based designs targeting fpgas," in *FPL '08: Proceedings of 18th International Conference on*, sep. 2008, pp. 59 –64.

[15] C. Brandolese, W. Fornaciari, and F. Salice, "An area estimation methodology for fpga based designs at systemc-level," in *DAC '04: Proceedings of the 41st annual*, New York, NY, USA, 2004, pp. 129–132. [Online]. Available: http://doi.acm.org/10.1145/996566.996606

[16] P. Kannan and D. Bhatia, "Interconnect estimation for FPGAs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 8, pp. 1523–1534, Aug. 2006.

[17] L. M. Chuong, S.-K. Lam, and T. Srikanthan, "Area-Time Estimation of Controller for Porting C-Based Functions onto FPGA," in *RSP '09: Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, 2009, pp. 145–151.

[18] D. Kulkarni, W. A. Najjar, R. Rinker, and F. J. Kurdahi, "Compile-time area estimation for LUT-based FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 1, pp. 104–122, 2006.

[19] T. Degryse, H. Devos, and D. Stroobandt, "FPGA Resource Estimation for Loop Controllers," in *ODES'08: Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems*, Boston, MA, USA, 2008, pp. 9–15.

[20] L. Deng, K. Sobti, and C. Chakrabarti, "Accurate models for estimating area and power of fpga implementations," in *ICASSP 2008: IEEE International Conference on*, 31 2008-april 4 2008, pp. 1417 –1420.

[21] S. Vassiliadis and et al., "The molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[22] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Moscu Panainte, "The molen programming paradigm," in *Computer Systems: Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science, A. Pimentel and S. Vassiliadis, Eds., vol. 3133. Springer Berlin / Heidelberg, 2004, pp. 1–10.

[23] S. A. Ostadzadeh, R. Meeuws, C. Galuzzi, and K. Bertels, "QUAD - a memory access pattern analyser," in *ARC 2010*, 2010, pp. 269–281.

[24] S. A. Ostadzadeh, M. Corina, C. Galuzzi, and K. Bertels, "tQUAD - memory bandwidth usage analysis," in *ICPP 2010*, September 2010, pp. 217–226.

[25] R. J. Meeuws, K. Sigdel, Y. D. Yankova, and K. Bertels, "High level quantitative interconnect estimation for early design space exploration," in *ICFPT '08*, December 2008, pp. 317–320.

[26] S. A. Ostadzadeh, R. J. Meeuws, K. Sigdel, and K. Bertels, "A clustering framework for task partitioning based on function-level data usage analysis," in *FPGA '09*, 2009, pp. 279–279.

[27] C. Galuzzi, "Automatically fused instructions - algorithms for the customization of the instruction-set of a reconfigurable architecture," Ph.D. dissertation, TU Delft, May 2009.

[28] S. A. Ostadzadeh, R. J. Meeuws, K. Sigdel, and K. Bertels, "A multipurpose clustering algorithm for task partitioning in multicore reconfigurable systems," in *CISIS '09*, 2009, pp. 663–668.

[29] E. M. Panainte, K. Bertels, and S. Vassiliadis, "The molen compiler for reconfigurable processors," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 1, 2007.

[30] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable VHDL generator," in *Proc. of FPL07*, 2007, pp. 697–701.

[31] S. A. Ostadzadeh, M. Corina, C. Galuzzi, and K. Bertels, "Runtime extraction of memory access information from the application source code," in *to appear in the proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, July 2011.

[32] R. J. Meeuws, "A quantitative model for hardware/software partitioning," Master's thesis, Delft University of Technology, Delft, Netherlands, Delft, Netherlands, May 2007.

[33] R. J. Meeuws, Y. D. Yankova, K. Bertels, G. N. Gaydadjiev, and S. Vassiliadis, "A quantitative prediction model for hardware/software partitioning," in *FPL '07: Proceedings of 17th International Conference on*, August 2007, pp. 317–320.

[34] R. Meeuws, K. Sigdel, Y. Yankova, and K. Bertels, "High level quantitative interconnect estimation for early design space exploration," in *FPT'08. International Conference on Field Programmable Technology*, dec 2008, pp. 317 –320.

[35] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable vhdl generator," in *FPL '07: Proceedings of 17th International Conference on*, 27-29 2007, pp. 697 –701.

[36] Y. Ben-Asher and N. Rotem, "Synthesis for variable pipelined function units," in *System-on-Chip, 2008. SOC 2008. International Symposium on*, nov. 2008, pp. 1 –4.

[37] ——, "Automatic memory partitioning: increasing memory parallelism via data structure partitioning," in *CODES: IEEE/ACM/IFIP international conference on*, ser. CODES/ISSS '10, New York, NY, USA, 2010, pp. 155–162. [Online]. Available: http://doi.acm.org/10.1145/1878961.1878989

[38] A. A. C. Experts. Cosy: Compiler system. [Online]. Available: http://www.ace.nl/

[39] C. Luk and et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05*. New York, NY, USA: ACM, 2005, pp. 190–200.

[40] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, pp. 490–499, September 1960.

[41] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, pp. 679–698, November 1986.

[42] "Canny Edge Detector, Image Analysis Research Lab., USF," http://marathon.csee.usf.edu/edge/edge_detection.html.

How Parameterizable Run-time FPGA Reconfiguration can Benefit Adaptive Embedded Systems

Dirk Stroobandt and Karel Bruneel
Ghent University, ELIS Department, Gent, Belgium, Dirk.Stroobandt@UGent.be

Abstract—*Adaptive embedded systems are currently investigated as an answer to more stringent requirements on low power, in combination with significant performance. It is clear that runtime adaptation can offer benefits to embedded systems over static implementations as the architecture itself can be tuned to the problem at hand. Such architecture specialisation should be done fast enough so that the overhead of adapting the system does not overshadow the benefits obtained by the adaptivity. In this paper, we propose a methodology for FPGA design that allows such a fast reconfiguration for dynamic datafolding applications. Dynamic Data Folding (DDF) is a technique to dynamically specialize an FPGA configuration according to the values of a set of parameters. The general idea of DDF is that each time the parameter values change, the device is reconfigured with a configuration that is specialized for the new parameter values. Since specialized configurations are smaller and faster than their generic counterpart, the hope is that their corresponding system implementation will be more cost efficient. In this paper, we show that DDF can be implemented on current commercial FPGAs by using the parameterizable run-time reconfiguration methodology. This methodology comprises a tool flow that automatically transforms DDF applications to a runtime adaptive implementation. Experimental results with this tool flow show that we can reap the benefits (smaller area and faster clocks) without too much reconfiguration overhead.*

Keywords: Automatic hardware synthesis, Dynamic Data Folding, FPGA, Run-time reconfiguration

1. Introduction

In order to keep up with more stringent requirements on power usage along with performance, current embedded systems are increasingly made adaptive. In a first step towards full adaptivity, task scheduling on embedded systems has been changed from static (off-line) to dynamic (on-line) scheduling so as to cope with dynamism in the applications. Generally, application scenarios are detected at run-time and for each scenario, the proper schedule is chosen from the set of statically derived schedules for all application scenarios on the architecture at hand [1], [2], [3]. In more advanced embedded systems, also the architecture itself is made adaptive [4], [5]. In this way, not only the schedule can change but also the resource allocation can be altered depending on the application scenario at hand. It is clear that such runtime architecture adaptation can offer benefits to embedded systems over static implementations as the architecture itself can be tuned to the problem at hand. Such architecture specialisation should be done fast enough so that the overhead of adapting the system does not overshadow the benefits obtained by the adaptivity.

One hardware component that is extremely well fit for combining performance with adaptivity is the FPGA. The inherent reconfigurability of SRAM-based FPGAs makes it possible to dynamically optimize the configuration of the FPGA for the situation at hand. Since optimized configurations are smaller and faster than their generic counterparts, this may result in a more efficient use of FPGA resources [5]. Therefore, dynamically reconfiguring FPGAs is a good way of introducing the architecture adaptivity in the context described above.

If the number of possible application scenarios is limited, a dynamically reconfiguring system can easily be implemented with a conventional FPGA tool flow. One simply generates an FPGA configuration optimized for each possible situation and stores these in a configuration database. At run-time, a configuration manager loads the appropriate configuration from the database in the FPGA depending on the situation at hand.

However, in most cases the number of possible configurations is very large. This is especially the case for Dynamic Data Folding (DDF). DDF is a technique to implement applications where some of the input data, called the parameters, change only once in a while. Each time the parameters change value, the FPGA is reconfigured with a configuration that is specialized for the new parameter values. It's easy to see that the number of possible configurations grows exponentially with the number of parameter bits. This makes it impossible to store all possible configurations. On the other hand, a conventional FPGA tool flow is too slow to be executed at run-time. DDF can therefore not be implemented with a conventional tool flow.

Our research group at Ghent University is the first to present an *automatic* tool flow that builds DDF implementations, thus bringing the FPGA architecture to the level where it could be useful in a dynamic run-time adaptive embedded system [5]. Our methodology and tool flow starts from parameterized HDL designs. These are RT-level HDL designs in which a distinction is made between regular inputs

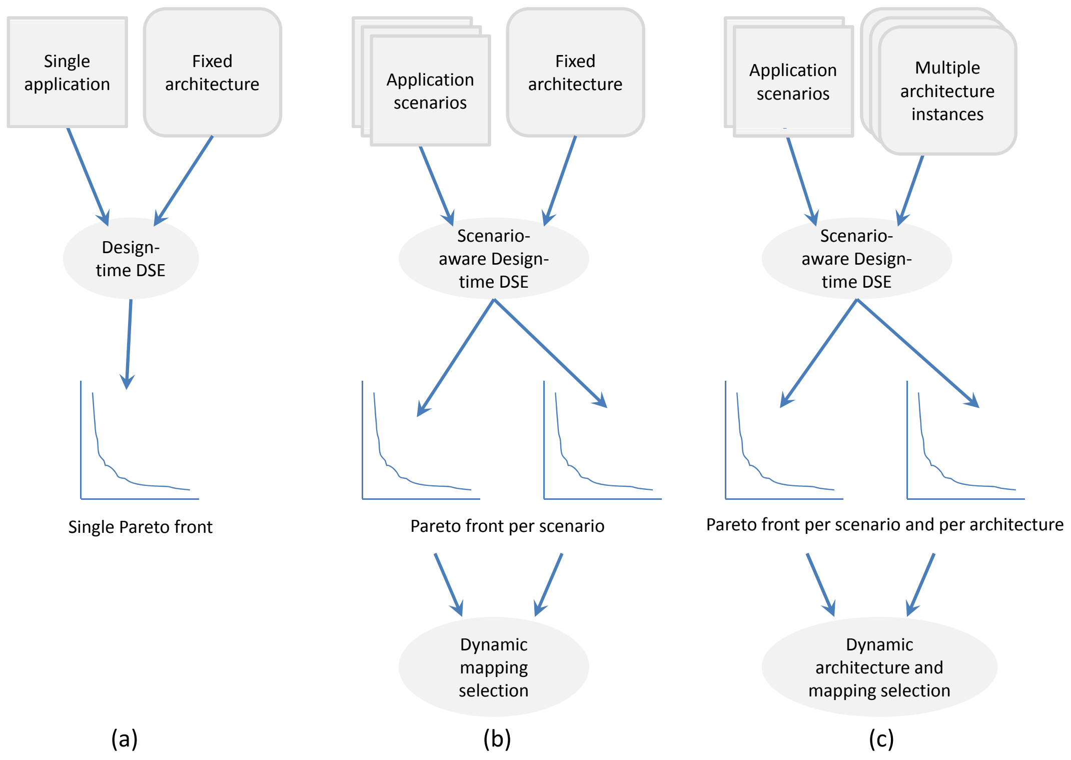

Single application
Fixed architecture
Design-time DSE
Single Pareto front
Application scenarios
Fixed architecture
Scenario-aware Design-time DSE
Pareto front per scenario
Dynamic mapping selection
Application scenarios
Multiple architecture instances
Scenario-aware Design-time DSE
Pareto front per scenario and per architecture
Dynamic architecture and mapping selection
(a)
(b)
(c)

Fig. 1: Different forms of system-level DSE, with increasing dynamism from (A) to (C).

and parameter inputs. The parameter inpus will define the reconfiguration intervals. The result of the tool flow is a parameterizable configuration. This is an FPGA configuration in which some of the configuration bits are expressed as a closed-form Boolean expression of the parameters. At run-time, the configuration manager does not fetch the specialized configuration from a large configuration database. Instead, it generates the required configuration on the fly by evaluating the closed-form Boolean expressions for the new parameter values.

This paper starts with a brief discussion on the context of adaptive embedded systems (Section 2), as well as a description of DDF and an overview of related work in Section 3. In Section 4, we give a high-level overview of our staged mapping tool flow. The tool flow uses the same steps as a conventional tool flow: synthesis, technology mapping, place and route. The main difference between our tool flow and the conventional tool flow lies in the technology-mapping step, which we generalize to obtain a new technology mapper called TMAP, suited for our DDF tool flow. We show that our new method for parameterizable run-time reconfiguration can be implemented on current FPGA devices and without too many changes in the FPGA implementation tool flow (Section 5). Furthermore, we apply TMAP on adaptive FIR filters and Ternary Content-Addressable Memory in Section 6. Experimental results show that the use of self-reconfiguration with our tool flow improves the resource demands of the application by 39% for a 32-tap adaptive filter and 66% for a Ternary Content Addressable Memory implementation, without introducing a prohibitively large reconfiguration generation overhead. Finally, we conclude in Section 7.

2. Adaptive Embedded Systems

Today, modern embedded computing systems are not only rapidly evolving towards MultiProcessor System-on-Chîps (MPSoC), they are also increasingly dynamic and adaptive. The application workload can change dramatically over time. For this reason, the notion of application scenarios has gained interest in the past years [1], [2], [3]. An application scenario describes the evolution of system use cases, i.e., the combinations of applications that can be active at the same time. This has important implications on the scheduling process that maps tasks to the computing nodes. A careful trade off has to be made of non-functional requirements such as performance, power, cost, etc. Design Space exploration (DSE) is therefore a very important aspect in this mapping process. DSE is a multi-objective optimization problem that searches through the space of different mapping alternatives in order to find Pareto-optimal design instances. A design instance is said to be Pareto-optimal when it is optimal for at least one of the optimization objectives (e.g., performance, power, cost, etc.). The traditional, design-time DSE is illustrated in Figure 1(A). With the occurrence of different application scenarios, this traditional DSE has to be extended. The mapping decisions can no longer be made at design time. A run-time system configuration manager will be needed to dynamically map and re-map applications onto the underlying architectural resources. The current state-of-the art in this field has only very recently started to address this situation (Figure 1(B)) [1], [6], [7], [8], [9], [10].

However, even more flexibility and optimization options are available when also the underlying architecture of embedded systems is adaptive. One way of achieving this is to include reconfigurable hardware components such as Field Programmable Gate Arrays (FPGAs), now popular architectural elements that enable to accelerate specific computational kernels in applications by means of run-time hardware reconfiguration (e.g. [4], [5]). Making not only the applications but also the hardware adaptive, poses additional challenges to DSE. Ideally, a system configuration manager in such a system continuously optimizes the system for non-functional requirements (performance, power consumption, etc.) at run-time, both by means of mapping of application tasks and by reconfiguraing architectural processor and network components (Figure 1(C)). This type of DSE is still only in the research planning phase. However, the optimization possibilities clearly outnumber the ones in previous DSE frameworks.

In such a new type of DSE optimizations, this increased flexibility poses an additional problem. Application scenarios can be considered a given (and can be measured or modelled). However, the use of FPGAs offers a seemingly infinite architecture implementation space and the best architecture has to be found for each application scenario instance. The main challenge in this kind of framework is hence the right choice of FPGA implementations to consider in DSE. In the next section, we will show that dynamic data folding applications offer an interesting perspective that offers the possibility to limit the implementation choices while retaining the flexibility needed to implement an almost optimal architecture for each application scenario.

3. Dynamic Data Folding

Dynamic data folding (DDF) applications have two types of inputs that are treated differently: fast changing inputs (regular inputs) and slow changing inputs (parameter inputs). Instead of building generic circuitry where both types of inputs are normal input signals, we build a dynamic data folding system where only the regular inputs are inputs to a reconfigurable module implemented in the FPGA fabric. The parameters are inputs to a second subsystem, the configuration manager (CM), in our case an instruction set processor (ISP). Every time the parameters change, the CM specializes the reconfigurable module for the new parameter values. Once specialized, the module is ready to process the fast changing input data. The reason to build a DDF system is that the reconfigurable module can be implemented more efficiently in the FPGA fabric than the generic circuitry. With convential FPGA tools only handcrafted DDF systems are possible [11], [12]. The TMAP tool flow on the other hand (see Section 4) automatically maps dynamic data folding applications to a self-reconfiguring system [13]. The input of the tool chain is a behavioral description of the functionality in which a distinction is made between regular inputs and the parameter inputs. The output is a Tunable LookUp Table (TLUT) circuit that consists of a fixed LUT-structure and a Boolean circuit we call the Partial Parameterizable Configuration (PPC). The PPC describes the Boolean dependency of the truth table bits on the parameters as a Boolean circuit that consists of AND and inverter gates. This is also called an AND-Inverter Graph (AIG) [14]. As an example we chose the selection bits of a 4-input multiplexer as parameters and mapped it to 3- LUTs.[1] The resulting fixed LUT structure and AIG of the PPC are shown in Figure 2. We note that making a generic 4-input multiplexer with 3-LUTs takes 6 LUTs, while this datafolded version only takes 2 LUTs. The fixed LUT circuit can be placed and routed on the FPGA fabric using conventional tools. The PPC is compiled to an evaluation function that has to be carried out by the CM. More specifically, the evaluation function consists of C-code that can run on an instruction set processor (ISP). From the locations of the LUTs on the FPGA and the evaluation function of the PPC the specialization procedure is synthesized. The specialization procedure takes the parameters as arguments, generates new truth tables for the reconfigurable module and writes them in the configuration memory. The specialization procedure thus consists of an evaluation of the PPC and a reconfiguration of the truth tables of the fixed LUT circuit.

4. Staged Mapping Tool Flow

In DDF, the specification of the specialized configuration becomes available in two stages. At compile time, the generic functionality is available, but the parameter inputs are not yet bound. Only at run time, the parameters get bound and the full specification is available. A conventional tool flow needs a full specification from the start. Therefore, the complete mapping process needs to be executed at run time in order to generate the specialized configuration. Generating the specialized configuration from scratch every time the parameters change results in a large specialization overhead. However, since a large part of the specification (the generic functionality) is available at compile time, one would expect that it should be possible to complete a large part of the mapping process at compile time, which can then be refined at run time when the parameter values become available. In this case, one would expect a large reduction

[1]K-LUTs are LUTs with K inputs.

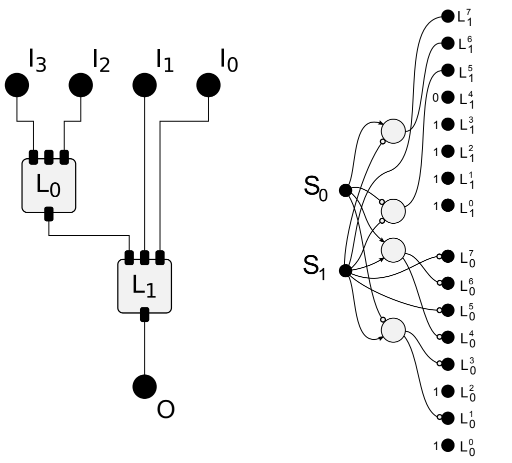


Fig. 2: The fixed LUT structure and AIG of the PPC for the 4-input multiplexer example.

in specialization overhead since only the refinement step needs to be executed at run time. Our tool flow uses this technique, which we call *Staged Mapping*. A similar concept has been used in software compilation, where it is called staged compilation. It has also been used in FPGA mapping, e.g. in [15] where part of the synthesis process was moved from run time to compile time.

Figure 3 gives an overview of our tool flow. The final result, a *Specialized Configuration* for the FPGA, is generated in two steps or stages: the *Generic Stage* and the *Specialization Stage*. The generic functionality is presented to the generic stage in the form of a *Parameterizable HDL Design*, while the parameter values are only known at the beginning of the specialization stage. The generic stage produces a *Parameterizable Configuration* (PC). The specialization stage combines this with the parameter values to produce the specialized configuration each time the parameters change.

A *Parameterizable HDL Description* is an HDL description in which we make a distinction between *regular input ports* and *parameter input ports*.[2] The parameter inputs will not be inputs of the final specialized configurations. They will be bound to a constant value during the specialization stage.

A PC is a function that takes parameter values as arguments and produces a specialized configuration. Since both parameters and FPGA configurations are bit vectors, the parameterizable configuration is a multi-output Boolean function. Since many of the output bits of the PC are independent of the parameter inputs, we can reduce the number of configuration bits that need to be reconfigured by splitting up the PC in a *Template Configuration* (TC) and a *Partial Parameterizable Configuration* (PPC). The TC contains all static bits and is used to configure the FPGA once when the system is started. Just like the PC, the PPC is a multi-output Boolean function. The PPC will be used by the reconfiguration procedure to generate a new partial configuration for the FPGA. In previous work [16], [5] we have represented the PPC as a vector of closed-form single-output Boolean expressions of the parameter inputs (called *Tuning functions*). In this paper, we represent the PPC as a Boolean network (Figure 2). This enables the use of combined logic optimization and thus leads to a more compact representation and faster evaluation.

In the parameterizable configurations generated by the tool flow presented in this work, only the truth tables of the LUTs are expressed as a function of the parameter inputs. All other configuration bits are static and will thus be part of the TC. In other work [17], we have built a tool flow where the routing bits can also be expressed as a function of the parameter inputs. This tool flow can in some cases further reduce the number of FPGA resources. However, in this paper we focus on the reconfiguration of LUTs.

The steps needed in the generic stage of our two-stage approach are similar to those used in conventional FPGA mapping: synthesis, technology mapping, place and route. It is important to note here that these algorithms are computationally hard and thus have a long run time. The specialization stage on the other hand generates a specialized FPGA configuration by evaluating the PPC, which is represented as a Boolean network. It can be shown [18] that the number of Boolean gates in this network scales linearly with the number of gates in the generic implementation. The specialization stage is thus not computationally hard and will run a lot faster than the generic stage. Therefore, the staged mapping tool flow is much more efficient in generating specialized configurations than a conventional tool flow. This is because our staged flow can reuse the parameterizable configuration for each parameter value. The effort spent in the generic stage thus is divided over all invocations of the specialization stage. For large sets of parameter values, the average mapping effort asymptotically reaches the effort spent in the specialization stage.

5. Practical Tool Flow Instance

In the previous section, we presented the general tool flow to map an application to a self-reconfiguring platform. However, to enable a commercial introduction of this tool flow without too many hurdles, we have searched for a practical tool flow that uses current commercial tools as much as possible and only needs a very limited amount of additional tools. The tool flow presented in this section targets Xilinx components and reuses many Xilinx tools.

The self-reconfiguring platform (Fig. 4) targeted by our tool flow is implemented on a Xilinx Virtex-II Pro FPGA. The configuration manager is implemented on an embedded

[2]One should be careful not to confuse a parameter input with a generic as defined in VHDL or a parameter as defined in Verilog. A parameter input is a special kind of input port.

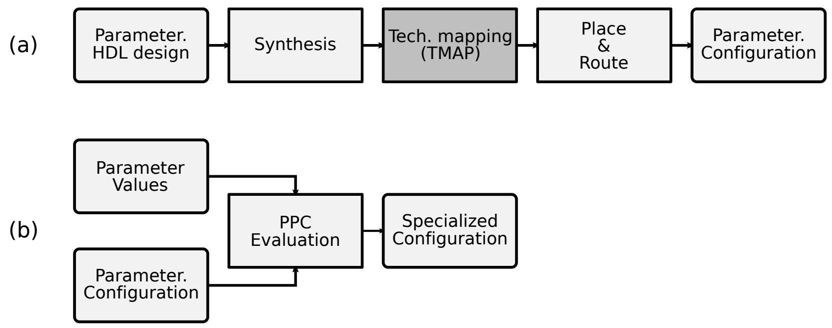


Fig. 3: Overview of our staged mapping tool flow. (a) The generic stage. (b) The specialization stage.

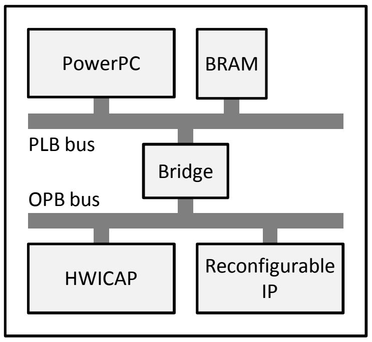


Fig. 4: Self-reconfiguring platform on a Xilinx Virtex-II Pro FPGA.

PowerPC of the Xilinx Virtex-II Pro FPGA, which ensures a tight connection to the FPGA fabric [19]. The connection between the configuration manager and the configuration memory is realized through the Xilinx HWICAP module, which provides the interface between the OPB bus and the FPGA's ICAP (Internal Configuration Access Port). To configure parts of the FPGA fabric (LUTs) after a parameter value has changed, the PowerPC evaluates the tuning functions (which are individual PPC instances for every TLUT), generates the new configuration, and sends this new configuration to the FPGA configuration memory through the ICAP port of the FPGA via the HWICAP module. The entire reconfiguration flow is thus executed within the system and no external source is needed to reconfigure the FPGA, nor to take the decision to reconfigure. Therefore, this system is a true *self-reconfiguring* system.

The self-reconfiguring platform shown in Fig. 4 is implemented using Xilinx XPS [20]. The XPS tool flow is implemented in a makefile and it is therefore easy to insert our tools in the flow. The adapted tool flow is shown in Fig. 5.

5.1 Generating a Master Configuration

We assume that the parameterizable HDL design contains a number of parameterizable modules and a number of non-parameterizable modules. A parameterizable VHDL module

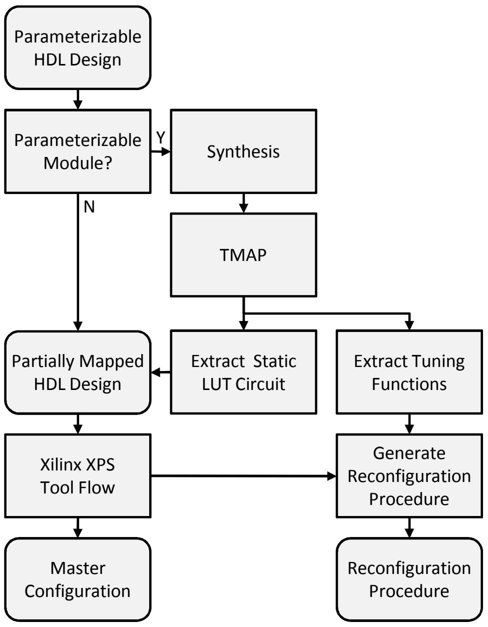

Parameterizable HDL Design
Parameterizable Module?
Y
Synthesis
N
TMAP
Partially Mapped HDL Design
Extract Static LUT Circuit
Extract Tuning Functions
Xilinx XPS Tool Flow
Generate Reconfiguration Procedure
Master Configuration
Reconfiguration Procedure

Fig. 5: Practical tool flow for mapping a parameterizable HDL design to a self-reconfiguring platform.

is nothing more than a regular VHDL description with annotations indicating which of the inputs are the parameter inputs. The parameterizable module of the 6:1 multiplexer example we will use in this section is shown in Fig. 6. The annotation `--PARAM` indicates that the select inputs are parameters. As the annotations are in a comment line, any conventional synthesis tool can be used to synthesize the circuit. We used Altera Quartus II because it can dump a .blif file that can then be used as input for our mapper TMAP [21], which maps the circuit to a TLUT circuit.

We make a distinction between parameterizable modules and non-parameterizable modules. Indeed, the Virtex-II Pro architecture is a very heterogeneous architecture compared to the homogeneous LUT architecture that TMAP targets. Therefore, using TMAP to map the full design would result in a very inefficient use of the Virtex-II Pro architecture. We thus limit the use of TMAP to the parameterizable modules,

```
entity mux6 is
port(
  s : in  std_logic_vector(2 downto 0); --PARAM
  i : in  std_logic_vector(5 downto 0);
  o : out std_logic);
end mux6;

architecture behavior of mux6 is
begin
  o <= i(conv_integer(s));
end behavior;
```

Fig. 6: Parameterizable VHDL module of the 6:1 multiplexer example.

as is shown in Fig. 5. The static LUT circuit of these modules is expressed in VHDL by directly instantiating LUTs in the VHDL module. Combining these modules with the non-parameterizable VHDL modules of the original design forms the partially mapped HDL design. This VHDL design can now be efficiently mapped to the Virtex-II Pro architecture by the Xilinx tools without corrupting the mapping done by TMAP. The result of this last mapping is the master configuration. This workaround could of course be avoided if the ability to map to TLUTs would be incorporated in the Xilinx mapper.

It is important to note that every LUT instantiated in VHDL is given a unique name. This enables our tools to find the LUT's location after place and route, see Section 5.2. Although it is not strictly necessary, we also lock the pins of the LUTs with the `lock_pins` attribute so that the router does not interchange the pins during routing. This greatly simplifies generating the reconfiguration procedure.

5.2 Generating the Reconfiguration Procedure

The reconfiguration procedure reconfigures all the TLUTs instantiated in a parameterizable module according to the parameter values that are passed as arguments to the procedure.

We need both the tuning functions of each TLUT and the location of each TLUT in order to do the reconfiguration upon a parameter change. The tuning functions for each TLUT are provided by TMAP, this is explained in detail in [21]. The LUT locations are harder to come by. On the Virtex-II Pro a LUT location is specified by a slice row, a slice column and whether it's the F or the G LUT of the slice [22]. Finding these locations for each instantiated LUT is done in the following way. The Xilinx tool flow generates a .NCD file that contains all the information on the mapped circuit including the location of the LUTs. This .NCD file is first converted to a .XDL file, a clear-text representation of the .NCD file, using the Xilinx XDL program [23]. We find the LUT locations in this .XDL file by searching the unique names given to the LUTs when they were instantiated in VHDL, as explained in Section 5.1.

A reconfiguration procedure is then generated as follows. For each of the TLUTs in a parameterizable module we

```
void L1( XHwIcap *hwIcap,
         Xuint8 S0, Xuint8 S1, Xuint8 S2) {

  Xuint8 truthTable[LUT_SIZE];
  truthTable [0] = !(0);
  truthTable [1] = !(S0 && S1);
  truthTable [2] = !(!S0 && S1);
  truthTable [3] = !(S1);
  truthTable [4] = !(S0 && !S1);
  truthTable [5] = !(S0);
  truthTable [6] = !((!S0 && S1) || (S0 && !S1));
  truthTable [7] = !( S0 || S1);
  truthTable [8] = !(!S0 && !S1);
  truthTable [9] = !((S0 && S1) || (!S0 && !S1));
  truthTable [10]= !(!S0);
  truthTable [11]= !(!S0 || S1);
  truthTable [12]= !(!S1);
  truthTable [13]= !(S0 || !S1);
  truthTable [14]= !(!S0 || !S1);
  truthTable [15]= !(1);

  XHwIcap_SetClbBits( hwIcap, 31, 45, G_LUT,
                      truthTable, LUT_SIZE);
}
```

Fig. 7: The TLUT reconfiguration procedure for LUT L_1 of our 6:1 multiplexer example. We assume that LUT L_1 is located in the G LUT of the slice at row 31 and column 45.

generate a TLUT reconfiguration procedure that takes the module parameter values as inputs, evaluates the tuning functions generated by TMAP and reconfigures the LUT. The TLUT reconfiguration procedure for LUT L_1 of our 6:1 multiplexer example is shown in Fig. 7. The code that evaluates the tuning functions of a TLUT is generated by simply translating the expressions produced by TMAP into C-style expressions.[3] When executed, these expressions result in a new truth table for the LUT. The reconfiguration of the LUT is then done by calling the procedure `XHwIcap_SetClbBits`, which is provided by Xilinx in the HWICAP module driver. This procedure takes the LUT location and the new truth table to reconfigure the LUT. In our example we assume that LUT L_1 is located in the G LUT of the slice at row 31 and column 45. The reconfiguration procedure for a module simply calls the TLUT reconfiguration procedure for each of the TLUTs of a module. The reconfiguration procedure for our 6:1 multiplexer example is shown in Fig. 8.

On a last practical note, we should warn the reader that, in the Virtex-II Pro family, reconfiguring a LUT will cause corrupted data in the SRL16s and LUT RAMs that are located in the same column. Therefore, placing TLUTs in the same columns as SRL16s or LUT RAMs must be avoided. This can be done using `AREA_GROUP` constraints. This is no longer an issue in the Virtex-5 family.

[3]It must be noted that, since the Virtex-II Pro family LUT configurations are stored in an inverted way, the configuration data must be inverted before configuring the LUTs [24].

```
void mux2w1 ( XHwIcap *hwIcap,
              Xuint8 S0, Xuint8 S1, Xuint8 S2) {
  L0(hwIcap, S0, S1, S2);
  L1(hwIcap, S0, S1, S2);
}
```

Fig. 8: The reconfiguration procedure for our 6:1 multiplexer example.

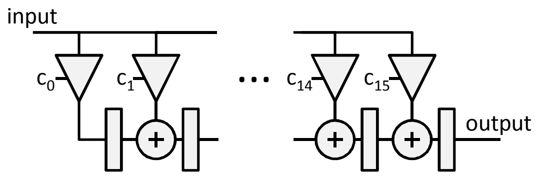


Fig. 9: 32-tap fully pipelined adaptive FIR filter.

6. Experiments and results

In this section, we apply tunable LUT mapping on more complex circuits. In Section 6.1 we create adaptive FIR filters that are adapted by means of reconfiguration and in Section 6.2 we create TCAMs of which the content is written by means of reconfiguration. Both designs are implemented on a Virtex-II Pro XC2VP30 using Xilinx ISE 9.2.

6.1 Adaptive FIR filter

Adaptive FIR filters that are adapted by means of reconfiguration can be created with TMAP by simply choosing the filter coefficients as the parameters of the design. In what follows we create this sort of adaptive filter for different numbers of taps and input widths and we compare them with conventional adaptive filters. The filters used are fully pipelined FIR filters as shown in Figure 9 for a 32-tap filter.

On the one hand, the filters are implemented using the conventional ISE 9.2 tool flow stating from RTL descriptions of the filters. Synthesis is done using Xilinx XST 9.2 with the default settings except for the multiplier style which we set to LUT. This way the multipliers are implemented using LUTs rather than the hardwired multipliers available in the Virtex-II Pro. This is necessary to allow a fair comparison between the conventional implementation and the TMAP implementation. Technology mapping (Xilinx MAP 9.2) and Place and Route (Xilinx PAR 9.2) are done using default settings. The number of LUTs and the maximum clock frequency for the filters implemented using ISE can be found in columns 3 and 4 of Table 1.

On the other hand, the filters are implemented using TMAP, again starting from RTL descriptions of the filters. This RTL description is first synthesized using Quartus II 7.2. Quartus is set to dump a .blif file after synthesis. This is possible due to the Quartus II University Interface Program (QUIP). Next, the .blif file is converted to an .aig file using ABC [14]. Together with a list of parameter inputs (the filter coeficients in this case) this .aig file is used as input for

Table 1: Hardware properties for a set of differend-sized adaptive FIR filters implemented without using dynamic reconfiguration (Conventional) and with using dynamic reconfiguration (TMAP). The numbers between brackets are relative compared to the conventional implementation.

Size		Conventional		TMAP			
Width	Taps	LUTs	f [MHz]	LUTs		f [MHz]	
8 bit	32	2641	80.84	1520	(0.58)	123.82	(1.53)
8 bit	64	5298	72.41	3056	(0.58)	89.06	(1.23)
8 bit	96	7954	65.41	4592	(0.58)	87.96	(1.34)
8 bit	128	10611	53.81	6128	(0.61)	74.23	(1.38)

a Java implementation of TMAP which produces both a PPC represented as a .aig file and a LUT structure. In this experiment, the LUT structure is represented as VHDL file that directly instantiates Virtex-II Pro LUTs and FFs [25]. Finally the LUT structure is implemented on the Virtex-II Pro using the Xilinx ISE 9.2 tool flow (XST 9.2, MAP 9.2 and PAR 9.2) with default settings. For more information on how to integrate TMAP with the ISE tool flow we refer to [13].

The number of LUTs and the maximum clock frequency for the dynamically reconfigurable filters can be found in columns 5 and 6 of Table 1. If we compare the number of LUTs in both implementations we see that the TMAP implementations need at least 39% fewer LUTs than the conventional implementation. We also see that the TMAP implementation can be clocked at least 23% and up to 53% faster than the conventional implementation.

Of course, the gain in area and speed of the FPGA hardware of our dynamically reconfigurable adaptive filters comes at the cost of a larger adaptation time. While the filter coefficients of the conventional adaptive filter can be changed by simply rewriting the registers that store the coefficients, the TMAP implementation requires us to both generate a specialized configuration for the FPGA by evaluating the PPC and write this configuration in the configuration memory of the FPGA. The total time needed to change the coefficients of the filter is called the specialization overhead, $t_{special}$. It contains both the time needed to evaluate the PPC, t_{eval}, and the time to reconfigure the FPGA, t_{reconf}. In what follows we discuss the specialization overhead in detail.

As shown in Figure 2, the evaluation of the PPC is done on an Instruction Set Processor (ISP). In our case, we use the PowerPC which is hardwired on the Virtex-II Pro FPGA. Efficiently evaluating a Boolean network on an ISP is an area of research by itself, and is beyond the scope of this paper. However, in order to give an estimate of the evaluation time we have implemented a simple compiled evaluation technique. In compiled evaluation, a dedicated function is created that takes the input values of the network as its arguments and returns the output values of the network. In our case, the network is the PPC created by TMAP, the input values are the parameter values (the coefficients of the filter) and the output values are the truth tables for the TLUTs.

Starting from the PPC, an evaluation function is created by first generating a C function and then compiling it for the PowerPC. We generate the C code of the evaluation function by traversing the PPC in topological order from the inputs towards the outputs. For every node a statement is added to the C code. The statement calculates the output value of the node from the output value of its predecessors and stores the output in an local array (`node`). The size of the local array is minimized by freeing its elements when they are no longer needed and by always storing a node output value at the smallest available index. E.g. if left predecessor is inverted, the smallest available index is 3 and and the output values of the left and right predecessors are respectively stored at indices 9 and 6, the expression would be `node[3] = !node[9] && node[6];`.

Unfortunately, when we generate an evaluation function as described above for the complete flattened FIR filter, this leads to very large evaluation function and poor evaluation time. However, many designs, including our FIR filters, contain hierarchy and have a repetitive nature that can be used to build a more compact evaluation function. In our filters, one multiplier is instantiated for every tap. Instead of generating one flat evaluation function for the complete FIR filter, we generate an evaluation function for one multiplier, as described above, and build the FIR evaluation function by calling this function for each instantiation of the multiplier. We could even further optimize this by calculating up to 32 of the multiplier evaluation functions at a time by using bitwise logic operations and packing 32 Boolean values in each 32-bit word. Although we have built the FIR evaluation function manually for our experiments, it could easily be synthesized automatically from the hierarchy found in the HDL design.

In our experiment we created the evaluation function as described above for each of the FIR filters and executed it on the PowerPC. The PowerPC was clocked at 300 MHz and both the instruction and data caches were enabled. The evaluation time, t_{eval}, and the size of the compiled evaluation function S_{eval} are shown in Table 2. We see that the evaluation time for our adaptive filters takes in the order several hundreds of μs depending on the size of the filter. The ratio of the evaluation time and the number of AND nodes in the PPC shows that the evaluation time of the filters is very linear in the size of the PPC. The size of the evaluation functions is about 15 kB and slowly grows as the number of taps increases. The program size is almost independent of the size of the PPC because one multiplier evaluation function is created, that is reused to generate the truth tables for each of the multipliers in the design.

After evaluating the PPC, the PowerPC needs to write the calculated truth tables in the configuration memory of the FPGA. The PowerPC can access the configuration memory of the Virtex-II Pro from within the FPGA fabric through the ICAP (Internal Configuration Access Port) which we connected to the bus of the PowerPC. To reconfigure the FPGA, the PowerPC needs to send a partial bitstream to the ICAP which can be done at a maximum rate of 66 MB/s. The size of the bitstreams, S_{bit}, and the reconfiguration time, t_{reconf}, are shown in column 5 and 6 of Table 3. The reconfiguration time ranges from 1 ms to a maximum of 1.75 ms depending on the size of the filter. As can be seen, the reconfiguration time is not linear in the number of TLUTs as one could expect, but linear in the number of frames that need to be reconfigured. This is because the atom of reconfiguration of the Virtex-II Pro is not a LUT truth table but a frame [26]. All the truth tables of a column of CLBs (Configurable Logic Blocks) are stored in only two frames. If only one LUT in a CLB column changes half of the LUTs in that column need to be reconfigured. Because it's frames are smaller, the importance of this overhead is reduced for the Virtex-5 [27].

Finally, the total specialization overhead which is the sum of the evaluation time and the reconfiguration time, is shown in Table 4. As can be seen, the specialization time is of the order of a few ms, depending on the size of the filter. We can thus exploit the area and clock frequency benefits of our adaptive filters (Table 1) as long as the time in between coefficient changes is a few orders of magnitude higher that the specialization overhead.

Table 2: Evaluation of the PPC of different-sized adaptive FIR filters on the hardwired PowerPC of the Virtex-II Pro.

Size		Evaluation Time			Program Size
Width	Taps	$\lvert PPC \rvert$	t_{eval} [μs]	$\frac{t_{eval}}{\lvert PPC \rvert}$ [$\frac{ns}{AND}$]	S_{eval} [B]
8 bit	32	28672	317	11.06	14150
8 bit	64	57344	634	11.06	14918
8 bit	96	86016	951	11.06	15686
8 bit	128	114688	1268	11.06	16454

Table 3: Reconfiguration of different-sized adaptive FIR filters through the ICAP of the Virtex-II Pro.

size		Reconfiguration Time			
Width	Taps	TLUTs	frames	S_{bit} [B]	t_{reconf} [μs]
8 bit	32	768	52	66573	1009
8 bit	64	1536	88	111357	1687
8 bit	96	2304	92	115493	1750
8 bit	128	3072	91	114669	1737

6.2 Ternary Content Addressable Memory

In conventional memories, the read operation returns the data associated with a given address. The read operation of a Content Addressable Memory (CAM) does the opposite: it finds the address associated to a given data value. In both cases, the write operation stores a given data value at

Table 4: Specialization overhead of different-sized adaptive FIR filters implemented on the Virtex-II Pro.

size		Specialization Overhead		
Width	Taps	t_{eval} [μs]	t_{reconf} [μs]	$t_{special}$ [μs]
8 bit	32	317	1009	1326
8 bit	64	634	1687	2321
8 bit	96	951	1750	2701
8 bit	128	1268	1737	3005

a given address. CAMs have many applications [28]. The most important commercial application is packet forwarding in network routers [29].

A TCAM (Ternary CAM) is a special kind of CAM that stores ternary patterns instead of pure data. Each digit in a ternary pattern can either be zero, one or don't care. The digits are represented by two bits: the data bit and the mask bit. A full pattern entry in the TCAM is represented by two bit vectors (the data and the mask) and one bit which indicates whether the entry contains a pattern or not. When new input data is provided to the TCAM, it simultaneously compares this data to all stored patterns. The incoming data matches a pattern if all bits of the incoming data for which the corresponding mask bit of the pattern is zero are equal to the corresponding value bit of the pattern.

In a conventional TCAM implementation, the pattern entries will be provided by flip-flops (FFs) arranged as a memory. Each memory element uses a FF in the FPGA because all data needs to be accessed in every clock cycle. In our reconfigurable implementation, these inputs are the parameters of the design and will thus be provided by means of reconfiguration. In important applications such as Internet core routers, this approach is feasible, since the update rate is usually rather limited (at the very most a few hundred updates per second [30]), while the read rate is orders of magnitude higher (up to several millions of packets per second).

The problem with TCAMs is that their implementation requires many FPGA resources, even for small TCAMs. When we synthesize a description of a full TCAM (256 entries of 32-bit) using ISE for a Virtex II Pro, the implementation requires 16,874 FFs and 10,441 4-input LUTs and can be maximally clocked at 69 MHz. This is true for different sizes of the TCAM (see Table 5).

These resource requirements can be drastically reduced with the use of TMAP. In the TMAP design, we chose the entry array of the TCAM as the parameter input of the design, by adding the `-PARAM` annotation. This means that the patterns stored in the TCAM will be changed by means of reconfiguration. When we map this design using TMAP it only requires 3,497 LUTs (a reduction by 67%), the maximum clock frequency rises from 69 MHz to 90 MHz (a gain of 30%) and the number of FFs is reduced dramatically from 16,874 to only 226 (see Table 5). This reduction in FFs is possible because the pattern information is no longer stored in FFs that are part of the FPGA fabric, but in the memory elements of the configuration memory that stores the truth tables of the LUTs. Only a few FFs are left for some output registers.

Because of the importance of TCAMs and the high resource usage of architecture independent HDL implementations, FPGA vendors offer TCAM constructor software which constructs TCAM structures that are highly optimized for a specific architecture [28], [31]. The designer can generate such a structure using the software and then instantiate it in his design. A good example of such a generator is the SRL16 TCAM generator [11] embedded in Xilinx Coregen, which generates TCAM structures that are very similar to the TCAMs that TMAP synthesizes. The results for the SRL16 TCAM are shown in Table 5. As can be seen the SRL16 TCAM (256 entries of 32-bit) is 45% larger and clocks 34% slower than the TMAP design. This is mainly due to the infrastructure needed to write new entries in the TCAM.

Again, the gain in area and speed of the FPGA hardware of our dynamically reconfigurable TCAM comes at the cost of a larger time to write an entry. While in the ISE implementation an entry can be rewritten in one clock cycle and in the SRL16 implementation in 16 clock cycles, the TMAP implementation requires us to both generate a specialized configuration for the FPGA by evaluating the PPC and write this configuration in the configuration memory of the FPGA. In the next experiment, we measured the specialization overhead for the TMAP implementation. As is explained in Section 6.1, the evaluation of the PPC is done on the embedded PowerPC and the reconfiguration is done using the ICAP of the Virtex-II Pro. We did the measurement both in the case only one entry needs to be rewritten and in the case all entries are rewritten. The results are shown in Table 6. If only one entry is written, the reconfiguration time depends on the way the TLUTs of the entry are placed on the FPGA, because the placement determines the number of frames that need to be reconfigured. Table 6 shows the reconfiguration time for the worst case entry. For the largest TCAM (256 entries of 32 bit), reconfiguring one entry takes 245 μs in worst case and reconfiguring the full TCAM takes 1716 μs. More details can be found in the table.

The disadvantage of using generator software is that it results in architecture dependent designs, because the TCAM structures internally instantiate architecture specific resources. Our TMAP design does not have that problem as its VHDL code is architecture independent. The same design can be mapped to several FPGA architectures by simply changing the mapper. Of course these mappers must have TLUT capability in order to benefit form the resource reduction. To strengthen this point we have also mapped our code to architectures with different LUT sizes. The LUT usage for $K = 3$, $K = 4$ and $K = 5$ can be found in Table 7. In the table one can clearly see that for the TCAMs the relative area gain improves when the LUT size increases.

Table 5: Comparison of different implementations of a TCAM on a Virtex-II Pro. (ISE) Synthesis from behavioral VHDL using ISE 9.2. (SRL16) Generated with Xilinx Coregen. (TMAP) Synthesis from behavioral VHDL using TMAP.

Design		ISE			SRL16			TMAP		
Width	Entries	LUT	FF	f_{max} [MHz]	LUT	FF	f_{max} [MHz]	LUT	FF	f_{max} [MHz]
16	128	2516	4302	86.72	1504	127	79.26	1095	56	88.72
16	256	5100	8577	74.36	2886	130	68.24	2217	85	83.51
32	128	4569	8419	79.97	2664	223	68.13	1735	237	95.57
32	256	10441	16874	69.01	5070	226	59.69	3497	259	90.00

Table 6: Specialization overhead of different-sized TCAMs implemented on the Virtex-II Pro. (One Entry) Only one of the TCAM entries is written. (All Entries) All the entries of TCAM are written.

Design		One Entry (Worst Case)					All Entries				
Width	Entries	S_{eval} [B]	t_{eval} [μs]	frames	t_{reconf} [μs]	$t_{special}$ [μs]	S_{eval} [B]	t_{eval} [μs]	frames	t_{reconf} [μs]	$t_{special}$ [μs]
16	128	7362	1.50	4	104	106	7446	190	41	795	985
16	256	7362	1.50	4	104	106	7446	380	52	1034	1414
32	128	9750	2.84	9	205	208	9834	360	49	946	1306
32	256	9750	2.84	9	242	245	9834	720	52	996	1716

Table 7: Several TCAMs mapped to different-sized LUTs: $K = 3$, $K = 4$ and $K = 5$.

Design		$K = 3$			$K = 4$			$K = 5$		
Width	Entries	Conv.	TMAP		Conv.	TMAP		Conv.	TMAP	
16	128	3637	1604	(0.44)	2516	1095	(0.44)	2471	867	(0.35)
16	256	7278	3197	(0.44)	5100	2217	(0.44)	4863	1724	(0.36)
32	128	6958	3022	(0.43)	4569	1735	(0.38)	4780	1527	(0.32)
32	256	13919	6013	(0.43)	10441	3497	(0.34)	9337	3018	(0.32)

7. Conclusions

Run-time hardware reconfiguration provides ample opportunities for optimizations of an implementation in time intervals in between two parameter changes. In this paper we introduced a tool flow that automatically generates a dynamic data folding implementation starting from an RT-level HDL design. Its main contribution is a novel technology mapper called TMAP. The mapper maps Boolean circuits to Tunable LUTs (TLUTs), these are LUTs of which the truth table is expressed as function of the parameter inputs. We have effectively integrated our tool flow in the Xilinx XPS tool flow that targets Virtex-II Pro FPGA devices. We used the embedded PowerPC of the Virtex-II Pro device as reconfiguration manager. On top of this, in our DDF architecture, specialized configurations are also generated on the fly by evaluating Boolean functions. We expressed these functions as a single Boolean network, which opened up the possibility of using well-known combined Boolean optimization techniques. Our approach is validated by implementing adaptive FIR filters and Ternary Content-Addressable Memories (TCAMs) on a Virtex-II Pro.[4] We show large reductions in the number of LUTs (39% for the FIR filter and 66% for the TCAMs) and significant improvements of the maximum clock frequency (38% for the FIR filter and 30% for the TCAM). The specialization of both designs was done using the embedded PowerPC and the ICAP of the Virtex-II Pro. The total time needed to change the coefficients of the filter is 1.74 ms and the content of the TCAM can be rewritten in 1.72 ms. FIR filters and TCAMs are only two of a large class of applications that can benefit from DDF. Because of its general applicability and the RT-level design, our technique makes designing DDF systems feasible for many applications. Other applications that may benefit from our DDF technique are: encryption algorithms like AES and DES, template matching [32], regular expression matching [33], DNA aligning [34], [35], serial fault emulation [36] and many others.

[4]The FIR filter has 128 taps with 8-bit wide coefficients. The TCAM has 256 entries that are 32 bit wide.

References

[1] S.V. Gheorghita et al., *System-scenario-based design of dynamic embedded systems.*, ACM Transactions on Design Automation of Electronic Systems, 14(1):1–45, 2009.

[2] G. Palermo, C. Silvano, and V. Zaccaria. *Robust optimization of SoC architectures: A multi-scenario approach.* In Proc. of the IEEE Workshop on Embedded Systems for Real-Time Multimedia, 2008.

[3] J. Paul, D. Thomas, and A. Bobrek. *Scenario-oriented design for single-chip heterogeneous multiprocessors.* In IEEE Trans. on Very Large Scale Integration Systems, 14(8), PP. 868–880, 2006.

[4] K. Compton and S. Hauck. *Reconfigurable computing: a survey of systems and software.* ACM Computing Survey, 34(2): pp. 171–210, 2002.

[5] K. Bruneel and D. Stroobandt. *Reconfigurability-aware structural mapping for LUT-based FPGAs.* In 2008 International Conference on Reconfigurable Computing and FPGAs (ReConFig). IEEE, Piscataway, NJ, USA, 223–8.

[6] L. Benini, D. Bertozzi, and M. Milano. *Resource management policy handling mulitple use-cases in MPSoC platforms using constraint programming.* In Logic Programming, vol. 5366, pp. 470 – 484, 2008.

[7] E.W. Brião, D. Barcelos, and F.R. Wagner. *Dynamic task allocation strategies in MPSoC for soft real-time applications.* In Proc. of the conference on Design, Automation and Test in Europe (DATE), pp. 1386–1389, 2008.

[8] P. Hölzenspies, J. Hurink, J. Kuper, and G. Smit. *Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSoC).* In Proc. of the Conf. on Design, Automation and Test in Europe (DATE), 2008.

[9] V. Nollet. *Run-time Management for Future MPSoC platforms.* Ph.D. thesis, TU Eindhoven, 2008.

[10] P. Yang and F. Catthoor; *Pareto-optimization-based run-time task scheduling for embedded systems.* In Proc. of the Int. Conference on Hardware/Software Codesign and System Synthesis, 2003.

[11] J.-L. Brelet and B. New, B. *XAPP203: Designing Flexible, Fast CAMs with Virtex Family FPGAs.* Xilinx. 1999

[12] M.J. Wirthlin, *Constant coefficient multiplication using look-up tables.* Journal of VLSI Signal Processing, vol. 36, no. 1, PP. 7–15, 2004.

[13] K. Bruneel, F. Abouelella, and D. Stroobandt. *Automatically mapping applications to a self-reconfiguring platform.* In Proceedings of Design, Automation and Test in Europe, K. Preas, Ed. Nice, 964–969. 2009

[14] *ABC: A System for Sequential Synthesis and Verification.* Berkeley Logic Synthesis and Verification Group.

[15] A. Derbyshire, T. Becker, and W. Luk. *Incremental elaboration for run-time reconfigurable hardware designs.* In CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems. ACM, New York, NY, USA, 93–102. 2006

[16] K. Bruneel and D. Stroobandt. *Automatic generation of run-time parameterizable configurations.* In Proceedings of the 2008 International Conference on Field Programmable Logic and Applications, U. Kebschull, M. Platzner, and T. J., Eds. Kirchhoff Institute for Physics, Heidelberg, 361–366. 2008

[17] K. Bruneel and D. Stroobandt. *TROUTE: a reconfigurability-aware FPGA router.* In Lecture Notes in Computer Science. Vol. 5992. Springer Verlag Berlin, Berlin, Germany, 207–218. 2010

[18] K. Bruneel, W. Heirman and D. Stroobandt. *Dynamic Data Folding with Parameterizable FPGA COnfigurations.* To be published in ACM Trans. on Design Automation of Embedded Systems, 2011.

[19] B. Blodget, P. James-Roxby, E. Kelle, S. McMillan, and P. Sundararajan, *A selfreconfiguring platform*, International Conference on Field-Programmable Logic and Applications, pp. 565– 574, 2003.

[20] *Embedded System Tools Reference Manual*, Xilinx.

[21] K. Bruneel and D. Stroobandt, *Automatic generation of run-time parameterizable configurations*, in Proceedings of the International Conference on Field Programmable Logic and Applications, 2008, pp. 361–366.

[22] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, Xilinx.

[23] J.-B. Note and Éric Rannaud, *From the bitstream to the netlist*, in FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays. New York, NY, USA: ACM, 2008, pp. 264–264.

[24] A. Upegui and E. Sanchez, *Evolving hardware by dynamically reconfiguring Xilinx FPGAs*, in Evolvable Systems: From Biology to Hardware, ser. LNCS, J. M. et al., Ed., vol. 3637. Berlin Heidelberg: Springer-Verlag, 2005, pp. 56–65.

[25] *Virtex-II Pro Libraries Guide for HDL Designs.* Xilinx. 2008.

[26] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide.* Xilinx. 2007

[27] *Virtex-5 FPGA Configuration User Guide.* Xilinx. 2010

[28] *Application Note 119: Implementing High-Speed Search Applications with Altera CAM.* Altera. 2001

[29] K. Pagiamtzis and A. Sheikholeslami. *Content-addressable memory (CAM) circuits and architectures: A tutorial and survey.* IEEE Journal of Solid-State Circuits 41, 3, 712–727. 2006

[30] C. Labovitz, G. Malan, and F. Jahanian. *Internet routing instability.* IEEE/ACM Transactions on Networking 6, 5 (Oct.), 515 –528. 1998

[31] *DS253: Content-Addressable Memory v6.1.* Xilinx. 2008

[32] M. J. Wirthlin and B. L. Hutchings. *Improving functional density through run-time constant propagation.* In FPGA '97: Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays. ACM, New York, NY, USA, 86–92. 1997

[33] B. Hutchings, R. Franklin, and D. Carver. *Assisting network intrusion detection with reconfigurable hardware.* In Proceedings of the 10th annual IEEE symposium on field-programmable custom computing machines. 111–120. 2002

[34] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, P. Athanas, and A. Dickerman. *A run-time reconfigurable system for gene-sequence searching.* VLSI Design, International Conference on 0, 561. 2003.

[35] Y. Yamaguchi, Y. Miyajima, T. Maruyama, and A. Konagaya. *High speed homology search using run-time reconfiguration.* In FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications. Springer-Verlag, London, UK, 281–291. 2002.

[36] L. Burgun, F. Reblewski, G. Fenelon, J. Berbier, and O. Lepape. *Serial fault emulation.* In DAC '96: Proceedings of the 33rd annual Design Automation Conference. ACM, New York, NY, USA, 801–806. 1996.

SESSION

REGULAR PAPERS

Chair(s)

PROF. ROMAN LYSECKY

A Transparent and Adaptable Multiple-ISA Embedded System

Jair Fajardo Junior, Mateus B. Rutzig, Luigi Carro, Antonio C. S. Beck
{jffajardoj, mbrutizg, carro, caco}@inf.ufrgs.br
Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil

Abstract - *In these days, every new added hardware feature must not change the underlying instruction set architecture (ISA), in order to avoid adaptation or recompilation of existing code. Therefore, Binary Translation (BT) opens new possibilities for designers, previously tied to a specific ISA and all its legacy hardware issues, since it allows the execution of already compiled applications on different architectures. To overcome the BT inherent performance penalty, we propose a new mechanism based on a dynamic two-level binary translation system. While the first level is responsible for the BT de facto to an intermediate language, the second level optimizes the already translated instructions to be executed on the target architecture. The system is totally flexible, supporting the porting of radically different ISAs and the employment of different target architectures. This paper presents the first effort towards this direction: it translates code implemented in the x86 ISA to MIPS assembly (the intermediate language), which will be optimized by the target architecture: a dynamically reconfigurable architecture. We show that is possible to maintain binary compatibility with performance improvements when compared to native execution.*

Keywords: Binary Translation, Reconfigurable Systems

1 Introduction

With the constant growth of the embedded systems market, more complex applications have been developed to fulfill consumer needs. At the same time, technological development has already started to stagnate as a result of the decline in Moore's law [1], and one can observe that the processing capabilities of traditional architectures are not growing in the same pace as before [2]. In this scenario, new alternatives are necessary to minimize this problem. However, the support for binary compatibility, so that the large quantity of tools and applications already deployed can be reused, is an important requirement to introduce new processors into the market. With this in mind, companies develop their products focusing on the improvement of a given architecture that will execute the same Instruction Set Architecture (ISA) as before. Nevertheless, this need for compatibility imposes a great number of restrictions to the design team.

Binary translation systems can give back to designers the freedom previously lost, since they do not need to be tied to a specific ISA anymore. Therefore, the ideal scenario would be as shown in Figure 1: the execution of instructions compatible to any given ISA on the very same underlying architecture. However, the maintenance of binary compatibility only is not enough to handle market needs. It is also necessary to translate code execution in a competitive fashion, when compared to native execution [3]. This way, the concept of binary translation must also be tightly connected to code optimization and acceleration [4].

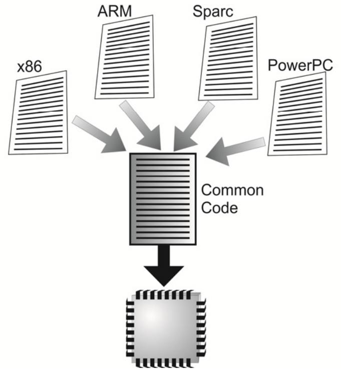


Fig. 1. Ideal scenario for current embedded systems.

With the aforementioned issues in mind, this work proposes a new approach based on a dynamic two-level binary translation system that, besides maintaining binary compatibility, amortizes its costs. An overview of the proposed system is presented in Figure 2. The first BT level is responsible for translating the source code to an intermediate (common) code, as any conventional BT machine would do. The second BT level is responsible for transforming the already translated code (intermediate code) to be executed on the target architecture. With the two-level BT mechanism, and having a clear interface between the translation and the optimization levels, another advantage emerges: during design time, by only changing the first BT level it is possible to execute different ISAs in a completely transparent fashion to the second BT level, thus greatly facilitating the porting of radically different ISAs without the need for changing the underlying architecture, as long as different first BT level layers are available. In the same way, it is possible to switch to another target architecture, according to the application needs or to the available architecture at the moment.

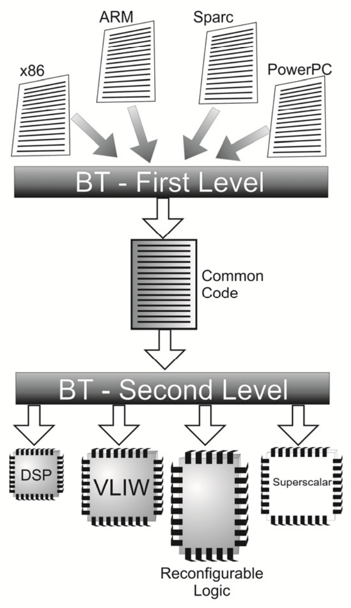


Fig. 2. Proposed Approach.

In the case study presented in this paper, which presents the first step towards this objective, x86 code is translated at the first BT level, MIPS assembly is used as the intermediate language, and a dynamically reconfigurable system [5] is used as the optimization machine, as shown in Figure 3a. This way, performance improvements are reached because both BT mechanisms are completely implemented in hardware for fast translation and minimum performance overhead; and once a sequence of code has passed through the two levels, the next time it is found both BT levels will be skipped (both translations will not be necessary, as illustrated in Figure 3b), and the reconfigurable array will be directly used, as long as there memory available. This is another advantage of the proposed technique and will be explained in more details later.

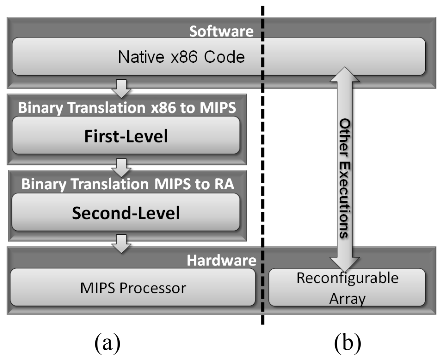


Fig. 3. Execution layers of the case study of the proposed approach.

The rest of this paper is organized as follows. In section 2, we show some related binary translation architectures, with a brief explanation about their operation. In the next section, an overview about the proposed architecture is given. In section 4 we present the experimental results and a discussion on these tests. In Section 5 we conclude this article and discuss future works.

2 Related Work

2.1 BT Systems

Binary translation systems have been used mainly because companies need to reduce the time-to-market and maintain backward software compatibility. They can work at different layers in a computing architecture: as if it was another regular application, visible to the user; or yet implemented in hardware, working below the operating system [7]. One example is Rosetta [8]: used into Apple systems to maintain compatibility between the PowerPC and x86. It works in the application layer with the sole purpose of maintaining binary compatibility, causing a great overhead. Another case is the FX!32 [9] [10] that allows 32-bit programs to be installed and executed like an x86 architecture running Windows NT 4.0 on Alpha systems. The FX!32 is composed of an emulator and a binary translator system. The emulator performs code conversion, and also provides profiling information at run-time. The binary translator uses the profiling information to generate optimized images through identified hotspots into the code, saving them for future reuse and avoiding excessive run time overhead. As other examples, the HP Dynamo [25] analyzes the application at runtime in order to find the best parts of the software for the BT process, while the Daisy architecture uses BT at runtime to better exploit the ILP of a PowerPC application, transforming parts of code to be executed on a VLIW micro architecture [26].

The Transmeta Crusoe processor [11] had the main purpose of translating x86 code to execute onto a VLIW processor, reducing power consumption and saving energy. In this case, the BT is implemented in software, but the Crusoe hardware (a VLIW processor) was designed to speed up the BT process with minimum energy, which decreases the translation overhead. The Godson3 processor [12] has the same goal as the Transmeta Crusoe: using a software layer for binary translation (QEMU), it converts the x86 to MIPS instructions. However, it uses a different strategy to optimize the running program. Godson3 is a scalable multicore architecture, which uses a mix between a NOC (network on-chip) and a crossbar system for its communication infrastructure. This way, up to 64 cores are supported. Each core is a modified superscalar MIPS to assist the dynamic translation. Therefore, Godson3 can achieve satisfactory execution time for applications already implemented and deployed in the field. However, the cost of having several superscalar processors is not small.

In this case study, instead of using Superscalar or VLIW processors, we use reconfigurable logic as the main optimization mechanism. Reconfigurable systems have already proven to accelerate software and reduce energy consumption, showing gains over both systems [14][15]. Moreover, it is common sense that as the more the technology shrinks, the more an important characteristic of reconfigurable systems is highlighted: regularity – since this will impact the reliability of printing the geometries employed today in 65 nanometers and below [16]. Besides being more predictable, regular circuits are also low cost, since the more customizable the circuit is, the more expensive it becomes. This way, regular fabric could solve the mask cost and many other issues such as printability, power integrity and other aspects of the near future technologies.

2.2 Implementation

The binary translation process in software is more flexible due to the possibility to execute the BT system on other processors by recompiling the translator. However, it causes a huge overhead in execution time [13]. On the other hand, the implementation of the BT in hardware amortizes the translation overhead. In this case, the flexibility is strongly reduced: the hardware translation is hence tied to a specific ISA. Consequently, there is no opportunity to migrate to a new ISA or target architecture (or a new version of them) because the hardware was specifically tailored to that system. As a meet in the middle approach considering the two aforementioned methods, some BT systems present some kind of hardware modification to give better support for the software execution of the BT system. That is the case of Godson and Crusoe. Nevertheless, these works still rely on software for the main binary translation mechanism.

Our proposed approach is different because, besides being totally implemented in hardware, with the fastest translation speed, it uses a two-level BT mechanism: the first level is responsible for translating binary code from the source to the target processor, while the second is responsible for the code optimization. Since there is a well-defined interface between both, there is the possibility of easily ISA or target architecture migration at design time by only changing the correspondent level of the BT system. In this way, hardware modifications can be fine-tuned to several markets with different requirements. Therefore, comparing the proposed technique with other BT implementations, our main contributions are:

- Amortized performance overhead in the translation from the source to the target machine, because it is implemented in hardware, so it is faster than if it were implemented in software;
- Performance gains when compared to the execution of the original code in the source machine, since an optimization mechanism is used (in this case, a dynamically reconfigurable system);
- Flexibility through the employment of the two-level BT system, making it easier to migrate to another ISA or target architecture (or update them to a new version of the family), so the system has almost the same flexibility as if it were implemented in software. The next section details some implementation aspects of our system.

3 System Overview and Operation

Figure 4 gives a general overview of the system. The first BT level represents the hardware to make x86 to MIPS translations. It interfaces the memory and the rest of system, which is composed of the second-level BT mechanism, a special cache, a MIPS processor and a dynamically reconfigurable array. The BT in the second level analyzes the MIPS code at run-time and uses the reconfigurable logic to optimize and execute the hotspots found in the code. The system works like a native x86 processor, but with an additional possibility to run MIPS code too.

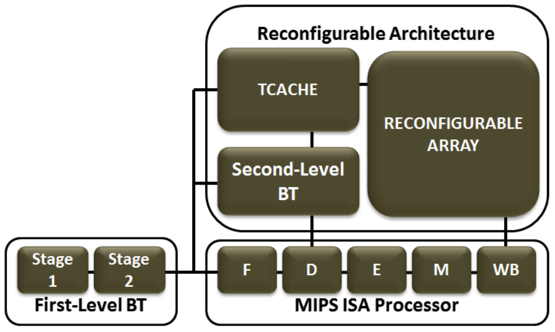


Fig. 4. Case Study Implementation.

3.1 Architecture Operation

Let us consider an application compiled using the x86 ISA that will be executed for the first time. Initially, the first level starts to fetch instructions from memory. As the first level is translating the code, the MIPS processor actually executes the instructions. In this level there are no translations savings for future reuse: all the data is processed at run-time by the first BT level in order to maintain small storage overhead. It also must be said that the same code, but in the optimized form, will be saved by the second-level for future reuse, as we shall see next. The second BT level analyzes the interpreted code (already in MIPS ISA) during execution. When a hotspot is identified, this level generates and saves a configuration of the reconfigurable array for that hotspot in a special cache (TCache), indexed by the x86 Program Counter (PC). The next time a chuck of x86 code that has already been transformed to MIPS and been optimized to the reconfigurable array is found, the equivalent configuration if fetched from the TCache. Then, the first BT level, MIPS processor and the second level mechanism are stalled, and the reconfigurable array starts its reconfiguration and execution.

Therefore, once a sequence of x86 instructions that were translated from the x86 to MIPS ISA was found and, after that, also became a configuration for the array, none of the BT mechanisms neither the processor need to work. As sequences of instructions are executed and translated, and the TCache is being filled, the impact of the two levels of BT are amortized and the performance gains provided by the array starts to appear. In the next subsections the whole system is explained in more details.

3.2 First BT Level

In the current implementation, it is possible to translate 50 different instructions in a total of 150 considering the IA32 ISA, with all addressing modes supported. The implemented subset is enough to compile and execute all the benchmarks tested. Segmentation is emulated, but there is no support for paging. Interruptions, and other multimedia instructions, such as the MMX and SSE, are still not implemented. The First BT level is composed of four different hardware units, with two pipeline stages: Translation, Mounting, Program Counter and Control Units.

The main component of the system is the Translation Unit. It is responsible for fetching x86 instructions from the memory, analyzing their format in order to classify them according to the type, operators, and addressing mode and generating the equivalent MIPS instructions. It takes one or more cycles to perform such operations. This unit is constituted mainly of a ROM memory that holds all possible equivalent MIPS instructions translations. For this reason, it concentrates the major part of the BT system area. Besides that, this unit provides some information to the other auxiliary units, such as: number of generated MIPS instructions, quantity of bytes to calculate the next PC and the type of instructions (e.g. logical operation, conditional or unconditional jumps, etc).

The role of the Mounting Unit is to provide an interface between the processor and the BT mechanism, by fetching all the equivalent MIPS instructions in a parallel fashion from the Translation unit and sending them serially to the MIPS processor, making BT mechanism behave as if it were a regular memory. The Mounting Unit is composed of a queue of registers in which each MIPS instruction is allocated. As instructions are processed, this unit constantly sends to the other ones the number of occupied slots in its queue, in order to guarantee that it will not empty and the MIPS processor will not stall. The Program Counter Unit was developed to calculate the address of the next x86 instruction that must be fetched from memory. In opposite to the MIPS instructions, x86 instructions have different sizes, so the x86 and MIPS addresses translation cannot be considered on a one-to-one basis. Finally, the function of the Control Unit is to keep the timing and consistency of information between the other units by using the information flags found in each unit. Through this information, the control unit decides the behavior for all the system, such as the fetch of a new instruction from memory at the instant there are free slots in the queue in the Mounting Unit; or the need for the calculus of a new x86 PC when the instruction (branch or regular) is fetched from memory.

3.3 Extended MIPS

The great advantage of using the MIPS ISA is the regularity of code with well-known behavior, making it easy to translate another ISA to this one. However, the translation of a complex ISA such as the x86 to MIPS is inefficient, because in several times one x86 instruction is converted to many MIPS instructions. For example, in X86 instructions it is possible to use the memory contents as operators in arithmetic instructions. Furthermore, there are flag registers, which are automatically updated in most of the arithmetic/logical operations, so these can be used in branch instructions. Such flags are not supported in the MIPS architecture. In this case, more than 20 instructions would be necessary per x86 instructions to correctly emulate these flags on the MIPS processor. The same can be considered for segment addressing modes and so on for several other constructs. Therefore, to lower this overhead, the MIPS processor was extended to give hardware support to these issues, but still maintaining compatibility with the standard code, as follows:

- *Byte Manipulation* – Several operations that occur in x86 code are based on manipulation of variables with 8, 16 and 32-bit in a register. As the MIPS processor only executes 32-bit operations, a special hardware was added to manipulate small variables the same way as x86 processors do, avoiding the need of using mask operations to insert or extract information to/from 32-bit registers.
- *Address Mode* – The MIPS is a load-store architecture. In contrast to the X86 ISA, which supports several addressing modes, the MIPS supports only the Base (register) + Index (immediate) addressing mode. To reduce this gap, the MIPS was extended, so Base (register) + Index (Register) and Base + Index + Immediate (byte) operations are possible.
- *EFlags*– Additional hardware that generates the flag values and stores the results into a mapped register in the MIPS processor was implemented.

3.4 Reconfigurable Array

The reconfigurable unit [5] is a dynamic coarse-grain array tightly coupled to the processor [27]. It works as an additional functional unit in the execution stage of the pipeline, using similar approach as Chimaera [28]. This way, no external accesses (in respect to the processor) to the array are necessary. An overview of its general organization is shown in Figure 5. The array is two dimensional, and each instruction is allocated in an intersection between one row and one column. If two instructions do not have data dependences, they can be executed in parallel, in the same row. Each column is homogeneous, containing a determined number of ordinary functional units of a particular type, e.g. ALUs, shifters, multipliers etc. Depending on the delay of each functional unit, more than one operation can be executed within one processor equivalent cycle. It is the case of the simple arithmetic ones. On the other hand, more complex operations, such as multiplications, usually take longer to be finished. The delay is dependent of the technology and the way the functional unit was implemented. Load/store (LD/ST) units remain in a different group of the array. The number of parallel units in this group depends on the amount of ports available in the memory. The current version of the reconfigurable array does not support floating point operations. For the input operands, there is a set of buses that receive the values from the registers. These buses will be connected to each functional unit, and a multiplexer is responsible for choosing the correct value (Figure 5a). As can be observed, there are two multiplexers that will make the selection of which operand will be issued to the functional unit. We call them input multiplexers. After the operation is completed, there is a multiplexer for each bus line that will choose which result will continue through that bus line. These are the output multiplexers (Figure 5b). As some of the values of the input context or old results generated by previous operations can be reused by other functional units, the first input of each output multiplexer always holds the previous result of the same bus line. Furthermore, note that in the example used in Figure 5, the first group supports up to two loads to be executed in parallel, while in the second group four simple logic/arithmetic operations are allowed.

3.5 Second BT Level

The second level of the binary translation hardware was extended from [5]. It starts working on the first instruction found after a branch execution, and stops the translation when it detects an unsupported instruction or another branch (when the limit for speculative execution is reached). If more than three instructions were found, a new entry in the cache (based on FIFO) is created and the data of a special buffer, used to keep the temporary translation, is saved. This translation relies on a set of tables, used to keep the information about the sequence of instructions that is being processed, e.g. the routing of the operands as well as the configuration of the functional units. Other intermediate tables are also needed; however, they are used only in the detection phase. This information is not saved in the TCache since it is not needed during the reconfiguration phase.

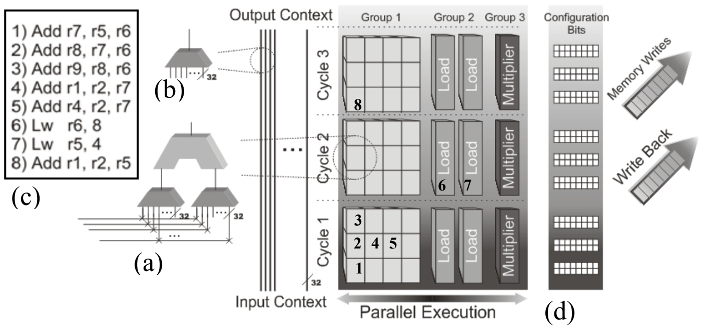


Fig. 5. The Reconfigurable Array

The BT algorithm takes advantage of the hierarchal structure of the reconfigurable array: for each incoming instruction, the first task is the verification of RAW (read after write) dependences. The source operands are compared to a bitmap of target registers of each row (which compose the dependence table). If the current line and all above do not have that target register equal to one of the source operands of the current instruction, it can be allocated in that line, in a column at the first available position from the left, depending on the group (using the resource table). When this instruction is allocated, the dependence table is updated in the correspondent line. Finally, the source/target operands from/to the context bus (input/output tables) are configured for that instruction. For each row there is also the information about what registers can be written back or saved to the memory (context table). Hence, it is possible to write results back in parallel to the execution of other operations. Figures 5c and 5d show an example of how a sequence of instructions would be allocated in array after detection and translation.

The algorithm supports functional units with different delays and functionalities. Moreover, it handles false data dependencies, and it also performs speculative execution. In this case, each operand that will be written back has an additional flag indicating its depth concerning speculation. When the branch relative to that basic block is taken, it triggers the writes of these correspondent operands. The speculative policy is based on bimodal branch predictor [6]. For each level of the tree of basic blocks, the counter must achieve the maximum or minimum value (indicating the way of the branch). When the counter reaches this value, the instructions corresponding to this basic block are added to that configuration of the array. The configuration is always indexed by the first PC address of the whole tree. If a miss speculation occurs a predefined number of times for a given sequence, achieving the opposite value of the respective counter, that entire configuration is flushed out and the process is repeated.

4 Results

4.1 Simulation Enviroment

To perform all of the tests we used a MIPS R3000 Processor with a unified instruction/data cache memory with 32 Kbytes. The reconfigurable array has 48 columns and 16 rows; each column has 8 ALUs, 6 LD/ST units and 2 multipliers. The Translation cache is capable of holding 512 configurations. In previous works [5], this setup has already shown to be the best tradeoff considering area overhead and performance boosts. The Mibench benchmark set [17] was executed on a Linux based OS. In all cases the applications were compiled and statically linked using GCC with -O3 optimization. X86 execution traces were generated by using the Simics instruction set simulator [18]. After that, cycle accurate simulators were used for the BT mechanisms, reconfigurable architecture and the MIPS processor. For the area evaluation, we used the Mentor Leonardo Spectrum [19] (TSMC 0.09u library) with VHDL versions of the MIPS [20], the reconfigurable architecture and the first BT level [5]. None of them increased the critical path of the MIPS processor, which runs at 600Mhz.

4.2 Binary Translation Data

In Figure 6, we analyze the memory occupation of the generated binary code. On average, considering the whole set of benchmarks, the MIPS compiler generated a binary code 26.69% bigger than the same code for the x86 processor. Furthermore, as can be observed in Table 1, the MIPS processor executes, on average, 36,63% more instructions than the x86, considering the execution of the same algorithm.

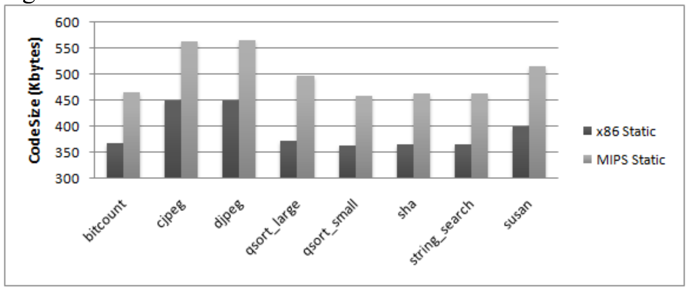


Fig. 6. Memory usage for different benchmarks, in Kbytes

Table 1. Number of executed instructions considering the two different ISAs

Benchmark	MIPS	x86
String Search	279,725	199,362
Sha	15,976,677	12,274,689
Bitcount	59,810,191	41,334,546
Qsort	51,695,224	27,386,935
Gsme	30,578,227	16,975,259
Gsmd	13,896,515	11,038,642

As explained before, the MIPS processor was modified to give additional support to the binary translation process. The Figure 7 shows the mean number of MIPS instructions generated from an x86 instruction when there is no support for the translation (Original Hardware), when there is support to EFlags computation only (EFlags Support) and when other hardware modifications, as explained in section III.C, are also included (Extended ISA). Figures 6 and 7, and Table I, show a clear difference in number of executed instructions between both architectures and how expensive the translation is. Both facts reflect in a performance overhead that the BT system must overcome.

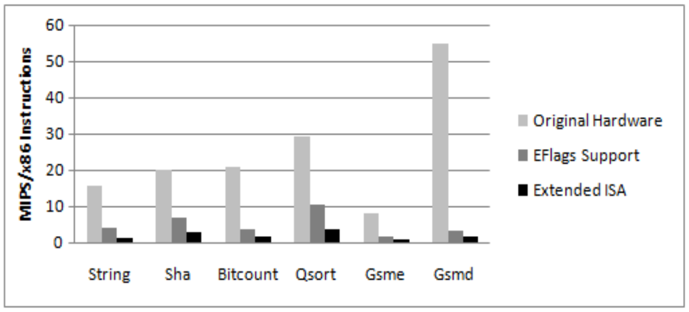


Fig. 7. The impact of using hardware support for the BT process.

4.3 Performance Evaluation

Figure 8 demonstrates the performance for four different setups:

- Native MIPS code execution on the standalone MIPS processor (MIPS Code Execution);
- Native MIPS code execution with reconfigurable acceleration. In this case, the first BT level is bypassed: only the second BT level plus the Reconfigurable Architecture (RA) are used (MIPS Code Execution +RA);
- x86 code execution without reconfigurable optimization, so only the first BT level is used (X86 Code Execution without RA);
- x86 execution using the two BT mechanisms and the reconfigurable array (X86 Code Execution – Two Levels BT).

The native code execution on the standalone MIPS processor was normalized to 100%. The MIPS Code Execution + RA presents a speedup of more than two times on average. For example, Sha presents a speedup of 3.43 times, Bitcount has gains of 2.42 times, whereas the GSM Encoder presents a speedup of 1.53 times, which is the worst case considering the benchmark set. Similar speedups are found when using other reconfigurable architectures [21][22]. Now, let us consider the x86 code being translated to MIPS code but not optimized by the reconfigurable system. As it should be expected, there are performance losses because of the translation mechanism. In the GSMD, a slowdown of more than 2 times is presented when compared to the native execution of the same algorithm in MIPS code. However, X86 code execution on the proposed system is faster than the native MIPS code execution on the standalone MIPS processor, hence amortizing the original BT costs. The speedup over the standalone MIPS execution varies between 1.11 and 1.96 times. On average, the performance gains are of 45%.

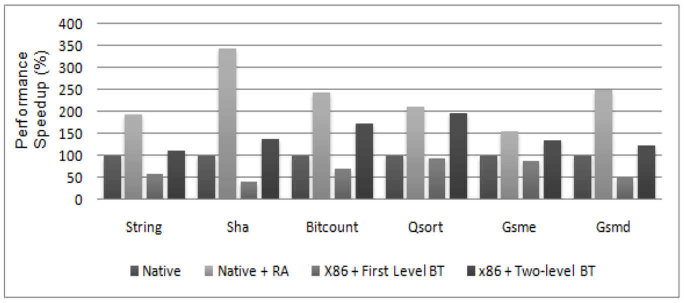


Fig. 8. Performance evaluation

These results can be considered very satisfactory: only the virtualization process (with no binary translation involved) of the Qemu virtual machine (VM) is 4 times slower than native execution of x86 instructions [22]. Because of such overhead, the Godson3, when translating code from X86 to MIPS using Qemu and without hardware support for the BT, is on average 6 times slower than native MIPS execution of the same software [12]. With hardware support for the BT mechanism, Godson3 performs on average 1.42 times slower than MIPS native execution. Although it is not our intent to directly compare our architecture to Godson3, since the Godson3 supports the whole X86 ISA, including interrupts and virtual memory, it gives one an idea that the translation is not an easy task to do, and that it presents significant overhead even if heavy hardware support is given.

4.4 Area Overhead

Table 2 demonstrates the number of gates that each hardware component takes. As can be observed, the fist BT level represents only 2% of the total system area. If we consider that each gate is composed of 4 transistors, the whole system would nearly take 4,87 million of transistors to be built. It is important to note that, if one compares the proposed system, which executes X86 instructions, to the standalone MIPS processor, there is a significant reduction in the instruction memory footprint, which amortizes the area overhead. According to our experiments, the MIPS compiler generated a binary code 26.69% bigger than the same code for the x86 processor, considering the whole set of benchmarks. Moreover, as already stated, the Godson-3 processor uses 4-superscalar MIPS R10000 cores. According to [24], each one of them takes nearly 2.4 million gates. Therefore, around 9.6 millions of transistors would be necessary to implement Godson, which is 2 times the size of our system.

Table 2. Area overhead of each component into the system.

Unit	**Area (Gates)**
First-Level BT	22,406
MIPS R3000	26,866
Second-Level BT	15,264
Rec. Array	1,017,620
Total	**1,219,535**

5 Conclusions

In this paper, we demonstrated the first step towards a totally flexible binary system, where both source and target architectures can be easily changed. In this case study, we proved the effectiveness of our technique by showing the possibility of executing the large amount of available x86 applications in a non x86 architecture in a totally transparent fashion, where no kind of user intervention is necessary and no performance losses are presented. We intend to improve our system, by increasing the number of supported X86 instructions and by adding support for different ISAs (e.g. ARM); and for different target architectures (VLIW, DSP and Superscalar processors). Moreover, we will also measure the power and energy consumption of the mechanism.

6 References

[1] Kim, N. S., Austin, T., Blaauw, D., Mudge, T., Flautner, K., Hu, J. S., Irwin, M. J., Kandemir, M., Narayanan, V.: Leakage current: Moore's law meets static power. Computer 36(12), 68–75 (2003)

[2] Mak, J., Mycroft, A.: Limits of parallelism using dynamic data dependence graphs. WODA, Chicago, Illinois, USA (2009)

[3] Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., and Robinson, S. G. 1993. Binary translation. Commun. ACM 36, 2, 69-81 (1993)

[4] Altman, E. R.; Kaeli, D.; Sheffer, Y.: "Welcome to the opportunities of binary translation," Computer , vol.33, no.3, pp.40-45, (2000)

[5] Beck, A. C., Rutzig, M. B., Gaydjiev, G., Carro, L. Transparent reconfigurable acceleration for heterogeneous embedded applications. In Proceedings of *DATE* . ACM, New York, NY, USA, 1208-1213 (2008)

[6] Smith, J. E. "A study of branch prediction strategies". In Proceedings of the 8th annual symposium on Computer Architecture, p.135-148, May 12-14, 1981

[7] Altman, E. R., Ebcioglu, K., Gschwind, M., Sathaye, S.: Advances and Future Challenges in Binary Translation and Optimization, In: Proceedings of the IEEE Special Issue on Microprocessor Architecture and Compiler Technology (2001)

[8] Rosetta, Apple Inc., http://www.apple.com/rosetta/

[9] Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, S. B., and Yates, J.: FX!32: A Profile-Directed Binary Translator. In: IEEE Micro, 56-64, (1998)

[10] Hookway, R. J., Herdeg, M. A.: DIGITAL FX!32: combining emulation and binary translation. In: Digital Tech. J. 9, 1, 3-12 (1997)

[11] Dehnert, J. C. et al. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, vol. 37. IEEE Computer Society, Washington, DC, 15-24 (2003)

[12] Hu, W., Wang, J., Gao, X., Chen, Y., Liu, Q., and Li, G.: Godson-3: A Scalable Multicore RISC Processor with x86 Emulation. In: IEEE Micro 29, 2, 17-29 (2009)

[13] Gschwind, M., Altman, E., Sathaye, P., Ledak, Appenzeller, D.: Dynamic and Transparent Binary Translation. In: IEEE Computer, vol. 3 n. 33, 54-59 (2000)

[14] Beck Filho, A. C. S.; Carro, L.: Dynamic Reconfiguration with Binary Translation: Breaking the ILP barrier with Software Compatibility. In: Design Automation Conference, DAC, 42, Anaheim. Proceedings… New York: ACM Press, p. 732-737 (2005)

[15] Lysecky, R., Stitt, G., Vahid, F., "Warp Processors". In ACM Transactions on Design Automation of Electronic Systems (TODAES), pp. 659-681, July 2006

[16] Or-Bach, Z. "Panel: (when) will FPGAs kill ASICs?". In 38th DAC, (2001)

[17] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. "MiBench: A free, commercially representative embedded benchmark suite." in Proceedings of the Workload Characterization. IEEE international Workshop. WWC. IEEE Computer Society, Washington, DC, 3-14 (2001)

[18] Magnusson, P. S., Christensson, M., Eskilson, et al: Simics: A Full System Simulation Platform, Computer, vol. 35, no. 2, pp. 50-58, (2002)

[19] Leonardo Spectrum, http://www.mentor.com

[20] Minimips VHDL, http://www.opencores.org

[21] Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., Taylor, R. R.: Piperench: A reconfigurable architecture and compiler, Computer vol. 33, n.4, pp. 70–77, (2000)

[22] Clark, N., Kudlur, M., Park, H., Mahlke, S., Flautner, K.: Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization, In: MICRO-37, pp. 30-40, (2004)

[23] Bellard, F. "QEMU, a Fast and Portable Dynamic Translator", USENIX 2005 Annual Technical Conference, FREENIX Track (2005)

[24] Yeager, K.C.: The Mips R10000 superscalar microprocessor, In: Micro, IEEE , vol.16, no.2, pp.28-41, (1996)

[25] Bala, V., Duesterwald, E. , Banerjia, S. "Dynamo: A Transparent Dynamic Optimization System". In PLDI'00, pp. 1–12, ACM Press, 2000.

[26] Daisy K. Ebcioglu, E. A.. "DAISY: Dynamic compilation for 100% architectural compatibility". IBM T. J. Watson Research Center - Technical Report, Yorktown Heights, NY, 1996.

[27] Compton K., Hauck, S., "Reconfigurable computing: A survey of systems and software". In ACM Computing Surveys, vol. 34, no. 2, pp. 171-210, June 2002.

[28] Hauck, S., Fry, T., Hosler, M., Kao, J., "The Chimaera reconfigurable functional unit". In Proc. IEEE Symp. FPGAs for Custom Computing Machines, Napa Valley, CA, pp. 87–96, 1997.

SESSION

INVITED SESSION - HOW TO EFFECTIVELY PROGRAM RECONFIGURABLE MULTI-CORE EMBEDDED SYSTEMS?

INVITED TALKS

Chair(s)

DR. PEDRO C. DINIZ

INVITED TALKS

How to Effectively Program Reconfigurable Multi-Core Embedded Systems?

Extended Abstract
for Special Session on EU-funded Research Projects

Pedro C. Diniz

Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento em Lisboa (INESC-ID)
Rua Alves Redol, 9
1000-029 Lisboa, Portugal
pedro.diniz@inesc-id.pt

The continued increase of the number of available transistors on a die has lead to the emergence of the many-core and multi-core computing architectures. These architectures promise the potential for orders of magnitude performance improvements over single core solutions through sheer concurrency. The abundance of transistors also enable the development of heterogeneous and (dynamically) reconfigurable architectures targeting selected markets such as real-time and/or embedded high-performance computing through a mix of traditional and customized cores.

These sophisticated computing architectures offer a wide variety of resources and for efficiency expose distinct execution models (e.g., threading and streaming) in addition to a wide range of hardware resources such as internal memories, custom configurable caches or dedicated functional units. This diversity has exacerbated the already complex application developing process as programmers must be aware, and explicitly manage, all the details of the target systems. The lack of powerful programming abstractions at various levels forces a plethora of tools to coexist leaving the programmer to bridge the gap between them at huge development costs. Developers cannot easily express high-level applications non-functional requirements (such as data rates or throughput requirements) in the de facto standard programming languages. They have to rely on resource management layers with application-specific management policies to best leverage the available resources in an extremely cumbersome and error-prone development process. As a result applications typically do not exploit the true potential of the target architectures.

This session presents a small sample of focused research projects funded by the European Commission involving both academic and industrial partners. We selected these projects as together they cover a wide spectrum of the approaches to various programming issues raised by these multi-core heterogeneous and possible reconfigurable architectures.

We begin by first presenting an heterogeneous reconfigurable System-On-Chip architecture: MORPHEUS [7]. This is a very dynamic reconfigurable architecture that exposes the raw programming issues at the hardware level such as the allocation and configuration of heterogeneous reconfigurable engines (HRE) at various levels of granularity, which are interconnected via multilayer AMBA buses. The execution model imposed in MORPHEUS is that of the Molen machine [8] where a traditional main processor (in this case an ARM core) manages and orchestrates the execution of an application code with its HREs.

While the MORPHEUS architecture offers a wide set of reconfigurable resources, the hArtes project (www.hartes.org) focused on the development of a compilation and synthesis tool chain targeting the Molen reconfigurable architecture with a single processor core and a configurable computing unit (CCU), typically implemented as an FPGA, acting as an accelerator. The hArtes tool chain [3] uses profiling information to identify the “hot” spots of an application code which then isolates and maps to the CCU for acceleration. It uses its own high-level compiler hardware back-end – the DWARV tool [9] – with which it generates the computation specification for the CCU in VHDL communicating with the host processor via a special set of registers.

Further up in the abstraction chain, the 2PARMA project (http://www.2parma.eu) tackles the programmability complexity by offering a set of flexible abstractions at the run-time level that offer adaptive task and data allocation as well as scheduling of the different tasks and the accesses to the data [2]. Their approach provides a layer to the compiler that can effectively leverage application knowledge to best exploit the heterogeneity of the target architecture. The 2PARMA project does address the need to capture those abstractions in the form of services and application meta-data with which an adaptive system can reason and provide different classes of applications (divided between critical and best-effort applications) with suitable Quality-of-Service (QoS) levels. The interface and the meta-data will ultimately allow developers to specify adaptive policies for the co-location and scheduling of computational tasks scheduling as well as dynamic memory management.

At a higher level of abstraction, the REFLECT project (www.reflect-project.eu) attacks the programmer productivity issues by combining aspect-oriented programming (AOP) techniques [5] to express non-functional applications requirement beyond the capabilities of current high-level programming languages. REFLECT Aspects [4] allow programmers to specify requirements such as data rates and data throughput, with which it steers compilation and synthesis tools to explore alternative software/hardware designs possibly with the use of pre-defined hardware design templates. In addition the REFLECT aspects also allow for the specification of compilation and synthesis strategies, including sequences of high-level program transformations, which are integrated with the CoSy [1] and Harmonic [6] high-level source-to-source compilers and the DWARV [9] VHDL synthesis tool.

While the mechanisms developed by the REFLECT do directly interact with the compilation and synthesis tool chains in the development of specific hardware designs, ultimately the execution of the application on the target architecture would benefit from the wealth of run-time services and reconfiguration infrastructure such as the one being developed in the context of the 2PARMA project.

Undoubtedly the problems raised by the emerging multi-core heterogeneous computing platforms are inherently very hard. Combined, these four projects propose specific solutions that address the key aspect of programmability of these configurable multi-core heterogeneous systems at the key levels of abstraction of the programmability challenge, namely, at the level of hardware, operating and run-time system and at the high-level programming language. We believe that only holistic approaches such as the ones targeted by the research projects presented in this session will have some chance of success. The continued funding in these areas of research and the growing interest from industry in tackling them does show how critical and widespread these issues have become. This session should therefore be of interest to any engineer and developer of current embedded solutions, as reconfigurable multi-core systems will undoubtedly take center stage in the upcoming decade as the de facto standard computing platform.

Keywords: Multi-Core Programming, Programming Languages and Execution Models, Aspect-Oriented Specifications, Reconfigurable Computing.

ACKNOWLEDGEMENT

This work was partially supported by the European Community's Framework Programme 7 (FP7) under contract No. 248976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the European Community. The authors are grateful to all team members of the REFLECT project for their help and support.

REFERENCES

[1] ACE CoSy Compiler Development System, http://www.ace.nl/compiler/cosy.html.

[2] A. Bartzas, P. Bellasi, I. Anagnostopoulos, C. Silvano, W. Fornaciari, D. Sourdris, D. Melpignano and C. Ykman-Couvreur, "Run-Time Resource Management Techniques for Many-Core Architectures: The 2PARMA Approach", in Proc. of the Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA'11), Invited Paper, July, 2011.

[3] K. Bertels, G. Kuzmanov and V.-M. Sima, "Heterogeneous Multicore Computing: Challenges and Opportunities – Experience from the hArtes Project", in Proc. of the Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA'11), Invited Paper, July, 2011.

[4] J. M. P. Cardoso, R. Nane, P. C. Diniz, Z. Petrov, K. Krátký, K. Bertels, M. Hübner, F. Gonçalves, J. Coutinho, G. Constantinides, B. Olivier, W. Luk, J. Becker, G. Kuzmanov, "A New Approach to Control and Guide the Mapping of Computations to FPGAs", in Proc. of the Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA'11), Invited Paper, July, 2011.

[5] G. Kiczales, "Aspect-Oriented Programming," in *ACM Computing Surveys (CSUR)*, special issue: position statements on strategic directions in computing research, 1996. 28(4es).

[6] W. Luk, J. Coutinho, T. Todman, Y. Lam, W. Osborne, K. Susanto, Q. Liu, and W. Wong, "A High-Level Compilation Toolchain for Heterogeneous Systems," in *Proc*. IEEE Int'l SOC Conf. (SOCC'09), pp. 9-18, Spet. 2009.

[7] F. Thoma, M. Kühnle, A. Grasset, P. Brelet, P. Millet, P. Bonnot. F. Campi, N. Voros, W. Putzke-Roeming, A. Schneider, M. Hübner, K. Müller-Glaser and J. Becker,"A Heterogenesou Reconfigurable System-on-Chip: MORPHEUS", in Proc. of the Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA'11), Invited Paper, July, 2011.

[8] S. Vassiliadis, S. Wong. G. Gaydadjiev. K. Bertels. G. Kuzmanov and E. Panainte, " The MOLEN Polymorphic Processor". IEEE Trans. on Computers, Vol. 53, n. 11, pp 1363-1375, 2004.

[9] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, J. Lu and S. Vassiliadis, "DWARV: Delft Workbench Automated Reconfigurable VHDL Generator", in *Proc. of the 17th Intl. Conf. on Field Programmable Logic and Applications (FPL07)*: 697-701, Aug. 2007

Heterogeneous Multicore Computing: Challenges And Opportunities

Experiences From The hArtes Project

Koen Bertels, Vlad-Mihai Sima and Georgi Kuzmanov
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
http://ce.et.tudelft.nl

Abstract— *This paper discusses the different challenges that were encountered during the hArtes project and how those challenges where met. A key objective of hArtes was to find the best mapping of an application on a particular heterogeneous hardware platform. The mapping process involves determining what parts of the application should be executed by what hardware component. When viewed in isolation, kernel based acceleration can produce significant speedups. However, when mapping the entire application, this potential never seems to live up to its full potential, due to other concerns than mere Amdahl's law. Many problems have to do with communication bottlenecks. As hArtes always used sequential C-code as the starting point, finding enough parallelism in those applications was also one of the other limitations limiting overall performance improvement. Nevertheless, the hArtes project succesfully adressed many of these problems resulting in a toolchain that assists the developer in mapping code on heterogenous multicore computing platofrms. The speedups obtained, measured for the whole applications, are between 1.94 and 31, compared to pure software execution.*

Keywords: heterogeneous, multicore, reconfigurable hardware, toolchain, embedded

1. Introduction

The hArtes project addresses the development of embedded systems. It investigates hardware/software integration and its main objective was to develop an integrated toolchain that provides (semi-)automatic support for the entire HW/SW co-design process. The applications used as input were written in various high-level algorithm descriptions, and, using the toolchain, a semi automatic âĂİbest fitâĂİ mapping to the platform was generated. The toolchain was intended to provide a fast development trajectory from application coding to the design of a reconfigurable embedded computing system. Having a (semi-)automatic process meant that a constant idea was to allow developers experimennt with different solutions.

How hArtes addresses the problems can be summarized as follows:

- Improved mapping and allocation algorithms were developed to be able to deal with non-uniform memory accesses and specific communications bottlenecks.
- OpenMP compliance allows to express and exploit parallelism while respecting the sequential consistency paradigm.
- Specific code optimizations such as loop unrolling or load balancing were developed to provide improved performance.

The flow is described, and the information flow is detailed. The hardware platform is presented next, by describing the high-level organization of the platform and the components used. The applications, used to validate the whole approach, are presented in Section 4.

The remainder of this paper is organized as follows. Section 2 presents the Molen programming paradigm and the Molen abstraction layer, which represents the foundations on which the hArtes toolchains are build. Section 3 presents all the tools involved in the hArtes tool-chain. Section 4 presents the application analysis.

In Section 5 a detailed description of the problems is presented. In the same section, we describe, for each problem, the solutions that were applied in the context of the project. We summarize the conclusions in Section 6.

2. Molen Programming Paradigm

The hArtes design approach assumes the Molen architecture and programming paradigm. The Molen programming paradigm targets machines that adhere to the Molen machine organization [11]. The Molen machine organization is based on the processor-coprocessor model and allows the processor to control the execution of the coprocessor by a set of fixed primitives. By using these primitives, any number of operations can be implemented as accelerated components. Started as an extension for reconfigurable architectures, this machine organization can be used for any heterogeneous architecture like the hArtes hardware platform. For each targeted architecture, different mechanism can be used to implement the Molen primitives; nonetheless, this will not reduce the generality of the approach. This is achieved by developing and implementing a hardware abstraction layer.

The Molen programming paradigm is based on the sequential consistency model that allows multiple processing elements to act as coprocessors to a General Purpose Processor (GPP) [12]. The paradigm defines five programming primitives that have to be implemented by a platform, together with their semantics. The five primitives are SET, EXECUTE, MOVET, MOVEF and BREAK.

The SET Primitive

The SET primitive function is to start the configuration of the processing element with the operation that has to be executed. This will represent different actions for different processing elements. For a Field Programmable Gate Array (FPGA), it invokes the partial reconfiguration. For a Digital Signal Processor (DSP) processor, it represents the loading of an executable file into memory. This operation can take a different amount of time depending of the characteristics of the processing elements and the specific operation. Having a separate primitive to manage the configuration allows the compiler to perform scheduling of reconfigurations or loading. An appropriate scheduling would be able to hide the configuration time, by making the configuration in parallel with other useful computations.

The MOVET, MOVEF Primitives

The role of the MOVET and MOVEF primitives' is to send (MOVET) and receive (MOVEF) parameters to/from the processing element.

The EXECUTE Primitive

From the programmer's perspective, the EXECUTE primitive will start the execution of a Custom Computing Unit (CCU). This operation can be done asynchronously. The semantic is that the operation will start on a processing element, while the execution continues on the GPP. It is the responsibility of the programmer to check for the status of the execution, using the BREAK primitive.

The BREAK Primitive

The role of the BREAK primitive is to synchronize the execution of the GPP with the execution of the FPGA. This primitive semantic is that the GPP will be stalled until the execution of the FPGA finishes.

In Section 5 we will discuss how these primitives were modified and extended in the context of the hArtes project.

Source Annotations

Molen programming paradigm can be used with any compilation flow that can partition an application. The partition is done between GPP and the other processing elements. The C language was chosen as it is one of the most used languages in the embedded system world. The structure of the C language was analyzed, and it was decided that functions

```
#pragma map call_hw VIRTEX4 1
int funcA(char *p, int len) {
  ...
  return v;
}

int funcB(char *p, int len) {
  ...
  return v;
}

void main() {
  ...
  #pragma map call_hw VIRTEX4 2
  result = funcA(variable, 100);
  ...
  result = funcB(variable, 100);
}
```

Fig. 1: Examples of Molen pragma on function definition and on function call.

offer the necessary abstraction level to model computations that run on processing elements. To specify which functions will be mapped to other processing elements, the standard language extension mechanism of C was used, namely, pragmas. Two types of pragmas were introduced as it can be seen in Figure 1:

- On a function declaration. The semantic is that all calls to that function will be translated to Molen primitives. The Molen backend compiler will provide one implementation for the corresponding processing element.
- On a function call. The semantic is that the call will be replaced by the corresponding Molen primitives. The Molen backend compiler will provide an implementation for the corresponding processing element. This allows more optimization opportunities, as different implementations can be generated for that function, for example, taking into account constant function parameters.

The information in the pragma includes:

- the name of the processing element to which the computation will be mapped.
- the implementation identifier. This makes a connection between implementations and theÂătoken (function declaration or function call) to which the pragma applies.

3. hArtes Toolchain And Hardware Platform

The hArtes toolchain main objective is to support the entire process of mapping an existing application onto a specific reconfigurable heterogeneous system. The application has to be described at a high algorithmic level or in a high level programming language. For the algorithm description, the developer has the choice of using GUI based tools or specific signal processing oriented languages as long as the final output is ANSI C.

The toolchain is composed of several toolboxes, which provide different functionalities and together realize the main objective. A schematic representation of the toolchain is depicted in Figure 2.

The high level design alternatives (a graphic entry, a signal processing or computation oriented language, or just general purpose C language) are offered by the hArtes AET (Algorithm Exploration and Translation) toolbox. For the graphical part, NU-Tech was adopted as Graphical Algorithm Exploration (GAE) solution and Scilab as a computation-oriented language. NU-Tech [7] is a platform that supports the development of algorithms for real-time scenarios, emphasizing strict timing control. Scilab is a free and open source software for numerical computations, similar to Matlab [5].

The output from the Algorithm Exploration Toolbox will be processed by hArmonic which will partition the application and decide one one specific mapping. At each step, the application developer has full control over the process.

The tools available in the hArtes Framework are performing the following tasks:

- Automated partitioning of the high-level algorithm descriptions. Using a set of predetermined design criteria and information about available resources, the high-level algorithms are divided into tasks.
- Transformation of the high-level algorithm tasks. This includes, for example, transformation to make tasks compatible with specific processing elements, like the FPGA.
- Design space exploration. Exploring the potential mapping possibilities between the tasks available and the processing elements of the reconfigurable heterogeneous systems. This is done by using estimated or measured costs for each task.
- Code manipulations. This includes, for example, scheduling of lengthy operations like the reconfiguration.
- Mapping and generation of the code for the targeted GPPs and DSPs.

The final step invokes the backend tools that will produce the different branches for the different hardware components. Those tools are:

- VHSIC Hardware Description Danguage (VHDL) code generation, for the target FPGA.
- Synthesis of the VHDL code obtained, using vendor-specific tools.
- Compilation of C code for the GPP processors present in the system.
- Integration of all binary code generated and running on the platform.

The *hArtes Hardware Platform (hHP)* was designed in such a way so that it represents a reference target for the hArtes toolchain, while also providing computing resources for several high-performance applications in the audio domain. We present, briefly, the main characteristics of this platform.

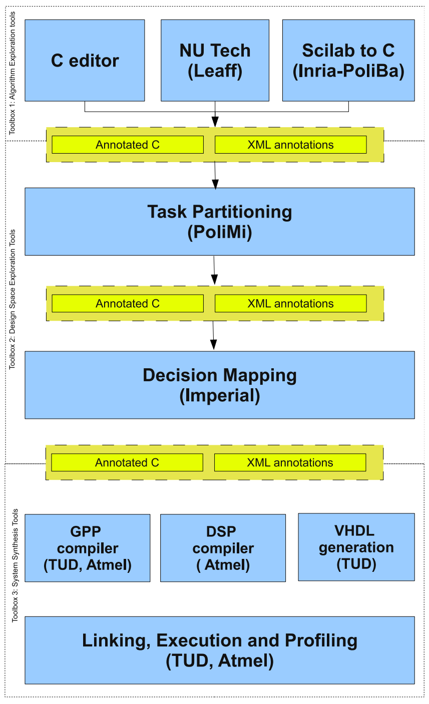

Toolbox 1: Algorithm Exploration tools
C editor
NU Tech (Leaff)
Scilab to C (Inria-PoliBa)
Annotated C
XML annotations
Toolbox 2: Design Space Exploration Tools
Task Partitioning (PoliMi)
Annotated C
XML annotations
Decision Mapping (Imperial)
Annotated C
XML annotations
Toolbox 3: System Synthesis Tools
GPP compiler (TUD, Atmel)
DSP compiler (Atmel)
VHDL generation (TUD)
Linking, Execution and Profiling (TUD, Atmel)

Fig. 2: hArtes toolchain overall architecture

Following the Molen Architectural template, the hHP consists of a GPP, several co-processing units. As many of the hArtes applications were audio oriented, special care was given to audio input and outputs from the platform.

The basic independent block of the system includes an ARM processor, a DSP processor (ATMEL Magic) and an application-specific reconfigurable block (Xilinx Virtex4 FPGA). This hardware block is called in hArtes terminology a Basic Configurable Element (BCE). In case these resources are insufficient for an application, an extension mechanism was designed, which would allow multiple BCEs to be connected.

For the actual hardware implementation, two BCEs were put on the hArtes board. The general organization of the hHP is shown in Figure 3. The current board contain two BCEs each, but multiple BCE could be chained if needed. The BCE are independent of one another, and can run any selected thread or application mapped by the hArtes toolchain.

For massive data streaming, dedicated hardware is configured on the board, represented by the "Audio I/O subsystem" block in Figure 4. Eight input and eight output Alesis Digital Audio Tape (ADAT) Lightpipe interfaces are available on the board. The ADAT Lightpipe is a standard for transfer of digital audio that uses fiber optic cables and has Toslink connectors at either end.

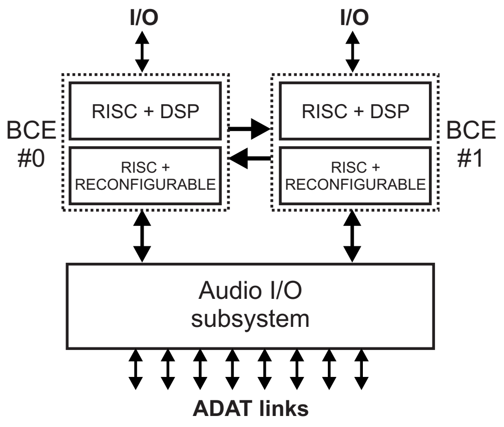


Fig. 3: Top level architectural structure of the hArtes Hardware Platform. The system has two independent heterogeneous and configurable processors that communicate among each other and with an audio I/O subsystems.

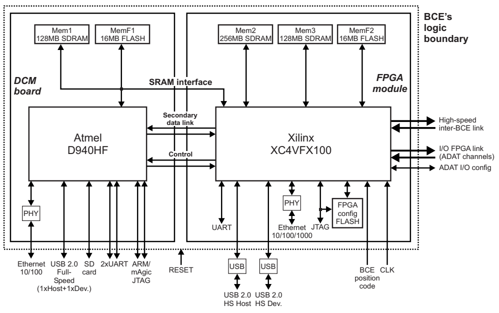


Fig. 4: Detailed block digram of the Basic Configurable Element (BCE) of the hArtes Hardware Platform. The BCE is the basic building block of the platform, supporting several processing architectures. One or more BCEs work in parallel to support an hArtes application.

Figure 4 provides a detailed overview of a single BCE. The two main blocks are a Reduced Instruction Set Computing (RISC)/DSP processor block (the D940HF produced by Atmel) and the reconfigurable processor (Xilinx Virtex4-FX100) [2]. The RISC processor is an ARM926EJ-S ARM processor, running at the 160MHz. It has a 16 kb instruction cache and a 16kb data cache. The DSP processor is a MagicV VLIW DSP running at 80Mhz. [1].

The memory is organized into private memory blocks and shared memory blocks. "Mem1" and "Mem3" in the figure are private to the D940HF to XC4VFX100, respectively. "Mem2" is a shareable memory bank.

4. hArtes Applications Analysis

The challenge with heterogenous/homogenous computing platforms is to efficiently map the application to the available resources. The hArtes project developed the following strategy on which the hArtes toolchain is based:

- Profile the selected applications on both the desktop computers and the real platform, if possible.
- Identify from the profile information, hot-spots, which take a significant portion of the application execution time.
- Select a function or only a fraction of a code, for mapping to another processing element.
- Profile the function on the processing element.

The function profiling on a different processing element is affected by the way in which memory is allocated. If the data does not reside in the processing's element local memory, then a transfer will be neeeded between main memory and the local memory. This transfer can represent a significant overhead that might negate the speedup obtained.

We focus in this paper on FPGA based mapping, but the framework also supports function mapping on the DSP.

In this section we describe the process for mapping the real world application provided by the individual partners of the project.

To give an idea of the complexity of the applications, the number of files and total lines of code are given in Table 1.

Table 1: Application metrics

Application	Number of files (headers)	Total number of lines (in headers)
x264	104(53)	38167(4565)
wfs	30(16)	2860(510)
incar	38(21)	3167(814)
tcf	76(18)	7216(583)

4.1 Video applications - H.264 codec

H.264 is the standard for video compression and decompression, developed jointly by ITU-T and ISO/IEC (with the name MPEG-4 AVC) [6]. To use this standard, the industrial partners in the project decided to use the free software library implementation, namely x264 [13].

4.1.1 Profiling

For this application, optimizations are written directly in assembly for various architectures. As we will see, this affects also the application structure and, because of that, an automated toolchain has to do more work to uncover the real structure.

We chose to run the encoder with the following default parameters: no B frames, subpixel motion estimation and partition decision quality of 5 on a scale from 1 (fast) to 7 (best), one reference frame, integer pixel motion estimation method hex. The profile information might differ when using other parameters. This analysis represents the basis for optimizing theÂă application.

In order to profile any application and obtain useful information for the hArtes toolchain at least one thing is needed, disable inline-ing of functions. This is necessary as the toolchain uses information about functions, so keeping the inlining optimization active will skew the result. Specifically for the x264 application another source transformation was needed, namely, transforming macro definitions to function calls. As this is a very specific optimization, we performed

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 30.38    110.16   110.16 619761780     0.00     0.00  pixel_satd_wxh
 26.65    206.78    96.62 693548340     0.00     0.00  pixel_sad_wxh
 10.69    245.55    38.77 360123870     0.00     0.00  get_ref
  7.19    271.63    26.08 290354340     0.00     0.00  motion_compensation_chroma
  4.55    288.11    16.48    406080     0.04     0.04  x264_frame_filter
  1.66    294.13     6.02 119345190     0.00     0.00  quant_4x4
  1.64    300.07     5.94  37245600     0.00     0.00  refine_subpel
  1.32    304.85     4.78  29886060     0.00     0.01  x264_me_search_ref
  1.23    309.32     4.47 121346040     0.00     0.00  sub4x4_dct
  0.95    312.75     3.43  34650000     0.00     0.00  ssim_4x4x2_core
```

Fig. 5: Profile on hHP board

it manually and didn't integrate it in the toolchain. The final profile information obtained after these modifications is show in Figure 5. We used input data videos from a repository for freely-redistributable test sequences [14].

4.1.2 Mapping

Using the profiling information, it was decided to map the two most CPU intensive functions to the FPGA. The first step was to evaluate the speedup obtained for eachÂăkernel. Not all kernels give a speedup when moved to another processing elements. So, even if, they are high in the profile list, that does not mean they should be mapped to the FPGA.

The steps performed to obtain the speedup obtained by mapping a kernel on a computation element, are:

- Create, using the processing compiler, in our case Delft Workbench Automated Reconfigurable VHDL Generator (DWARV), an FPGA implementation for both *pixel_sad_wxh* (referred from now, simply as SAD) and *pixel_satd_wxh* (referred from now, simply as SATD).
- Instrument the application to obtain all the possible data sets used by the kernels.
- Run the kernels using the data collected, using either local or shared memory.

The last step is necessary in order to determine the communication overhead.

After we perfomed the instrumentation, we observed that the kernel was called with some parameter combinations more often. This affected the execution time and the size of the data inputs. As an example, we show the results obtained for SATD kernel in Table 2. For example the case *s1=16, s2= 8 lx= 4, ly= 4* executes on average 21 times per video, which represent 22% of the total kernel invocations. The other column represents the sum of other six cases,

Running the SATD kernel for the cases identified in the application gives the results in Table 3. We performed two tests: one with the data available in the local memory of the FPGA and one with the data transfered first from the main memory. We can see that the FPGA is faster - up to 7.58 faster for one case - but there are 2 cases in which it is slower. This is directly related to the number of iterations performed: the worst performance is obtained for the smallest number of iterations (lx and ly represent iterations count) while the best is obtained for a large number of iterations (when lx and ly are 16).

When taking into account the transfer of memory performed between the ARM and the FPGA, the overall execution time is greater on the FPGA as can be seen from column 3. This is due to a extremely inefficient transfer speed between the main memory and the local FPGA memory.

4.1.3 Conclusion

From analyzing this application we can see that just taking the profilling information is not sufficient as code transformation might give a skewed view over which functions take most of the application execution time. More than that, bulk profiling information is not sufficient in all the cases. Rather, a way of detecting the correlation between parameters should be available in an instrumentation tool to help the designer

Table 2: Number of calls for each combination of parameters, and percentages from total number of invocations for SATD when running on more videos

Video	s1=16, s2= 8 lx= 4, ly= 4	s1=16, s2=16 lx= 8, ly= 8	s1=32, s2=16 lx= 4, ly= 4	s1=16, s2= 8 lx= 8, ly= 8	s1=16, s2=16 lx=16, ly=16	s1=32, s2=16 lx= 8, ly= 8	s1=16, s2= 8 lx= 8, ly= 4	s1=16, s2= 8 lx= 4, ly= 8	s1=16, s2=240 lx= 8, ly= 8	other
akiyo	18	13	3	19	12	5	4	4	4	12
carphone	22	12	17	8	4	4	6	6	4	11
claire	21	11	8	18	9	4	5	5	3	8
coastguard	16	13	26	5	4	4	4	4	4	11
container	18	14	23	10	8	6	2	2	4	6
foreman	22	13	14	7	4	4	7	7	4	13
hall	22	15	15	12	9	6	2	2	4	6
miss-amer	25	10	13	16	6	4	5	5	3	7
mobile	23	17	2	5	4	6	7	6	6	16
news	20	13	13	10	7	5	5	5	4	11
salesman	24	18	3	10	9	7	4	4	5	8
silent	24	14	12	8	5	5	6	6	4	9
suzie	21	10	20	9	4	4	6	6	3	10
Total	276	173	169	137	85	64	63	62	52	128
Average from total (%)	22	14	13	11	7	5	5	5	4	10

Table 3: Processing times and speedups in various scenarios for SATD unrolled

Case	Times μs			Speedup	
	FPGA	FPGA with transfer	ARM	FPGA vs ARM	FPGA w/t vs ARM
s1=16, s2= 8 lx= 4, ly= 4	9	134	7	0.79	0.05
s1=16, s2=16 lx= 8, ly= 8	9	141	28	2.92	0.20
s1=32, s2=16 lx= 4, ly= 4	8	137	7	0.90	0.05
s1=16, s2= 8 lx= 8, ly= 8	11	140	28	2.43	0.20
s1=16, s2=16 lx=16, ly=16	19	156	108	5.56	0.69
s1=32, s2=16 lx= 8, ly= 8	9	144	28	2.94	0.19
s1=16, s2= 8 lx= 8, ly= 4	8	135	14	1.68	0.10
s1=16, s2= 8 lx= 4, ly= 8	8	139	14	1.73	0.10
s1=16, s2=240 lx= 8, ly= 8	9	196	31	3.22	0.16
s1=32, s2=16 lx=16, ly=16	17	165	109	6.32	0.66
s1=16, s2=16 lx= 8, ly=16	14	152	55	3.94	0.36
s1=16, s2=16 lx=16, ly= 8	12	143	54	4.64	0.38
s1=16, s2=240 lx=16, ly= 8	12	197	60	5.12	0.31
s1=16, s2=240 lx=16, ly=16	17	272	130	7.58	0.48
s1=16, s2=240 lx= 8, ly=16	12	270	66	5.53	0.24

making the best decision when optimizing. We refer such a method in Section 5.

For this application, because of the memory transfer overhead, the FPGA is slower than the GPP. Optimizations that reduce this overhead are needed in order to be able to use the FPGA for a faster application execution. We present one such optimization in 5.1

4.2 Immersive audio - Beamforming and Wave-field Synthesis

In this section, we will discuss the implementation of a multi-beam, broadband beamforming and wave-field synthesis (WFS) [4]. These audio algorithms can be used in audio-visual transmission scenario like a telepresence application. A camera will tracks the recording directions (for example, by tracking human faces in the recorded scene) and using the beamformer algorithm the sound waves will be filtered based on its direction and sent through a transmission medium to the rendering side. There, the spatio-temporal properties of the recording space will be reproduced using a wave-field synthesis algorithm. Compared to other stereophonic approaches, the properties that are reproduced are not limited to a sweet spot, but to a much wider area, depending on the number of speakers array.

4.2.1 Profiling

This application is simpler than the x264 application and does not contain any assembly optimization. Most of the computations performed are floating point, which is a significant drawback for the simple ARM processor embedded in the hArtes platform, which does not have a FPU unit.

The application will be executed with the default parameters: the sampling frequency (48 khz), the number of sources (2), the room size (15m). By profilling the application, we observe that 80% of the time is spent in one function, *fFD_RealFIR_Pair_fpga* (referred from now simply as FFD).

4.2.2 Mapping

After using DWARV to generate the VHDL, synthesizing and implementing, we see that the area occupied by FFD kernel is 25% of the available area on the FPGA. This is due partly to the extensive use of floating point units (8 units, including addition, multiplication and division).

The FFD kernel will be called with a constant processsing window size of 1024. We tested more window sizes. The results are presented in Table 4. We can see that when the data is located in the local FPGA memory the speedup is between 5.99 and 6.43 and, if a data transfer is needed, the speedup is between 3.38 and 4.27.

4.2.3 Conclusion

Floating point operations are one of the most computational intensive operations of an application. Still, the floating point hardware units take a lot of area on the reconfigurable unit, so a careful trade-off has to be made in case of parallelism between area occupancy and multiple floating point units. The application speedup obtained is 1.94x compared to a pure software execution.

Table 4: Execution of units tests for FFD kernel on Virtex4-ML410

Size	Times μs			Speedup	
	FPGA	FPGA with transfer	ARM	FPGA vs ARM	FPGA w/t vs ARM
256	2047	3898	13169	6.43	3.38
512	4410	6965	26435	5.99	3.80
1024	9370	13671	58407	6.23	4.27

Table 5: Execution of units tests for FRACSHIFT kernel on Virtex4-ML410

Size	Times μs			Speedup	
	FPGA	FPGA with transfer	ARM	FPGA vs ARM	FPGA w/t vs ARM
original	11017	11263	19013	1.73	1.69
optimized	2389	2646	19176	8.03	7.25

4.3 In Car Audio - Enhanced Listening Experience

The focus of this application is enhancing the listening experience for travellers in a car. This is challanging due to the inherent dynamic nature of the environment with a lot of outside noise. The noises, spatial and spectral properties of the reproduced field, change the rendering of the sound from the loudspeakers. The system used as an example here is a real-time application with two objectives:

- to develop a complete set of audio algorithms for improving the audio quality, taking into account some features of the cabin.
- to have a modular system that can be adapted to other environments or that can use other algorithms

4.3.1 Profiling

Rather than profiling, the developer of the application suggested the kernel to be accelerated, namely FRACSHIFT.

4.3.2 Mapping

The kernel FRACSHIFT is represented by a delay line, followed by a floating point addition and accumulation.

As given in Table 5, we distinguish again betweeen the speedups obtained with and without taking memory transfers into account. When viewed in isolation, the FPGA is up to 1.73 times fastern then the ARM, This reduces to maximum 1.69 when including the memory transfers.

We also constructed an experiment to assess to what extent loop unroll and scalar replacement could provide additional speedups. To this purpose we manually implemented a design that fully unrolled two of the loops, partially unrolled the main computation loops and used a local array to store one of the parameters during the function execution. From

Table 5 we can see that the optimized version performs 4.25 times better than the unoptimized version, including the transfer time needed to transfer the data to and from the FPGA. The price for the gain in performance is an area increase of 7.3 times in slices.

When using the full capabilities of the platform (including the DSP processor and the two BCEs) the total speedup obtained is 31x compared to executing just on the GPPs of each BCE.

4.3.3 Conclusion

A similar conclusion as in the previous cases can be drawn. Taking the memory transfers into account limits the overall speedups. The use of specific optimizations such as loop unrolling can substantially improve the performance.

5. Identified Problems and Proposed Solutions

On the basis of the applications mapping experience we now discuss in a more generic way the different issues that were encountered and for which solutions had to be found. We distinguish below platform related issues, toolchain related issues and application related issue.

5.1 Platform related issues

Problem: The hardware platform developed for the hArtes project was designed for audio processing. The initial design decisions were taken considering that the FPGA will be used for the audio processing. For this reason, communications resources were allocated to the path between audio I/O and the FPGA, as the FPGA is the natural choice for processing a large number of audio streams. The connection between the FPGA and the ARM or the main memory was not considered to be important. The implication is that there is no Direct Memory Access (DMA) available when transferring between main memory and the FPGA local memory. Even if the absence of DMA is acceptable for reading or the writing of audio data, this limits the efficiency of the FPGA when dealing with generic applications or applications that have their input data stored somewhere else than in dedicated buffers.

This is not a core architecture problem, but rather a design decision, determined by the objective of the hardware board and its imagined uses. In order to evaluate the memory speed, we performed a series of experiments.

Since the DMA was not available to access the FPGA memory, all the transfers between the main memory and the FPGA local memory had to be performed by the GPP. Table 6 illustrates the results obtained when transferring a block of 32 kb between various memories. As we can see, the transfer between SDRAM and FPGA Scrach Pad Memory (SPM) is 2.85 times slower than transferring between Synchronous dynamic random access memory (SDRAM) and SDRAM.

Table 6: Transfer speed between SPM and SDRAM

Memory size (kb)	32
SDRAM to SPM (us)	1287
SPM to SDRAM (us)	1349
SDRAM to SDRAM (us)	451
Transfer speed SDRAM to SPM (Mb/s)	24.08
Transfer speed SDRAM to SDRAM (Mb/s)	69.23

Table 7: Mismatches between initial assumptions and current status of platform and applications.

	original	current
A	kernels are sharing code	kernels do not share code
B	memory is fully shared	only GPP sees memory as shared
C	break is synchronous	multiple applications have to execute

Experiments have shown that the bandwidth has the same value for both larger and smaller memory block sizes.

Solution: There are two solutions to solve this problem:

- adapt the mapping algorithm to map computations to the FPGA only if a speedup would be obtained. In order to be able to predict the speedup, the mapping algorithm needs as input the memory needed for each function. If possible the decision is taken at compile time, but, for the other cases a runtime solution was developed. The runtime solutions relies on runtime profiling information to assess which is the best mapping, and it is described in [9].
- reduce the transfer between SDRAM and the SPM. This can be accomplished by determining the type of the memory (write only, read only, read write), but also by improving the allocation algorithm, as some data might be accessed only from the FPGA. If this is the case, the data could reside directly in the SPM.

Problem: In order to support floating point operations, a floating point library of cores has to be available to the DWARV hardware compiler.

Solution: Xilinx provides floating point cores, but they can not be used directly by DWARV. The reason is that there are several differences between the Xilinx FP cores and the standard ones supported by C. One example of such difference is the rounding mode - truncation is used by C standard, rounding is used by Xilinx tools. VHDL wrappers were developed for each floating point operation, that made the Xilinx cores comply to the C standard.

5.2 Toolchain related issues

By studying the applications presented in Section 4, we identified mismatches between the assumptions made when Molen was designed and the existing applications. These mismatches are summarized in Table 7.

We give below more extensive description of the problems and solutions.

hArtes implementation of the Molen programming paradigm: While the Molen machine can be seen as an ideal machine organization, when building a real platform, choices have to be made resulting in a less than optimal physical implementation. Besides the functionality described in Section 2, other features are needed for a full fledged implementation, like profiling, automatic memory transfer, debug functionalities.

Solution: We extended the existing primitives and created the Molen Abstraction Layer (MAL). Its role is to abstract even further the details of the platform. Given the organization of the FPGA in the hHP the SET primitive will perform a full reconfiguration of the reconfigurable part.

As the processor is not tightly coupled with the FPGA (in fact, it is physically on another board) the EXECUTE and BREAK primitives are implemented using memory mapped control registers of the Molen controller mapped on the FPGA.

The debug and profiling facilities are offered at the level of the primitives. MOVET, MOVEF, EXECUTE and BREAK primitives have included facilities to profile the time spent in each of them. At the end of the program execution, all the information is dumped in a special profile result file.

For debugging purposes, MOVET and MOVEF dump the memory contents used by the kernel, before and after its invocation. This allows the developer to check the correct functioning of the processing element, by comparing execution resutls on the GPP with the results obtained from executing on the FPGA.

Memory transfer and allocation problem: One of the objectives of hArtes was to facilitate an automated mapping to different processing elements using the shared memory paradigm. In the hArtes implementation, the GPP has access to all the memory of each processing element, but the processing elements only have access to their local memory. In case the local memory of each processing element is not enough for all the data it needs to process, transfers have to be performed to and from the local memory.

Solution: As the hardware configuration was given, we will discuss only the software solutions developed to address this problem:

- the memory is allocated to the local memory processing element memory. This solution is described in [9].
- the runtime library will manage the transfer automatically, when certain data are needed by a specific processing element.

We will describe the extensions implemented for the second solution. Even if it is a more straightforward, unoptimized approach, that does not perform any significant analysis or optimization, it is a good starting point that enables an application to be executed on our platform. Also, based on this implementation various optimizations can be performed in later stages.

Two new primitives are introduced, to differentiate the MOVET or MOVEF primitives for the special case of a pointer:

- molen_MOVETXaddr - used for each pointer parameter sent to a processing element, instead of the normal molen_MOVETX
- molen_MOVEFXaddr - used for each pointer parameter sent to a processing element, after the kernel finished the execution

These two primitives work together with the support of the runtime system. All the dynamic memory allocation will be made by special 'wrapper' functions that keep track of allocations. The replacement of the 'malloc' functions by the wrapper function 'hmalloc' (and similar functions, i.e. 'realloc' and 'calloc') is made by the source to source transformation tool. Then, at a later point in the execution of the program, for each address that is sent as a parameter to the CCU, the runtime system can determine the size of the block to which that address belongs.

Naturally this approach has several limits:

- except in parameters, addresses should not be present in the memory blocks used by the kernels. This means, for example, structures like linked list will not be supported. Although this is a limitation, none of the kernels analysed used such complex data structures.
- without further analysis or information from the developer the transfer can be inefficient. For example, some memory blocks are only written by the kernel while others are only read. The C language provides some information about this (example: const keyword), but it is incomplete (ex: there is no way to specify write only locations).

Molen parallelism problem: When targeting a multicore heterogenous system, one crucial aspect that has to be taken into account is expressing and using parallelism. For a shared memory system, one solution is to use OpenMP. The Molen programming paradigm is complementary to OpenMP in the sense that OpenMP annotations can be used by the compiler to generate the appropriate Molen primitives.

Solution: There are two solutions related to the use of parallelism:

- use Molen parallelism instead of the thread parallelism present in OpenMP, where possible. The advantage of this approach is that the overhead of thread management is eliminated (Molen overhead is much less than thread creation/switching overhead)
- use Molen primitives inside each thread. As Molen was developed for a single threaded environment, modification of the Molen primitive implementations are needed, in order for them to support the thread concept.

Generating Molen from OpenMP

This transformation can be done in almost all cases in which the code that has to be executed in parallel, also has to run on another processing element as in Figure 6. For this case, the compiler will understand the parallelism structure, and instead creating and synchronizing threads it will generate and schedule Molen primitives. The results are presented in Figure 7. Similar examples can be constructed for other OpenMP constructs.

```
#pragma omp sections nowait
{
  #pragma omp section
  {
    #pragma map call_hw VIRTEX4 1
    fft(p, n);
  }
  #pragma omp section
  {
    #pragma map call_hw VIRTEX4 4
    value = sad(d, l);
  }
}
```

Fig. 6: Molen pragma in the context of OpenMP sections

```
SET(1);
SET(4)
MOVTX_ADDR(1, p, n);
MOVTX(1, n);
EXECUTE(1);
MOVTX_ADDR(4, p, n);
MOVTX(4, n);
EXECUTE(4);
BREAK();
```

Fig. 7: Molen primitives generated for the OpenMP sections example

Using Molen Primitives in OpenMP Threads

When dealing with parallelism, for some cases, the scheduling of Molen primitives described earlier is not possible. One such case is shown in 8. Another option is to use the Molen primitives in the OpenMP generated code without the compiler understanding the semantic of the OpenMP pragmas. Then another problem arises: multiple Molen primitives will be invoked with the same identifier. To solve this, we proposed the following extension of the Molen primitives:

- use as identifier not only the identifier provided in the code, but the tuple (thread-identifier, identifier).
- provide an internal mechanism that the same CCU can be instantiated multiple times (for example, using different positions on the FPGA.

With these two additions, no other modification to the compiler or OpenMP runtime are needed, and Molen can be used like in Figure 8.

Development and toolchain debug problem: Building a hardware system involves changing a lot of parameters, sometimes it can happen that two components are used, one adapted for one specific value of a parameter of the platform and the other component for another value. This can result in an incompatibility and more specifically in a system failure at runtime. This means a function will produce

```
#pragma omp for schedule(dynamic,chunk)
for (i=0; i<N; i++)
{
  prepare(p[i]);
  #pragma map call_hw VIRTEX4 1
  fft(p[i],1024);
}
```

Fig. 8: Molen pragma in the context of OpenMP threads

incorrect results without any warning or error present during the compilation process.

Examples of such parameters are the CCU-frequency which affects the number of cycles needed for various operations, the endianess and the number of cycles to read from memory.

Solution: As the development went ahead, it was obvious a mechanism had to be provided to make sure the CCUs were generated for the system in which they were integrated. The identified parameters are:

- endianess
- the number of cycles needed to read/write the memory and eXchange Registers (XREG) memory area.
- the number of pipeline stages of each FP element.
- the frequency at which a specific CCU could run.

The solution depends on the moment at which the check can be performed. For the first three parameters, the check is performed at compile-time. The obvious choice was to add dummy signals to the VHDL code of the CCU. The names of these signals, encode the parameters for which the kernel was generated. In case the CCU was generated with a different configuration, the VHDL compiler would not have been able to match the signals and would report an error.

For example, a dummy signal could be named "check_fp_sp_mult_top_6". This means the CCU uses a floating point (fp) single precision (sp) multiplication unit (mult) that has a pipeline of 6 cycles. For each parameter that needs to be tested a corresponding signal is added.

In order to ensure the same frequency is used, a dynamic solution was chosen. As an FPGA has a complex Digital Clock Module (DCM) that can generate different frequencies, the compiler will set a special configuration register with the correct value, before running the CCU.

Synthesis times and maximum frequencies problem: For FPGA development, the time needed for synthesis can become a bottleneck in the development process. This happens because the synthesis time for a single VHDL kernel varies between 10 minutes to almost 2 hours, as we can see from Table 8. As the compilation flow involves a source to source transformation tool, each synthesis will regenerate all the files that are mapped to different processing elements. Traditional build system will detect a change in the timestamps of the VHDLs and restart the synthesis process.

Solution: The build system was adapted to check not only the timestamp but also the contents of the files before restarting the synthesis process.

Table 8: Compilation times using FPGA flow, at different frequency constraints

Kernel	DWARV time (s)	Xilinx time(s)	Frequency
sad		678	200
SATD		876	166
SATD unrolled		1529	166
FFD	10	7038	125
FRACSHIFT	10	900	125

Toolchain retargetability problem: Developing a toolchain for a heterogeneous architecture is very time and resource consuming task. To provide the highest degree of retargetability, the design of the framework must be done with care, to allow easy integration with 3rd party toolchains and changes in the architecture supported.

Solution: There are several aspects that have to be taken into account:

- use an open, generic standard for communication between tools. In the context of the hArtes project, these standards are Ansi C and XML annotations.
- keep the annotations (either in C and XML) independent from the platform. A good example for such design is the OpenMP standard, which abstracts away most of the details (like number of threads) from the users.
- keep platform dependent logic in libraries as oposed to hardcoding it in tools.
- provide an open build system. As specific compilers have to be used to build the final system, each with its own specificities, developing an open build system, will make it easier for the developer to control the whole process.

5.3 Applications related issues

Problem I: One problem is that the speedup was not always achieved for the application, given different input sets. This is illustrated in Table 3.

Solution: adapt the mapping, at runtime, based on the value of the parameters and on the speedup. A description of this solution is given in [8].

Problem II: Determine the optimal balance between several parallel kernels on a reconfigurable device. For the wavefield synthesis application, finding a balance between the number of kernel instances that compute the output waves and the number of kernels that compute the coefficients when needed, does not have an obvious solution.

Solution: at compile time Integer Linear Programming (ILP) can be used to determine the best balance. An efficient solution is given in [10].

Problem III Determine the best parameters for specific optimizations, to take advantage of the reconfigurable device. The two cases for this are the loop unroll number and the number of variables that are made local. Although a known optimization, in the context of the reconfigurable device, this has different implications, like reducing the frequency by making routing harder.

Solution Work has been done to identify the optimal unroll factor, taking into account profiling information, memory transfers and area utilization [3].

6. Conclusion

In this paper, we presented the work done in the context of hArtes project. We analysed the applications, and we highlighted the main challenges that had to be addressed before obtaining an efficient implementation on a Molen system. Several comments about the development flow were made, which, provide insights into what a complex toolchain for heterogeneous platforms has to provide. The hArtes tool chain is now being completely redesigned for later commercial release by a spin off that was created at the end of the project.

Acknowledgment

The authors would like to thank all the partners from the hArtes consortium. This research has been funded by the hArtes project (EU-IST-035143).

References

[1] ATMEL. At572d940hf preliminary summary.
[2] I. Colacicco, G. Marchiori, and R. Tripiccione. The hardware application platform of the hartes project. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 439 –442, 2008.
[3] O.S. Dragomir, T. P. Stefanov, and K.L.M. Bertels. Optimal loop unrolling and shifting for reconfigurable architectures. *ACM Transactions on Reconfigurable Technology and Systems*, pages 1–24, September 2009.
[4] Christoph; Hahn Volker; Leitner Michael Heinrich, Gregor; Jung. A platform for audiovisual telepresence using model- and data-based wave-field synthesis. In *Audio Engineering Society Convention 125*, 10 2008.
[5] INRIA. http://www.scilab.org/. 2010.
[6] ITU. http://www.itu.int/rec/t-rec-h.264. 2010.
[7] Leaff. http://www.nu-tech-dsp.com. 2010.
[8] V.M. Sima and K.L.M. Bertels. Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform. In *Proceedings of International conference on 16th IEEE Reconfigurable Architectures Workshop*, page 6, May 2009.
[9] V.M. Sima and K.L.M. Bertels. Runtime memory allocation in a heterogeneous reconfigurable platform. In *IEEE International Conference on ReConFigurable Computing and FPGA*, December 2009.
[10] V.M. Sima, E. Moscu Panainte, and K.L.M. Bertels. Resource allocation algorithm and openmp extensions for parallel execution on a heterogeneous reconfigurable platform. In *Proceedings of 2008 International Conference on Field Programmable Logic and Applications (FPL)*, September 2008.
[11] Stamatis Vassiliadis, Stephan Wong, and Sorin Cotofana. The molen rm-coded processor. In *In Proc. 11th Intl. Conference FPL-2001*, pages 275–285. Springer-Verlag LNCS, 2001.
[12] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The molen polymorphic processor. *IEEE Trans. Comput.*, 53(11):1363–1375, 2004.
[13] VideoLAN. http://www.videolan.org/developers/x264.html. 2011.
[14] xiph.org. http://media.xiph.org/video/derf/.

Runtime Resource Management Techniques for Many-core Architectures: The 2PARMA Approach

**Alexandros Bartzas[1], Patrick Bellasi[2], Iraklis Anagnostopoulos[1], Cristina Silvano[2], William Fornaciari[2]
Dimitrios Soudris[1], Diego Melpignano[3], Chantal Ykman-Couvreur[4]**

**[1]Institute of Communications and Computer Systems, Athens, Greece
[2]Politecnico di Milano, Italy
[3]STMicroelectronics, France
[4]IMEC, Interuniversity Micro-electronics Center, Leuven, Belgium**

Abstract—*Real-time applications, hard or soft, are raising the challenge of unpredictability. This is an extremely difficult problem in the context of modern, dynamic, multiprocessor platforms which, while providing potentially high performance, make the task of timing prediction extremely difficult. Also, with the growing software content in embedded systems and the diffusion of highly programmable and re-configurable platforms, software is given an unprecedented degree of control on resource utilization. Existing approaches that are looking into Runtime Resource Management (RTRM) still require big design-time efforts, where profiling information is gathered and analyzed in order to construct a runtime scheduler that can be lightweight. There is a trade-off to be made between design-time and runtime efforts. In this paper we present a framework for RTRM on many-core architectures. This RTRM will offer an optimal resource partitioning, an adaptive dynamic data management and an adaptive runtime scheduling of the different application tasks and of the accesses to the data. Furthermore, the 2PARMA RTRM takes into account: i) the requirements/specifications of many-core architectures, applications and design techniques; ii) OS support for resource management and iii) a design space exploration phase.*

Keywords: Runtime resource management, task, memory and power management

1. Introduction

The current trend in computing architectures is to replace complex superscalar architectures with many processing units connected by an on-chip network. This trend is mostly dictated by inherent silicon technology frontiers, which are getting as closer as the process densities levels increase. The number of cores to be integrated in a single chip is expected to continue to rapidly increase in the coming years, moving from multi-core to many-core architectures. This trend will require a global rethinking of software and hardware approaches.

Multi-core architectures are nowadays prevalent in general purpose computing and in high performance computing. In addition to dual- and quad-core general purpose processors, more scalable multi-core architectures are widely adopted for high-end graphics and media processing. Such platforms are becoming widespread as silicon technology develops in the sub-50nm nodes. The transition to multi-core is almost a forced choice to escape the silicon efficiency crisis caused by the looming power wall, the application complexity increase and the design complexity gap under tightening time-to-market constraints. While multi-core architectures are common in general-purpose and domain-specific computing, there is no one-size-fits-all solution. General-purpose multi-cores are still designed to deliver outstanding single-thread performance under very general conditions in terms of workload mix, memory footprint, runtime environment and legacy code compatibility. These requirements lead to architectures featuring a few complex, high-clock speed "mega-cores" with complex instruction sets, deep pipelines, non-blocking multi-level caches with hardware-supported coherency and advanced virtualization support. Today, we see a trend towards many-core fabrics, with a throughput-oriented memory hierarchy featuring software-controlled local memories, FIFOs and specialized DMA engines. As a result, an SoC platform today is a highly heterogeneous system. It integrates a general-purpose multi-core CPU, and a number of domain-specific many-core subsystems. Examples of such emerging multi-core platforms are the Intel's SCC [1] and ST's Platform 2012 [2].

System-level design and optimization of computing systems is a highly challenging task. Especially since such systems are becoming more and more complex, from both hardware as well as software perspectives [3]. Over the last few years, the main focus in the design of computing systems has been to provide good performance and at the same time achieve low-power consumption. To achieve optimal

This work is partially supported by the E.C. funded FP7-248716 2PARMA Project, www.2parma.eu

results, a good coordination between hardware and software design is required. Therefore, memory-intensive applications running on embedded platforms (e.g., multimedia) must be closely linked to the underlying Operating System (OS) and efficiently utilize the available hardware resources. Putting all this together, it is clear that developing a complete, working system is an integration nightmare [3].

In this paper we present a framework for Runtime Resource Management (RTRM) on many-core architectures. This RTRM will offer i) an optimal resource partitioning among the different resource requirements of the applications running on the hardware platform; ii) adaptive dynamic data management (dynamic allocation and de-allocation of heap data); iii) adaptive runtime scheduling of the different application tasks and the of accesses to the data. Furthermore, the adequate power management techniques as well as the integration to the Linux Operating System (OS) are currently developed. The 2PARMA RTRM takes into account: i) the requirements/specifications of many-core architectures, applications and design techniques; ii) OS support for resource management and iii) a design space exploration phase.

The rest of the paper is organized as follows. Background information regarding RTRM is provided in Section 2, whereas an overview of the proposed RTRM framework in Section 3. The runtime resource management component is presented in Section 4, the adaptive task management component is presented in Section 5 and the adaptive dynamic memory management component is presented in Section 6. Finally, conclusions are drawn and future work outlined in Section 7.

2. Background

Schaumont et al. [4] demonstrate the use of hierarchical configuration on an image processing example application. The authors' methodology starts by profiling a set of applications from a target application domain to determine the right point in the configuration design space. In this way, a set of commonly used and computationally intensive kernels can be identified. Parameterizable implementations of these kernels form the building blocks of the reconfigurable platform. These blocks are then pre-instantiated into the FPGA fabric and can be programmed at runtime with a minimal amount of configuration input. The authors suggest to use compiler techniques to map an application onto a given set of parameterizable IP blocks.

Keller et al. [5] describe the use of so-called software decelerators. By using freely available soft IP cores, the designer can take advantage of an easier software application design process. In addition, certain algorithms, like e.g., a sequential state machine use less hardware resources when implemented on a soft IP core, while still meeting the necessary performance requirements. The authors describe a configuration hierarchy case study.

Faruque et al. [6] present a runtime application mapping in a distributed manner using agents targeting for adaptive NoC-based heterogeneous multi-processor systems. Authors claim that a centralized RTRM may bear a series of problems such as single point of failure and large volume of monitoring-traffic. However, Nollet et al. [7] present a centralized runtime resource management scheme that is able to efficiently manage a NoC containing fine grain reconfigurable hardware tiles and two task migration algorithms. The resource management heuristic consists of a basic algorithm completed with reconfigurable add-ons. The basic heuristic contains ideas from multiple resource management approaches.

Shirazi et al. [8] present a method for managing reconfigurable designs, which supports runtime configuration transformation. This method involves structuring the reconfiguration manager into three components: a monitor, a loader, and a configuration store. To improve reconfiguration speed, authors have developed a scheme to implement the loader in hardware. The main motivation for implementing part of the runtime manager in hardware or a separate accelerator is to avoid the overhead caused by the runtime management functionality [9] [10], and to increase determinism. As presented in [11] when moving RTRM functionality to a platform hardware service, the runtime overhead is kept to a minimum as decision making is done in parallel by specialized hardware. In addition, hardware can operate at a finer granularity without incurring an performance penalty.

3. Runtime resource management: An overview

The development of new computing systems requires tuning of the software applications to specific hardware blocks and platforms as well as to the relevant input data instances. The behaviour of these applications heavily relies on the nature of the input data samples, thus making them strongly data-dependent. For this reason, it is necessary to extensively profile them with representative samples of the actual input data. An important aspect of this profiling is done not only at the dynamic data type level, which actually steers the designersâĂŹ choice of implementation of these data types, but also at the functional level. We characterize the software metadata that these optimizations require, and we present a methodology, as well as appropriate techniques, to obtain this information from the original application. Equally important is for the designer to have a good knowledge of the platform characteristics. With both this information at hand (software metadata and platform characteristics) the designer can characterize the runtime behaviour of the application and determine its working modes and the reconfiguration overheads.

This paper is focused on the design of a RTRM framework, targeting the optimization of both computing fabric resources usage and applications' Quality-of-Service (QoS) requirements. Specifically regarding Power Management, we investigate on the OS, services supporting runtime management. The main available OS frameworks related to power, hot-spots and process variation management have been analyzed and their characteristics have been compared to define the base for the future design of a new framework for supporting the QoS-based runtime management of a generic many-core computing platform. To support the new power manager, both application behaviour and computing platform characteristics should be identified, described and properly reported to the runtime manager framework. This involved a deeper investigation on two main aspects: the platform description and the interfaces with applications.

On one side, the platform description requires the identification and definition of a proper set of hardware metadata to represent a generic computation fabric. We identified a suitable formalism to support both portability and efficiency of the runtime controlling solution. On the other side, the need to interface and interact with applications is motivated by collecting application requirements, in terms of resources and expected behavior, and notifing control decisions according to a running optimization strategy.

All together, these aspects define the basic building blocks for a portable runtime resource manager, being both independent from any specific computation fabric and to control different applications. Based on these results we started the design of a runtime resource management framework at the OS level to support many-core computing platforms. In the meanwhile, we also investigated a runtime optimization policy, which could be used to efficiently allocate resources to applications according to multiple objectives.

3.1 The role of the Runtime Resource Manager (RTRM)

The RTRM framework is composed of a set of modules providing services to different "classes of users" (depicted in Figure 1). Two of these users are represented by applications and target-platforms. Applications usually have different Working Modes (WM), each one defining expected Quality-of-Service (QoS) and corresponding needs in terms of computational resources (e.g. processing elements, memory, bandwidth, etc). Target-platforms define a set of available resources, each one with specific: features, operating modes, monitoring and control points. Moreover, the platform architecture could define a specific functional relationship between available resources (e.g. clusters of processing elements and a certain memory hierarchy).

The RTRM is a component placed in between applications and the target-platform, which is in charge of managing applications access to platform available resources. This management is a quite complex activity generally aimed at meeting contrasting goals: maximizing applications' performance while reducing energy consumption. How this double goal could be obtained is behind the scope of this paper. Here it is important to understand that the RTRM tool should be supported on its role by the applications and the target-platform. Indeed, it is required for both these "users" to provide the framework with some information that could be effectively exploited to accomplish its task.

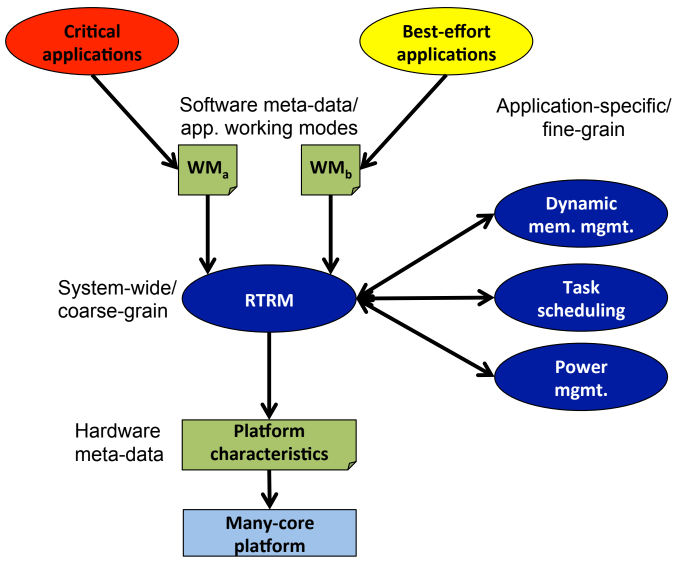


Fig. 1: Overview of the 2PARMA RunTime Resource Management Framework.

The overall structure of the required information, coming from applications and target-platforms (meta-data) should satisfy three main design goals:

1) **Completeness**: all the information required to properly support the RTRM management activities should be considered and represented.
2) **Portability**: the meta-data should describe both applications and target platform properties independently from each other. Indeed, for the success of the final solution it is considered interesting to support a "write once and run everywhere" approach. Thus, it should be possible to define application meta-data independently of the target platform they will run on.
3) **Simplicity**: the information required by the RTRM represents an "overhead" for developers of both applications and platforms, thus it is important to identify a solution that is as much as possible effortless to be used. Even better if the solution could be defined as an extension of the classical design and development flow of each system abstraction layer.

Overall, these requirements must be considered to properly define the collection of meta-data and the interfaces to acquire them from the applications and the target platform.

4. The RTRM Component

The RTRM component is composed of a *set of modules running at different levels of abstraction* and providing services to different "client" modules. The client modules represent the users of the services provided by this tool. Three main classes of users could be easily identified:

1) **User applications** which can be either critical or best-effort. The former are applications, generally known at design time and provided by the device producer/integrator, which implement fundamental tasks for the target device. The latter instead are applications unknown at design time but that each user could add and use once the system is already in production. Critical applications are usually fine tuned off-line to be highly efficient, e.g., by Design Space Exploration (DSE) techniques, and their resource requirement are considered mandatory for a proper functioning of the device. To the contrary, best-effort applications are expected to negotiate resource usage with the RTRM before actually accessing them.
2) **Resource tuning tools** for the optimization of specific resources. Within this class, three main users could be classified: code-optimizers (which are not further discussed in this paper), the runtime task manager (Section 5) and the runtime dynamic data manager (Section 6). The RTRM will provide support to these tools to better achieve their optimization goals while still granting to meet the system-wide optimization policy.
3) **Platform-specific controller** representing the lower-level interface to the many-core computing fabric services. This is usually the computing fabric device-driver, being it properly extended to provide the required runtime management specific services.

To provide a flexible and efficient implementation, a *hierarchical design* for the RTRM should be considered, where the monitoring, management, control and optimization strategies are operated at different granularity and abstraction levels. At least, three different granularities could be considered: user-space, kernel-space and fabric/device-space. The first two of them correspond to the host-side while the latter is the computation fabric side. An overall view of the tool architecture and main components is reported in Fig. 1.

The hierarchical architecture of the optimization policy allows to reduce the control complexity by distributing it at different levels, each one considering different details. Each granularity level collects requirements from higher level, runs a specific optimization policy, and finally identifies a set of constraints delivered to lower levels. This approach ensures control over sensible overheads at each level of the hierarchy. Moreover, it allows to keep time-critical and fine-tuning decisions running on lower levels, close to each controlled resource, while a global optimization strategy runs on higher levels. This approach will grant a prompt and low-latency handling of critical events while it still ensure a system-wide optimization.

An overall view of the requirements for the development of an effective RTRM tool is represented in Fig. 2 where it is proposed a target-based view. Indeed, we could identify design requirements, user interaction requirements, functional requirements and finally system-integration requirements. Some of these requirements are imposed by the RTRM tool to other system components while others are related to the tool itself.

4.1 Imposed requirements

The set of *imposed requirements* must be satisfied by the users of the tool, in order to be properly integrated with the RTRM. Each of these requirements is addressed either by the resources (controllers), i.e., platform devices and relative drivers, or by the applications[1].

a) Definition of resources working-modes [resources]: The RTRM tool needs a complete view on the working modes (WMs) of controlled resources. Each working mode is defined by a set of properties such as: the amount of available resources, the power consumption and the constraints to switch from one mode to another. These information on resources working modes will be collected by the RTRM tool by using a lower abstraction level module, presumably as an extension of the computing fabric driver. However, any component in a use-case, which represents a resource (i.e. platform subsystems), must completely define its working modes and notify them to the RTRM.

b) Definition of resources control points [resources]: In order for the RTRM tool to perform optimizations on resources usage, this tool requires the ability to tune some parameters of the corresponding platform subsystems. Thus, these subsystems are required to expose their control points to the RTRM and to define how the modification of these control points impact on the subsystem behaviors in terms of both power consumptions and performances. Usually, the optimization actions performed by the RTRM, using these control points, correspond on switching a subsystem from one of its working-mode to another.

c) Resources observability [resources]: To properly run its optimization policy, the RTRM will relay on an updated and complete view of the resources state, both in terms of power consumption and performance. Thus, every subsystem representing a resource to be controlled by this tool is expected to expose some observability points. These points will be represented by some metrics that can be mapped on

[1]For improved readability, the target of each requirement is indicated right after the requirement name

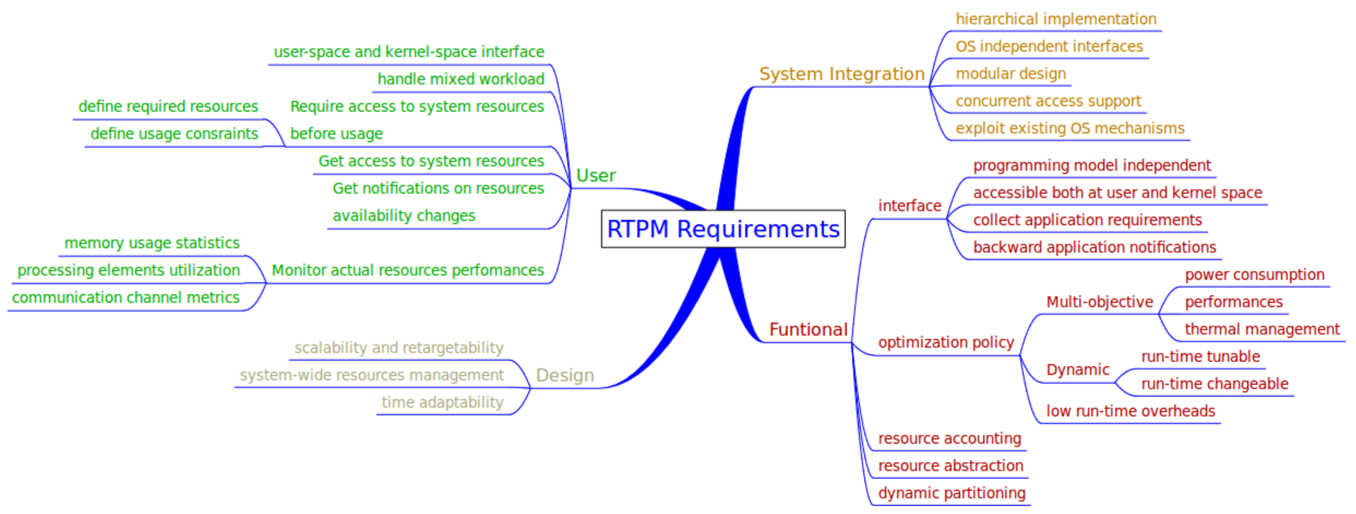

RTPM Requirements
User
user-space and kernel-space interface
handle mixed workload
Require access to system resources before usage
define required resources
define usage consraints
Get access to system resources
Get notifications on resources availability changes
Monitor actual resources perfomances
memory usage statistics
processing elements utilization
communication channel metrics
Design
scalability and retargetability
system-wide resources management
time adaptability
System Integration
hierarchical implementation
OS independent interfaces
modular design
concurrent access support
exploit existing OS mechanisms
Funtional
interface
programming model independent
accessible both at user and kernel space
collect application requirements
backward application notifications
optimization policy
Multi-objective
power consumption
performances
thermal management
Dynamic
run-time tunable
run-time changeable
low run-time overheads
resource accounting
resource abstraction
dynamic partitioning

Fig. 2: The main requirements for the RTRM tool

resource power consumptions and performances, and thus they will generally correspond to the properties defining a working mode.

d) Priority based access to resources [applications]: Access to resources is priority based: critical applications could preempt resources used by best-effort applications. In general, every application will be associated to a resource access and usage priority by means of a system configuration. In addition to a coarse grained classification based on critical and best-effort applications, it will be possible to define also a fine grained priority level. Each application will have a default priority, however a proper mechanisms (similar to the ones already available on operating systems to defined the priority of a process) will allow privileged users to change this level. Critical applications are required to use this mechanism to properly configure their priority.

e) User-space representing application [applications]: The RTRM tool allows kernel-space resident clients, however it will expect that every client has a corresponding controlling user-space application. To the purposes of the optimization policy, the user-space application will define resource access rights and priority.

f) Query resource availability [applications]: The system resources are shared among different applications which run concurrently and compete for their usage. The resource availability could change at runtime according to the working conditions, e.g., workload, hot-spot and failures. Thus, an application is required to query the RTRM tool to know about resources availability. According to the resource availability, an application will have the chance to know exactly what QoS level can be obtained from the system. This requirement is mandatory only for best-effort applications. On the contrary, critical applications could always suppose that their required resources are available and it is in charge of the RTRM to ensure that. Proper interfaces will be designed to allow clients to request information on resources by specifying:

- *Resource class*, e.g., processing elements, clusters, memory, communication channels
- *Usage constraints*, e.g., usage time, access policy, functional requirements (e.g., max latency between to PEs, minimum granted bandwidth, etc.)

g) Get and release resources [applications]: To provide a system-wide optimization, the RTRM must always have an updated and complete view on the resources state, and specifically on resources usage and availability. This information is required to properly support the *resource accounting* feature provided by the RTRM, and thus each user is required to notify when a resource is used and released. The “get” method allows to obtain a reference to a “virtual resource”, which will be properly mapped to a “physical resource” by the RTRM according to its optimization strategy and the runtime working conditions. The “release” method notifies the RTRM that the resource is not more needed by the asserting client and thus it is available for other usage or optimizations.

h) Handle RTRM notifications [applications]: Resource availability could change at runtime due to changing working conditions. Applications that have access to some resources could be requested to adapt to this changing conditions. In this case, a notification will be sent by the RTRM to the involved applications to give them the chance to reconfigure themselves.

4.2 Covered requirements

A different set of requirements could be identified which must be satisfied by the RTRM in order to provide an effective implementation of the runtime manager.

a) Monitor resource performance: To properly perform resource and performance optimization, the RTRM will relay on an updated view of the usage and behaviors of each subsystem, and their resources, with different levels of detail. This information could be exposed on request to use-case clients as well, according to their rights. For example, the RTRM will probably collect statistics on processing elements and memory usage, information on communication channels congestion (bandwidth) or behavior (latency).

b) Dynamic resource partitioning: Each application may have a different impact on the overall user experience, which defines its "priority level". Thus, the RTRM will provide some support to handle both critical workloads, that have hard requirements in terms of resource usage and execution behaviors, and best-effort workloads, for which a penalty either does not impact on perceived behaviors or produce a tolerated QoS degradation. Both classes of applications will access the available resources through a single runtime resource manager. Thus, the role of this tool is to account for available resources, and grant access to these resources to demanding applications according to their priority level. In general, critical applications are off-line profiled and optimized, for example using the DSE techniques, while the best-effort will not be optimized. In the former case, the applications will have specific resource requirements that should be granted by the RTRM in order to meet the desired and designed behaviors. In the latter case, however, a best-effort policy can be used, which could not guarantee the runtime behaviors. In any case, we want to be fair on granting access to resources usage once when they are available. To efficiently manage this scenario, the resources could be dynamically partitioned, taking into consideration current QoS requirements and resources availability. This dynamic partitioning will allow to grant resources to critical workloads while dynamically yield these resources to best-effort workloads when (and only while) they are not required by critical ones, thus optimizing resource usage and fairness.

c) Resource abstraction: An effective RTRM targeting mobile multimedia platforms, cannot disregard the context of systems which admit the presence of multiple applications, running concurrently on the same many-core computing fabric, each one having its own application-specific requirements. In such a context, the efficiency of managing the available resources is a challenging goal. The mapping of applications onto the available resources may change during the device life-time, and the current effective mapping should be based on specific quantitative metrics, e.g., throughput, memory bandwidth and execution latency. In parallel to the application-specific requirements, we also experience other non-functional aspects such as power consumption, energy efficiency and thermal profiles. The last one is especially relevant in scaling technologies like the one targeted by the upcoming many-core computing platforms. The presence of these two types of requirements makes the mapping decision more complex. To address this problem, the RTRM tool will handle a decoupled perspective of the resources between the users and the underlying hardware. The user applications should see virtual resources, e.g., the number of processing elements available, but they will not be aware of which of the physical resources are effectively available. At runtime the RTRM will perform the *virtual-to-physical mapping* according to the current *objective function* (low power, high performance, etc.) and *runtime phenomena* (process variation, temporal and spatial temperature gradients, hardware failures and, above all, workload variation).

d) Multi-objective optimization policy: The optimization policy should be system-wide and able to consider multiple metrics, e.g., power consumption, performance indexes and thermal gradients.

e) Dynamic optimization policy: The optimization policies of every abstraction levels will be runtime tunable to some extend, for instance to associate different levels of importance to the different optimization goals. Moreover, in the case of high-level and system-wide optimization policies, there will be the possibility to completely change the policy at runtime. This will enable to design multiple light-weight policies which fit well specific optimization scenarios.

f) Low runtime overheads: The overall RTRM overhead should not impact noticeably on the performance of the controlled system, both at the host-side and especially at the computation fabric side. This requirement will be satisfied by adopting an hierarchical design for the framework with control policies distributed at different levels performing a refined optimization.

5. The Task Scheduling Component

To address the challenges introduced by future embedded computing, the task scheduling component needs to fulfill the following features:

- First, a variety of applications should be supported: mobile communications, networking, automotive and avionic applications, multimedia in the automobile and Internet interfaced with many embedded control systems. These applications may run concurrently, start and stop at any time. Each application may have multiple configurations, with different constraints imposed by the external world or the user (deadlines and quality requirements, such as audio and video quality, output accuracy), different usages of various types of platform resources (processing elements, memories and communication bandwidth) and different costs (performance, power consumption).

- Second, a holistic view of platform resources should be supported. This is needed for global resource allocation decisions optimizing a utility function (also called Quality of user Experience - QoE), given the available platform resources. This QoE will allow trade-off, negotiated with the user, between diverse QoS requirements and costs.
- Third, the platform resource usage and the application mapping on the platform should be transparently optimized. This is needed to facilitate the application development and manage the QoS requirements without rewriting the application.
- Next, the task scheduling component should dynamically adapt to changing context. This is needed to achieve a high efficiency under changing environment. QoS requirements and platform resources must be scaled dynamically (e.g., by adjusting the clock frequencies and voltages, or by switching off some functions) in order to control the energy/power consumption and the heat dissipation of the platform.
- Different heuristics should be allowed, since a single heuristic cannot be expected to fit all application domains and optimization goals.
- Finally, since the task scheduling component is intended for embedded platforms, a lightweight implementation only is acceptable. To address this challenge, this component should interface with design-time exploration to alleviate its runtime decision making.

The task scheduling component manages and optimizes the application mapping taking into account the possible application configurations, the available platform resources, the QoS requirements, the application constraints, and the optimization goal. In the following we overview the issues the task scheduling component focuses on in the 2PARMA approach.

a) Interface with design-time exploration: Whereas the functional specification of an application is fixed, there may be several specific algorithms or implementations for a given application. Also an application implementation can take several forms (fixed logic, configurable logic, software) and offer different characteristics. These application configurations with associated meta-data (e.g., QoS, platform resource usage, costs) are provided at design time to enable fast exploration during runtime decisions. The interface will be extended from [12].

b) Reconfiguration: Ideally, for any application, all functionalities should be accessible at any time. However, based on the user requirements, the available platform resources, the limited energy/power budget of the platform, and the target platform autonomy, it may not be possible to integrate all these functionalities on the platform at the same time. Hence the application developer has to organize the application into application modes, each one specifying a different subset of functionalities. Moving from one application mode to another may be needed at run time due to user interactions or changes in the platform resource availability when new applications are activated. Adapting the mapping of an active application is called dynamic reconfiguration or task migration. The key challenge is of course to maintain the real-time behavior and the data integrity of the overall set of active applications. On the one hand, dynamic reconfiguration is a powerful mechanism to improve the platform utilization, avoiding some idle computing resources, while others are overloaded. On the other hand, important issues are not only task state representation and message consistency during reconfiguration, but also reconfiguration time that limits the performance of the overall system. A high-level modeling and simulation framework of this reconfiguration issue will be developed.

c) Switching: Environment changes may give rise to reselection of application modes and configurations. This resulting switching must be seamless. To that end, switching points are introduced in the applications where it is checked whether a switching is requested. A smooth transition from the current configuration to the new one is provided as follows. On the one hand, the RTRM can signal the concerned application at any time when a switching is requested. On the other hand, whenever the application reaches a switching point inside its code, the application checks whether a switching is requested. If yes, the application enters an interrupted state and transfers all relevant state information to the RTRM. The RTRM communicates the newly selected application configuration and the related received state information to the concerned cores of the platform.

d) Optimal selection of application configurations: The QoS requirements and the optimization goal are defined through the QoE manager. This goal is translated into an abstract and mathematical function, called utility function (e.g., performance, power consumption, battery life, QoS, weighted combination of them). A fast and lightweight priority-based heuristic extended from [13] selects near-optimal configurations for the active applications. It works as follows:

- It selects exactly one configuration for each active application from the multi-dimension set of configurations derived by the design-time exploration. The selection is done according to the available platform resources, in order to optimize the utility function, while satisfying the application constraints.

- The heuristic cannot always guarantee to find a feasible solution, i.e. to select a set of configurations, one per active application, within the available platform resources. In that case, the constraints of the application with the lowest priority is relaxed and the heuristic is executed once again.

e) Monitoring of application parameters: For many applications, the processing requirements to obtain results at real time can be impractical. This is due to the increasing complexity of the applications developed for advanced platforms. Nevertheless, a lower result quality might be acceptable if the results are obtained within the time required. The proposed technique will work as follows. In order to trade the result quality for the time required to obtain them, a set of well-chosen application parameters are tuned/monitored at run time once the time requirement is set by the user. The runtime monitoring technique is part of the application. It is intended to monitor the application behavior and act on the application parameters in order to meet the performance requirement while maximizing the result quality. This technique is called iteratively in the application to monitor the application execution time, check whether the deadline is met, and take decisions concerning the tuning of the application parameters. It modifies the application parameters either to improve the result quality if the deadline is met, or to lower the result quality if the deadline is not met. To enable an efficient runtime decision-making, the technique makes use of Pareto-optimal combinations of parameters also derived by the design-time exploration.

6. The Dynamic Memory Management Component

In [14], the design decisions that form the DMM design space have been presented and implemented inside a C++ library that enables the modular construction of every valid DMM configuration. Through automated exploration, combining together differing design decisions generates customized DMM configurations. Each design decision can be viewed as a building block, the DM manager is constructed by binding together DMM building blocks (Fig. 3). However, these solutions are fully static in the sense that the DMM mechanisms i.e. the fitting mechanisms, the coalescing/splitting thresholds etc. selected to customize the DMM can not be altered during runtime to adapt to workloads different than the simulated ones.

6.1 DMM Design Space

First we provide the basic DMM terminology that will be used throughout this Section.

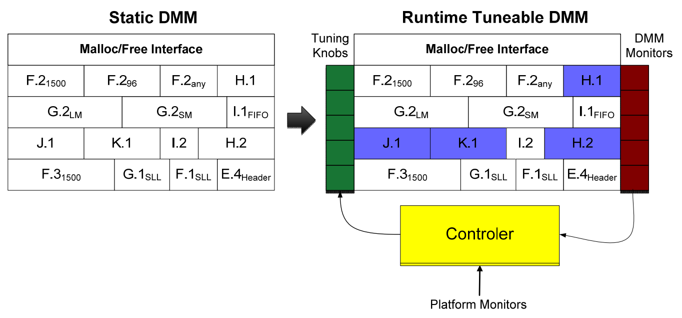


Fig. 3: Static versus adaptive DMM.

- **Heap:** Heap refers to the memory pool responsible for allocation or de-allocation of dynamic data (arbitrarily-sized data blocks allocated in arbitrary order that will live an arbitrary amount of time). In `C/C++` programming language dynamic memory management is performed through the `malloc/new` functions for allocation and `free/delete` functions for de-allocation, respectively.
- **Heap Fragmentation:** Fragmentation is defined as the maximum amount of memory allocated from the operating system divided by the maximum amount of memory required by the application. In multi-threaded memory allocators there are three types of fragmentation, namely internal fragmentation, external fragmentation and heap blow-up. Internal fragmentation [15] is generated when the DM manager returns a memory block that is larger than the initial size request. External fragmentation [15] refers to the situation that a memory request cannot be served even if there are available memory blocks in the heap that can serve the request if they merged. Finally, heap blow-up [16] is a special kind of fragmentation found in multi-threaded memory allocators.
- **Heap Memory Footprint:** The heap memory footprint refers to the maximum heap memory (allocated and freed) that the DM manager reserves. Actually, it refers to the maximum memory that is occupied, taking into consideration the memory consumed for the block's payload, the block's header and the unused space resulted from padding and alignment.

Fig. 4 shows the design/decision space concerning dynamic memory management. More specifically, we recognize the following taxonomy:

- **Inter-Heap Categories:**
  - *Architectural Scheme Category*: It determines the way the dynamic memory allocator organizes and architects its heaps in order to exploit the available thread-level parallelism into memory management.
  - *Data Coherency Decisions Category*: It deals with the existence or not and the structure of the synchronization mechanisms in order to ensure the

data coherency in each heap.

- *Inter-Heap Allocation Decisions Category*: It manages the way in which threads allocate memory in the inter-heap level. Allocation in this level is strongly connected with decisions which consider both the thread grouping in order to share a heap and the thread to heap mapping. Allocation decisions of finer granularity i.e. fit policies etc. are included into the intra-heap design space.
- *Inter-Heap De-allocation Decisions Category*: It includes the decisions concerning ownership [16] aware de-allocation of each memory block and placement decisions for the de-allocated blocks.
- *Inter-Heap Emptiness Decisions Category*: It manages the potential memory blowup of the multi-threaded application and consider decisions in order to reduce or bound the worst memory blowup.

- **Intra-Heap Categories**:
  - *Intra-Heap Block Structure Category*: It handles the data structures, which organize the memory blocks inside each heap of the DMM.
  - *Intra-Heap Pool Organization Category*: It defines per heap pools' organization i.e. single pool, one pool per size, traversing order etc.
  - *Intra-Heap Block Allocation/De-allocation Categories*: They deal with the operations that satisfy the DM allocation and de-allocation requests.
  - *Intra-Heap Splitting/Coalescing Categories*: They formalize the decisions to handle the current coalescing and splitting blocks techniques [15], i.e. the threshold logic for coalescing and splitting the blocks.

The selection of certain decisions heavily affects the coherency of other decisions within a DM manager. Thus, DMM decisions exhibit various inter-dependencies depicted as arrows in Fig. 4. We recognize two types of inter-dependencies. Excluding inter-dependencies (solid arrows) are generated when a DMM decision disables either semantically or structurally the incorporation of other categories or DMM decisions. Linked inter-dependencies (dashed arrows) are generated in cases which a DMM decision affect other decisions, but not disable their use. For example, if the coalescing frequency is set to zero, automatically the whole category K is excluded (excluding inter-dependency). However, coalescing frequency in category K, affects splitting frequency in category J, but it does not eliminate its usage, since a DM manager which only splits memory blocks but never coalesce is semantically and structurally viable DMM solution.

Each possible DMM configuration can be generated by properly combining the available design decisions, with respect to the parameter inter-dependencies. In [14], the parameter space has been used within a design-time exploration procedure, for generating application-specific MTh-DMM configurations. While in this paper we target the design of runtime adaptive DM managers, we briefly discuss the overall design space, since it forms the basis of our analysis regarding the selection of design decisions that can be configured during runtime. The DMM parameter space can be conceptually partitioned into two smaller sub-spaces, namely the inter-heap and the intra-heap sub-spaces. Inter-heap sub-space captures decisions that are shared between the threads such as overall heap organization, thread to heap mapping policies etc. Intra-heap subspace includes decisions which manage each instantiated heap's internal structure i.e. number of free-lists, allocation mechanisms etc. Each parameter sub-space is further partitioned into several decision categories (white boxes in Fig. 4), each including the DMM design decisions (colored boxes in Fig. 4).

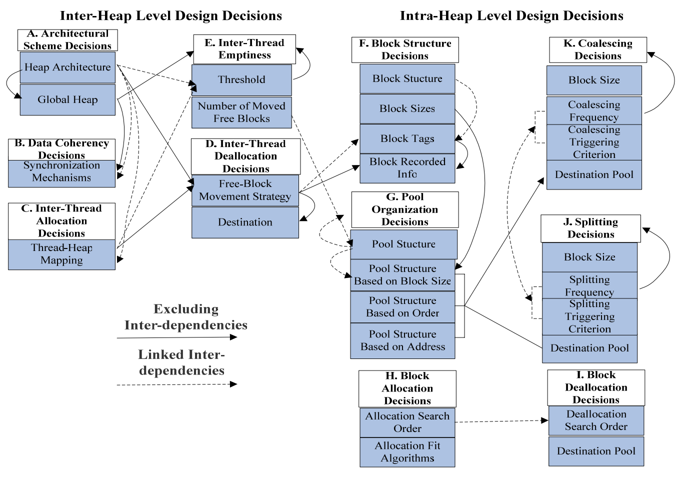

Inter-Heap Level Design Decisions
Intra-Heap Level Design Decisions
A. Architectural Scheme Decisions
Heap Architecture
Global Heap
B. Data Coherency Decisions
Synchronization Mechanisms
C. Inter-Thread Allocation Decisions
Thread-Heap Mapping
E. Inter-Thread Emptiness
Threshold
Number of Moved Free Blocks
D. Inter-Thread Deallocation Decisions
Free-Block Movement Strategy
Destination
F. Block Structure Decisions
Block Stucture
Block Sizes
Block Tags
Block Recorded Info
G. Pool Organization Decisions
Pool Stucture
Pool Structure Based on Block Size
Pool Structure Based on Order
Pool Structure Based on Address
H. Block Allocation Decisions
Allocation Search Order
Allocation Fit Algorithms
K. Coalescing Decisions
Block Size
Coalescing Frequency
Coalescing Triggering Criterion
Destination Pool
J. Splitting Decisions
Block Size
Splitting Frequency
Splitting Triggering Criterion
Destination Pool
I. Block Deallocation Decisions
Deallocation Search Order
Destination Pool
Excluding Inter-dependencies
Linked Inter-dependencies

Fig. 4: The multi-threaded dynamic memory management (MTh-DMM) design space.

6.2 Runtime Adaptive DMM

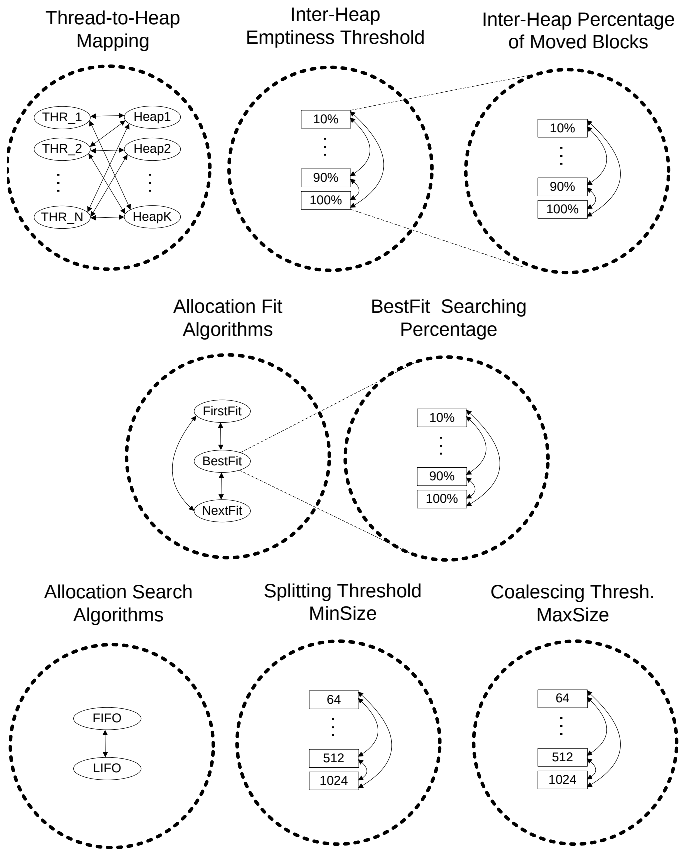
To enable runtime adaptivity of the DMM, we need to move towards the DMM solution proposed in the right side of Fig. 3. Comparing the static and the runtime-tunable DMM schemas, we recognize three major extensions that need to be considered in order to move towards more adaptive configurations:

- Identification of the subset of the available design decisions that will enhanced towards runtime reconfiguration (the identification of the runtime tuning knobs of the DM manager).
- Extension of the DM manager to provide runtime-monitoring data.
- Extraction of the decision making process that the runtime controller will implement in order to generate the new DMM configuration.

In order to extract the runtime-tunable DMM parameters, we have to evaluate the switching overheads imposed during parameter reconfiguration. DMM parameter reconfiguration occurs among the parameters of the same design decision (colored boxes in Fig. 4). For example, the parameters found in the design decision regarding the allocation fitting algorithms (category H) include the first fit algorithm, the best-fit algorithm with searching thresholds and the next fit algorithm. Since a DM manager has to incorporate a fit algorithm in order to be able to reuse the already allocated memory, we have to evaluate the trade-offs associated with switching during runtime from one fit algorithm to another. The same evaluation has to be performed for each DMM design decision.

The designer that wants to extract the runtime-tunable parameters, has to traverse through the DMM design decisions and analyzed the expected overheads in case that runtime switching is considered. The analysis indicated two classes of DMM decisions, namely the (i) the numerical DMM tunable parameters and (ii) the algorithmic DMM tunable parameters. Fig. 5 depicts the runtime-tunable DMM parameters along with the valid transitions that can be performed between them [17]. The rectangular boxes refer to the numerical DMM tunable parameters while the ellipsoid boxes refer to the algorithmic DMM tunable parameters. The DMM parameters that are numerical values are characterized as runtime-tunable since they can be tuned through a simple write at a memory address, which does not impose any severe switching overhead. The numerical DMM parameters that recognized as tunable are the following ones: (i) the threshold values regarding the maximum allowable inter-heap emptiness (category E), (ii) the decisions regarding the number of the free blocks moved to a global shared memory space whenever inter-heap emptiness crosses the allowable threshold (category E), (iii) the numerical values that set the maximum percentage in Best Fit algorithm of the available free space that has to be searched (category H), (iv) the parameters that control the triggering of splitting mechanism–minimum block size above which splitting has to be performed (category J), (v) the parameters that control the triggering of coalescing mechanism–maximum block size, resulted by the merging of two adjacent free blocks, above which no coalescing is triggered (category K).

The class of algorithmic DMM tunable parameters includes parameters that are algorithms but their tuning does not affect the architecture of the instantiated DM manager. In this case, multiple DMM management policies and mechanisms are switched during runtime and applied to the same data-structures in a mutual exclusive manner. More specifically, the analysis indicated the following algorithmic DMM tunable parameters (i) the thread-to-heap mapping algorithms in category C, (ii) the allocation search algorithms i.e. FIFO, LIFO etc., (category H) and (iii) the allocation fit algorithms, i.e. First Fit, Best Fit, Next Fit etc., (category H). In order to enable such runtime algorithmic switching, a large portion of the C DMM library has been rewritten to


Thread-to-Heap Mapping
THR_1
THR_2
THR_N
Heap1
Heap2
HeapK
Inter-Heap Emptiness Threshold
10%
90%
100%
Inter-Heap Percentage of Moved Blocks
10%
90%
100%
Allocation Fit Algorithms
FirstFit
BestFit
NextFit
BestFit Searching Percentage
10%
90%
100%
Allocation Search Algorithms
FIFO
LIFO
Splitting Threshold MinSize
64
512
1024
Coalescing Thresh. MaxSize
64
512
1024
Fig. 5: DMM runtime-tunable parameters.

completely decouple the DMM's data structures from their manipulation services. In addition, we have to mention that in this class of DMM tunable parameters no recompilation is needed, since the differing DMM mechanisms are compiled once to structure a specific DMM instance and they can be switched with each other during runtime by writing a specific global variable that guides the internal paths of the malloc/free functions.

To keep track on the state of the DM manager during runtime, proper software monitoring mechanisms have been integrated. Through software monitors, runtime statistics are collected to provide to the controller useful information at each moment for different variables. Through these runtime statistics, the controller makes the decision regarding the next DMM configuration (value assignment to the runtime-tunable DMM parameters). Through the DMM runtime statistics, the controller is keeping track of the actual fragmentation and when a certain threshold is exceeded, it reconfigures the DM manager towards a fragmentation aware configuration.

The monitor's structure can be configured at design-time. In particular, the following runtime statistics have been considered: (i) **Total memory footprint**, (ii) **Requested memory per time-slot**, (iii) **Actual memory per time-slot**

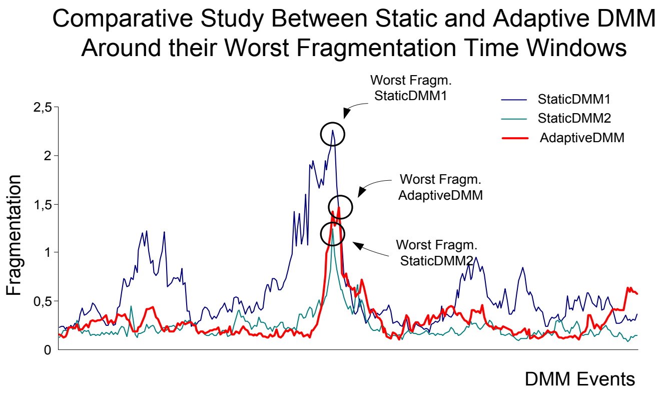


Fig. 6: Comparative fragmentation study around worst-case execution windows.

and (iv) **Per heap memory**.

The source code of the DMM library has been annotated in specific points with operations that updates the DMMStats by writing the new values of the dmmStats fields.

6.3 Adaptive DMM Evaluation

We evaluate the effectiveness of our approach based on the Larson benchmark [18], which simulates the workload for a multi-threaded server. The adaptive DMM technique is used by this application through the invocation of standard C APIs (`malloc()` and `free()`). Three different dynamic memory managers were used [17]: (1) A static performance-optimized DMM (StaticDMM1) employing a first-fit policy without splitting/coalescing mechanisms. (2) A static footprint-fragmentation optimized DMM (StaticDMM2) employing a best-fit policy with 100% search percentage and splitting/coalescing mechanisms with minimum split size. (3) A runtime tunable footprint-fragmentation optimized DMM (AdaptiveDMM).

Fig. 6 compares the three examined DMMs during a time window around their worst fragmentation cases. The proposed runtime-tunable DMM solution represents an efficient intermediate solution between the two static ones, since the proposed AdaptiveDMM is much more efficient than the StaticDMM1 and very close to StaticDMM2. Furthermore, Adaptive DMM is more efficient with respect to the StaticDMM1, with 25.1% and 69.9% gains on the average footprint and fragmentation, respectively. More details can be found in [17].

7. Conclusion

In this paper we presented the design of a new runtime resource management framework at the OS level to support many-core computing platforms. The RTRM framework is composed of a set of modules providing services to different "classes of users". We characterized the software metadata that QoS optimizations require, and we presented a methodology, as well as appropriate techniques, to obtain this information from the original application. In the 2PARMA approach, the RTRM is a component sitting in between applications and the target-platform, which is in charge of managing applications access to platform available resources. Early experimental results showed that the presented adaptive DMM used by the 2PARMA RTRM, is more efficient in comparison with the StaticDMM1, with 25.1% and 69.9% gains on the average footprint and fragmentation, respectively.

References

[1] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: the programmer's view," in *Proc. of SC*. IEEE Computer Society, 2010, pp. 1–11.

[2] STMicroelectronics and CEA, "Platform 2012: A Many-core programmable accelerator for UltraEfficient Embedded Computing in Nanometer Technology," 2010. [Online]. Available: http://www.cmc.ca/en/NewsAndEvents/~/media/English/Files/Events/20101105_Whitepaper_Final.pdf

[3] A. Sangiovanni-Vincentelli, "Quo vadis, sld? reasoning about the trends and challenges of system level design," *Proc. of IEEE*, vol. 95, no. 3, pp. 467 –506, 2007.

[4] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh, "A quick safari through the reconfiguration jungle," in *Proc. of DAC*. ACM, 2001, pp. 172–177.

[5] E. Keller, G. Brebner, and P. James-Roxby, "Software decelerators," in *Field-Programmable Logic and Applications*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2778, pp. 385–395.

[6] M. A. Al Faruque, R. Krist, and J. Henkel, "Adam: run-time agent-based distributed application mapping for on-chip communication," in *Proc. of DAC*. ACM, 2008, pp. 760–765.

[7] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet, "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles," in *Proc. of DATE*. IEEE Computer Society, 2005, pp. 234–239.

[8] N. Shirazi, W. Luk, and P. Y. K. Cheung, "Run-time management of dynamically reconfigurable designs," in *Proc. of FPL*. Springer-Verlag, 1998, pp. 59–68.

[9] P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in *Proc. of CODES+ISSS*. ACM, 2003, pp. 45–51.

[10] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu, "Parallel programming models for a multi-processor soc platform applied to high-speed traffic management," in *Proc. of CODES+ISSS*. ACM, 2004, pp. 48–53.

[11] V. Nollet, D. Verkest, and H. Corporaal, "A safari through the mpsoc run-time management jungle," *Journal of Signal Processing Systems*, vol. 60, pp. 251–268, 2010.

[12] C. Ykman-Couvreur, P. Avasare, G. Mariani, C. Silvano, and V. Zaccaria, "Linking run-time resource management of embedded multi-core platforms with automated design-time exploration," *IET Comput. Digit. Tech.*, vol. 5, no. 2, pp. 123–135, 2011.

[13] C. Ykman-Couvreur, V. Nollet, F. Catthoor, and H. Corporaal, "Fast multi-dimension multi-choice knapsack heuristic for MP-SoC run-time management," *ACM TECS*, 2011.

[14] S. Xydis, A. Bartzas, I. Anagnostopoulos, D. Soudris, and K. Pekmestzi, "Custom multi-threaded dynamic memory management for multiprocessor system-on-chip platforms," in *Proc. of IC-SAMOS*, 2010, pp. 102 –109.

[15] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Proc. of IWMM*. Springer-Verlag, 1995, pp. 1–116.

[16] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: a scalable memory allocator for multithreaded applications," *SIGPLAN Not.*, vol. 35, pp. 117–128, November 2000.

[17] S. Xydis, I. Stamelakos, A. Bartzas, and D. Soudris, “Runtime tuning of dynamic memory management for mitigating footprint-fragmentation variations,” in *Proc. of PARMA Workshop*. VDE Verlang, 2011, pp. 27–36.

[18] P.-A. Larson and M. Krishnan, “Memory allocation for long-running server applications,” in *Proc. of ISMM*. ACM, 1998, pp. 176–185.

A New Approach to Control and Guide the Mapping of Computations to FPGAs

João M. P. Cardoso[3*], Razvan Nane[4], Pedro C. Diniz[2], Zlatko Petrov[1], Kamil Krátký[1], Koen Bertels[4], Michael Hübner[5], Fernando Gonçalves[8], José Gabriel de F. Coutinho[6], George Constantinides[6], Bryan Olivier[7], Wayne Luk[6], Juergen Becker[5], Georgi Kuzmanov[4]

[1]Honeywell International s.r.o., HON, Czech Republic (coordinator)
[2]Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento em Lisboa, INESC-ID, Portugal
[3]Universidade do Porto, Faculdade de Engenharia (FEUP), Portugal
[4]Technische Universiteit Delft, TUD, The Netherlands
[5]Karlsruhe Institute of Technology, KIT, Germany
[6]Imperial College London, Imperial, UK
[7]ACE Associated Compiler Experts b.v., ACE, The Netherlands
[8]Coreworks – Projectos de Circuitos e Sistemas Electrónicos S.A., CW, Portugal

Abstract - *Field-Programmable Gate-Arrays (FPGAs) are becoming increasingly popular as computing platforms for high-performance embedded systems. Their flexibility and customization capabilities allow them to achieve orders of magnitude better performance than conventional embedded computing systems. Programming FPGAs is, however, cumbersome and error-prone and as a result their true potential is often only achieved at unreasonably high design efforts. The REFLECT (Rendering FPGAs to Multi-Core Embedded Computing) project's design flow consists of a novel compilation and synthesis system approach for FPGA-based platforms. Its design flow relies on Aspect-Oriented Specifications to convey critical domain knowledge to optimizers and mapping engines. An aspect-oriented programming language, LARA (LAnguage for Reconfigurable Architectures), allows the exploration of alternative architectures and design patterns enabling the generation of flexible hardware cores that can be incorporated into larger multi-core designs. We are evaluating the effectiveness of the proposed approach for applications from the domain of audio processing and real-time avionics. In this paper we describe the REFLECT approach and present a number of examples and results using REFLECT's compilation and synthesis tools.*

Keywords: FPGAs, Compilers, Aspect-Oriented Specifications, Reconfigurable Computing

* Contact author: João M. P. Cardoso
Universidade do Porto, Faculdade de Engenharia (FEUP)
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
Email: jmpc@acm.org

1 Introduction

Contemporary Field-Programmable Gate-Arrays (FPGAs) are powerful and sophisticated devices able to implement complex high-performance embedded computing systems [1][2]. Customization allows FPGAs to achieve orders of magnitude better performance than conventional processor systems as they can implement directly in hardware specific high-level operations crystallized as custom computing units. As a result, FPGAs are becoming commonplace in embedded systems and even in some cases in high-performance systems.

However, the benefits of FPGA-based systems over traditional systems come at a cost. The large numbers of potential custom functional units, coupled with the many choices of interconnecting these units, make the mapping of computations to these hardware/software architectures a highly non-trivial process. As a result, the mapping of complex applications to these architectures is accomplished by a labor intensive and error-prone manual process. Programmers must assume the role of hardware designers to synthesize or program the various custom hardware units in low level detail, and also to understand how these units interact with the software portions of the application code. Programmers must partition the computation between the code that is executed on traditional processor cores and the code that is to be synthesized in hardware with the consequent partitioning and mapping of data. The complexity of this mapping process is exacerbated by the fact that the custom computing units may internally exhibit different computation models (e.g., data flow, concurrent synchronous processes) and architectural characteristics (e.g., parallelism, customization), or that the various cores might support functional- or data-parallel concurrent execution paradigms.

It is the aim of the REFLECT project [3][4] to develop an approach to help designers achieve efficient FPGA-based heterogeneous multi-core computing systems. Our approach involves combining different areas of research: aspect-oriented specifications, hardware compilation, design patterns and hardware templates. The goal of this project is to develop, implement and evaluate a novel compilation and synthesis approach for FPGA-based platforms. We rely on Aspect-Oriented System Development (AOSD), with foundations on aspect-oriented programming (AOP) [5][6], to convey critical domain knowledge to mapping engines while preserving the advantages of a high-level imperative programming paradigm in early software development as well as programmer and application portability. We leverage aspect-oriented specifications using LARA (LAnguage for Reconfigurable Architectures), a new domain-specific aspect-oriented programming language, to specify complementary information, optimizations, and mapping strategies. The REFLECT design flow has unique characteristics that allow it to both adapt to and meet different non-functional requirements (e.g., safety requirements [7]).

We are evaluating the effectiveness of the proposed approach using real-life applications provided by REFLECT's industrial partners. This evaluation includes the development of two demonstrators: an avionics mission-critical embedded system and an audio encoder. Both these codes raise realistic and demanding challenges that highlight the power and impact of the base techniques and methodologies in the proposed REFLECT approach over traditional design and mapping methodologies.

In this paper we describe the REFLECT design flow [4] and how aspects and strategies are used to map computations to FPGA based systems. In particular, we show experimental results obtained by mapping kernels from two avionics applications, which illustrate strategies suited to meet high-performance requirements.

This paper is organized as follows. Section 2 presents the architecture being currently targeted in REFLECT. Section 3 illustrates the REFLECT design flow and the main tools being developed, used and extended. Sections 4 and 5 describe, respectively, two application case studies and the use of aspects and design patterns in REFLECT. Section 6 presents the results currently achieved when mapping these codes to a REFLECT target architecture. Section 7 presents related work. Finally, Section 8 concludes this paper.

2 REFLECT Target Architecture

Although the REFLECT design flow can target a multitude of reconfigurable architectures, it currently targets an architecture consisting of a general-purpose processor (GPP) connected to Custom Computing Units (CCUs) based on application-specific architectures. Both these components use a shared memory approach possibly connected via data communication channels. The application-specific architectures are implemented with reconfigurable logic (as in reconfigurable fabrics such as FPGAs) and are generated from the C code of the application being compiled.

An example of the target architecture is depicted in Fig. 1 and consists of a GPP, such as a Xilinx MicroBlaze or IBM PowerPC, tightly coupled with a reconfigurable hardware fabric where Custom Computing Units (CCUs) is defined according to application needs. Collectively, the CCUs define a reconfigurable computing system implementing various execution models in space and in time and can consist of specialized hardware templates. The coupling and interface between the processor and the CCUs are inspired on the Molen machine and programming paradigm [8]. We also envision high-end computing systems (akin to HPC systems) that are composed of several of these base reconfigurable systems interconnected using traditional multiprocessor organization arrangements (e.g., bus, hypercube or trees) and logically organized as distributed memory or shared memory heterogeneous multiprocessor systems. From a software-stack perspective, the heterogeneous system is viewed as a co-processor device of a host system. Reconfigurable resources are not exposed to the operating system of the host system. Instead, there is a simple resident "monitor" system responsible for the communication of data and synchronization with the host and/or I/O channels. The development of an operating system is beyond the scope of the REFLECT project.

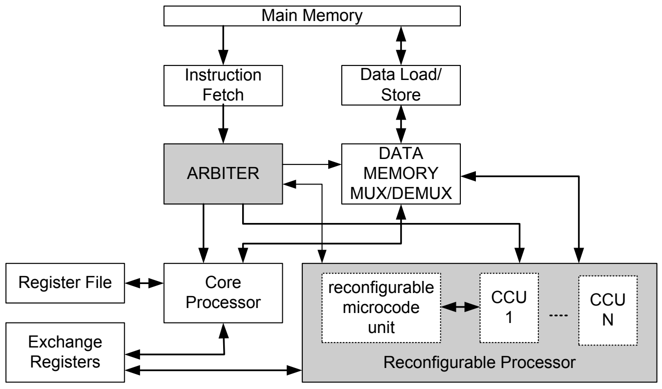


Fig. 1. Block diagram of the target architecture used by REFLECT.

The CCUs and Core processors use a shared memory system and a register file (XREG) to communicate data [8]. For a particular implementation there is a maximum number of CCUs supported by the Molen machine organization and specific FPGA area constraints for CCUs. Dynamic reconfiguration techniques are foreseen for virtualizing the hardware resources. This will allow applications to use during execution more CCUs than the physically available ones.

For prototyping, REFLECT's consortium is using the ML510 Embedded Development Platform which includes a Xilinx Virtex-5 XC5VFX130T FPGA (XC5VFX130T-2FFG1738CES). This FPGA includes two PowerPC 440 processors (PPC440) as hard cores, clocked at the maximum frequency of 400 MHz. The ML510 board consists of several peripheral interfaces, DRAM memory, and it was selected based on its suitability for prototyping and evaluation of high-performance embedded computing systems.

3 REFLECT's Design Flow

A goal of Aspect-oriented programming (AOP) [5][6] is to improve code modularity by allowing aspects to convey crosscutting concerns. Examples of these include the instrumentation of application code to monitor, debug, and visualize data. When mapping computations to reconfigurable hardware architectures, we are interested in the specification and use of different implementations for the same code. Each of those implementations may take advantage of the specific characteristics of a particular target system, such as memory organization or functional unit capabilities. Aspect modules can thus be used to describe features that a compiler, and other tools in the mapping flow, can use to derive customized solutions, i.e., solutions more suitable to the target architecture and meeting requirements. In this context, we distinguish three main abstractions in the REFLECT compilation/synthesis flow, described in detail in the following sub-sections.

Application Aspects

Application Aspects allow developers to specify application characteristics such as precision representation (e.g., error less than 1E-03), input data rates (e.g., 30 frames per second) or other non-functional requirements such as safety and reliability requirements for the execution of specific code sections/functions. These features act as "requirements" for acceptable design solutions and cannot be easily expressed using common programming languages (such as C). These aspects might be internally decomposed into a number of low-level aspects that guide the REFLECT design flow to generate an implementation which meets the requirements. Some low-level aspects and the ordering of their application can be specified by the user using strategies[1] or can be defined by a Design Space Exploration (DSE) approach. Strategies can thus be seen as rules that force the design flow to apply a specific design pattern.

Design Patterns

Design patterns act as a collection of transformations or "actions" to be used to transform the application code in search for a design implementation with specific features or performance characteristics. For example, an execution time requirement for a specific code section might require the concurrent execution of a specific function. This in turn will require a design pattern or transformation (via the application of strategies) that performs loop unrolling and data partitioning so that data are available to all the concurrently executing units.

[1] The term is used herein in a more generic way than in [9].

Hardware/Software Templates

These templates, which can include a mix of hardware and software implementations, define the "lower" layers of the mapping hierarchy. These templates are characterized in terms of resource usage and number of clock cycles in a specific custom design (e.g., as in FPGAs) as they expose the characteristics of the target resources to the design flow. As an example, the hardware versions of a FIFO or streaming buffer and the software implementation of the same components can be considered hardware/software templates.

Overall, the developer defines, as a first step, the application aspects related to the code at hand, relying on a wealth of existing design patterns and hardware/software templates together with DSE support to find a suitable set of transformations or design patterns that can lead to a specific feasible implementation. The REFLECT compilation flow will benefit from aspects to produce efficient FPGA implementations. This approach is also applicable to other contemporary reconfigurable and non-reconfigurable computing architectures.

In REFLECT we focus on the use of aspects, strategies, and design pattern modules for four types of features:

- SPECIALIZING: Specialization of a design for the particular target system (e.g., specializing data types, numeric precision, and input/output data rates);
- MAPPING AND GUIDING: Specification of design patterns, which embody mapping actions to guide the tools in some decisions (e.g., mapping array variables to memories, specifying FIFOs to communicate data between cores, use specific dynamic reconfiguration techniques, use specific fault-tolerance schemes).
- MONITORING: Specification of which implementation features, such as current value of a variable or the number of items written to a specific data structure, provide insight for the refinement of other aspects.
- RETARGETING: Specification of certain characteristics of the target system in order to make the tools adaptable and aware of those characteristics (i.e., retargetable).

An important component of the aspect-oriented programming model is the notion of a *weaver*. A weaver is a compilation framework component that receives as input the code of the application augmented with the aspect modules, and produces a new version of the code for the application as result of applying the descriptions (rules) in the aspect modules. The aspect modules usually define a *pointcut* and an *advice* [10]. An example of a pointcut and an advice are respectively "find invocations of functions" and "test if array arguments have size greater than 0". In this case, a weaver will insert additional code at each function invocation site to test if array arguments have size greater than zero.

We now describe the overall REFLECT compilation and synthesis design flow. In REFLECT, an input application in C is implemented as a system consisting of one GPP connected to one or more hardware cores (CCUs), as presented in Fig. 1. Such an application is partitioned for software and hardware components according to developer-provided requirements.

Fig. 2 depicts the main stages of the REFLECT design flow for the generation of hardware and software components. The design flow includes the Harmonic tool [11], which is used as a source-to-source transformation tool. Harmonic is responsible for analyzing and giving hints about code compliance[2], for partitioning the input applications in software/hardware components, for including the communication primitives, and for providing support for code insertion. Aspects related to these transformations are identified by the Aspect Front-End tool and input to Harmonic as Aspect-IR. Harmonic also performs cost estimates for a given platform to assist in the software/hardware partitioning of the input application code. When performing hardware/software partition, the software components are augmented with primitives to communicate data and to synchronize their execution with the hardware components.

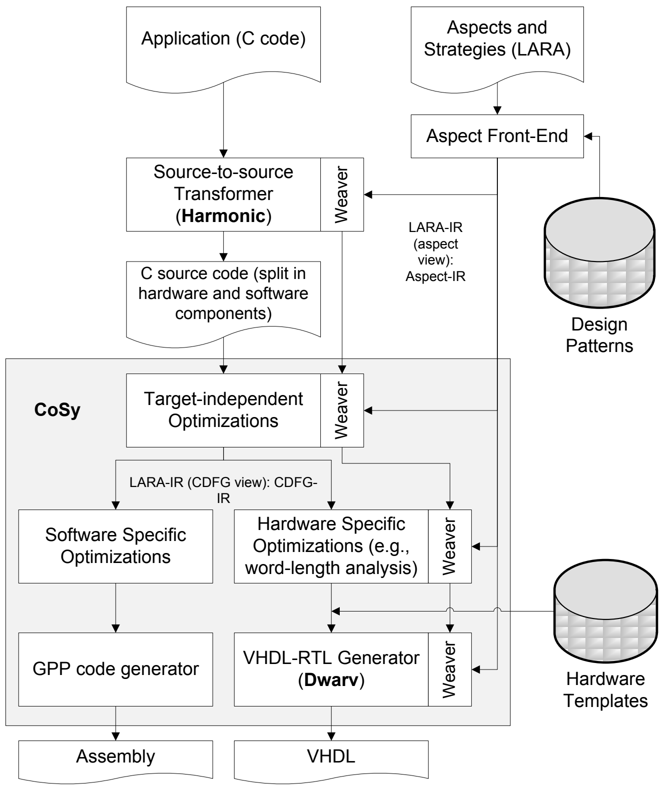

Application (C code)
Aspects and Strategies (LARA)
Aspect Front-End
Source-to-source Transformer (Harmonic)
Weaver
LARA-IR (aspect view): Aspect-IR
Design Patterns
C source code (split in hardware and software components)
CoSy
Target-independent Optimizations
Weaver
LARA-IR (CDFG view): CDFG-IR
Software Specific Optimizations
Hardware Specific Optimizations (e.g., word-length analysis)
Weaver
GPP code generator
VHDL-RTL Generator (Dwarv)
Weaver
Hardware Templates
Assembly
VHDL

Fig. 2. REFLECT's design flow and its main stages.

The C code output from Harmonic is then input to a CoSy [12] compiler. This CoSy compiler directly invokes the subsequent design flow components, including the weavers to implement some aspects, further target-required optimizations and transformations, and word-length optimizations. Then, the CoSy compiler is responsible for the generation of hardware (hardware components) and RISC code (software component). These components communicate through a common intermediate representation based on a CDFG (Control/Data Flow Graph) represented using CoSy CCMIR (Common CoSy Medium-level Intermediate Representation) and including data-dependences and annotations. This representation is common among the design flow components integrated in CoSy as depicted in Fig. 2.

Strategies, defined as sequences of aspects to be applied, are described in LARA using constructs based on aspect-oriented programming and scripting languages. These strategies enhance DSE via try-and-feedback schemes, implementation of the design patterns and their strategies, and alternative flows for host simulation and target compilation. The LARA-IR carries all information between the components: the transformed and gradually specialized and mapped representation of the application, and all kinds of attributes, not only simple attributes (such as memory spaces of variables) and structured (such as loop-nest information and dependences), but also those that support aspects. At some point in the design flow, the intended partitioning is reflected in the LARA-IR by creating one partition per target architecture and having separate further design flows for each one.

For hardware synthesis, the REFLECT flow uses a tool based on DWARV [13] and integrated in CoSy. As a result, our design flow generates VHDL for the hardware kernels using the same LARA-IR and the same options for arranging the order of transformations as described above. In particular, it applies transformations required to translate a computation from a Von Neumann model of computation to a structural model more suitable for FPGAs. Also, in this phase, the flow implements word-length optimization identified in earlier phases of the mapping. DWARV also implements the weaving phases of the flow what are related to hardware mapping and carries out the DSE for generating high-quality VHDL. To accomplish this, our tool flow is based on a CDFG representation LARA-IR view. This LARA-IR (CDFG view) is then input to DWARV to generate the VHDL descriptions of the hardware modules to be included in the final system as CCUs.

The software components are mapped to a RISC processor core and compiled using CoSy. A further design flow for the software components may include the generation by CoSy of a low-level C representation of the part of the application that should run in the processor, which is fed through a specialized compiler and linker (such as mb-gcc, or ppc-gcc).

Aspect modules, strategies, and design patterns bring to REFLECT's design flow the flexibility and modularity needed to obtain better results and implementations aware of certain concerns. The engines responsible for the application of the concerns described in the aspect modules can take advantage of code refactoring, code transformations (e.g., loop transformations), term-rewriting, etc.

Our approach to maintain aspect modules as primary entities which are not embedded in the application code is

[2] For instance, the VHDL generator used in the back-end of CoSy may not support all C programming constructs.

important to preserve the code's readability and maintenance. This also promotes the reusability of aspect modules and strategies. Multiple aspects and strategies can be applied to the same input application specialized according to the target system organization (e.g., including hardware cores, interface between the GPP and the hardware cores, memories connected to the FPGA, possible precisions). This approach leads to better adaptability of the tools to the specific target and/or non-functional requirements.

In the REFLECT design flow, an aspect includes two main sections: the *select* and the *apply* sections. A *select* section indicates the join points to which the user associates one or more actions specified in the *apply* sections[3]. Our join point model extends traditional join point models of AOP languages such as AspectJ and AspectC++. In our case, join points include system components, code artifacts (loops, functions, variables, assignments, etc.), and code sections (identified by specific pragmas). Each join point artifact has a number of attributes. Those attributes can be used by the actions (specified in the apply sections of the aspect). For instance, a *function* join point includes as attributes, its name, the number of lines, the number of statements, the hardware cost, and the latency. A custom computing unit (CCU) join point may have as attributes the clock frequency, the maximum hardware area, etc. Actions can depend on the attributes for a particular join point, and they can define values for those attributes. Most attributes are defined by the stages of the REFLECT design flow.

Fig. 3 depicts an aspect that can be used to map a function with name "fir" to hardware (in our example to a CCU [8]). This aspect invokes an aspect named "strategy1" which includes optimization rules, user's knowledge, mapping strategies, target architecture properties, and other information specific to the function. The aspect also specifies two constraints related to input data ranges and noise power.

```
aspectdef maxmizePerformance
  sel1 select: *.function{name="fir"}  // specification of pointcut
  apply to Sel1: map to hardware       // map action
  apply to Sel1: call strategy1        // call action
  constraints:                         // constraints
    define function{"fir"}.arg{output}.noise_power <= 1e-3;
    define function{"fir"}.arg{input}.range = -40..120;
  end
end
```

Fig. 3. Example of an aspect specifying non-functional requirements.

Each type of action is associated to a specific stage in the REFLECT design flow. For instance, the *optimize* action is associated to the CoSy compiler instance and includes compiler optimizations such as loop unrolling, scalar replacement, loop fusion, loop fission, code hoisting, word length analysis, and data-type conversion.

Aspect actions (apply section) can be of different types. Table 1 presents the current type of actions being considered. These actions include mapping and optimization directives as well as directives to specify the insertion of code in specific join points (used for monitoring and instrumenting) to define properties and to instruct tools to report information (e.g., values of attributes).

Table 1. Current keywords associated to actions.

Action (keyword)	Description
insert	insertion of code
report	instructs the tools in the REFLECT design flow to report information
optimize	instructs the tools for specific optimizations, including code transformations
map	Instructs the tools to map computations and data structures to specific hardware components
define	defines properties that can be used by the tools
call	invoke other aspects

4 Case Studies

We now describe opportunities for the application of various Aspects described above to the hot-spots of two applications from the avionics domain: 3D Path Planning and Stereo Navigation. In this section we briefly describe their computations, and the following section presents experimental results of the application of a set of high-level code transformations guided by the use of Aspects.

4.1 3D Path Planning

The 3D Path Planning core computation defines a 3D path $r(t)$ between the current vehicle position and required goal position, using Laplace's equation (see, e.g., [15][16]). It solves Laplace's equation in the interior of a 3D region, guaranteeing no local minima in the interior of the domain, leaving a global minimum of $v(r) = -1$ for r on the goal region, and global maxima of $v(r) = 0$ for r on any boundaries or obstacle. A path from any initial point $r(0)$, to the goal, is constructed by following the negative gradient of the potential, v.

Fig. 4 illustrates the computational contribution of the main 3D Path Planning functions to the global execution time on a PPC440 core (at 400 MHz) embedded in a Xilinx Virtex5 FPGA. The iteration steps represent over 90% of the global execution and are performed by the *gridIterate* function.

A possible code implementation for the *gridIterate* function is depicted in Fig. 5 where, for simplicity, details such as global variables definition, initialization and functions, are omitted. This function uses a 3D matrix representing an obstacle map (array *obstacle*) and outputs a 3D matrix representing the potential matrix (array *pot*).

[3] The select and apply are conceptually equivalent to the *pointcut* (set of join points) and *advice* in AspectJ [10] and have been previously used in the context of an aspect language for MATLAB [14].

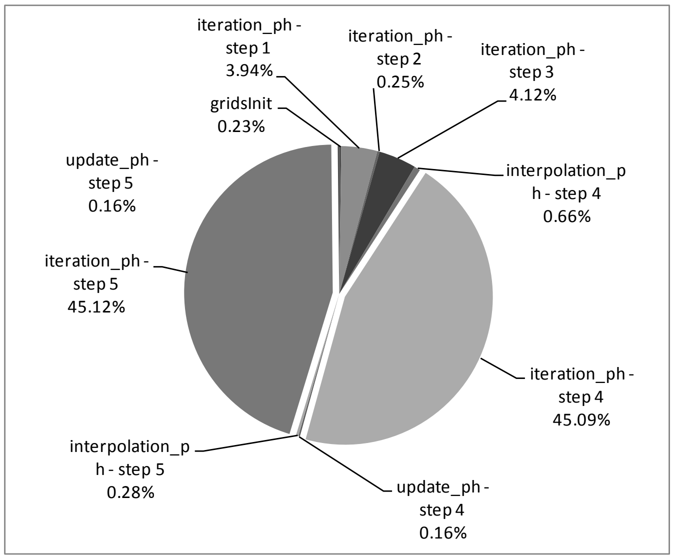


Fig. 4. Contribution of the main 3D Path Planning functions to the global execution time (obtained by using hardware timers).

```
#define ITER_STEPS_NUM ...
void gridIterate(int* obstacles, float* pot) { ...
 for (it = 0; it < ITER_STEPS_NUM; it++) {
  for (i = 1; i < (X_DIM - 1); i++) {
   for (j = 1; j < (Y_DIM - 1); j++) {
    for (k = 1; k < (Z_DIM - 1); k++) {
      val = obstacles[i][j][k];

      if (val == 1) pot[i][j][k] = POTENTIAL_ZERO;
      else if (val == -1) pot[i][j][k] = POTENTIAL_ONE;
      else {
         acc = (accType)pot[i-1][j][k]+
               (accType)pot[i+1][j][k] +
               (accType)pot[i][j-1][k]+
               (accType)pot[i][j+1][k] +
               (accType)pot[i][j][k-1]+
               (accType)pot[i][j][k+1];
         pot[i][j][k] = FIX_CORRECT(acc * SCALE);
      }}}}}}
```

Fig. 5. Function *gridIterate* C code from the 3D Path Planning application.

4.2 Stereo Navigation

The Stereo Navigation (*StereoNav*) application is intended for airplane localization when the GNSS (Global Navigation Satellite System) used in airplanes is temporarily unavailable. The idea of the application is that from two independent images derived from cameras, looking in approximately the same direction, features can be extracted (dominant entities in the image are invariant to rotation and translation). Using two cameras taking simultaneous images allows for localization of the features in 3D-space. The main components of the algorithm include: Debayering (optional), Rectification, Feature extraction, Feature matching, 3D reprojection, and Robust pose estimation and refinement.

Fig. 6 illustrates the contribution of the main StereoNav functions to the global execution time when executing the application in the PPC440 core (at 400 MHz) in the Xilinx Virtex5 FPGA. We used hardware timers to measure the execution time of each function. The core computation of the *StereoNav* application is presented in function *harrisTile_model_step* (identified in Fig. 6 as “do_tile”) and consists of a sequence of 8 convolutions using two kinds of *conv* function (*ConvVBConst* and *ConvVBRepl*). A section of the C code of the *ConvVBConst* function is depicted in Fig. 7.

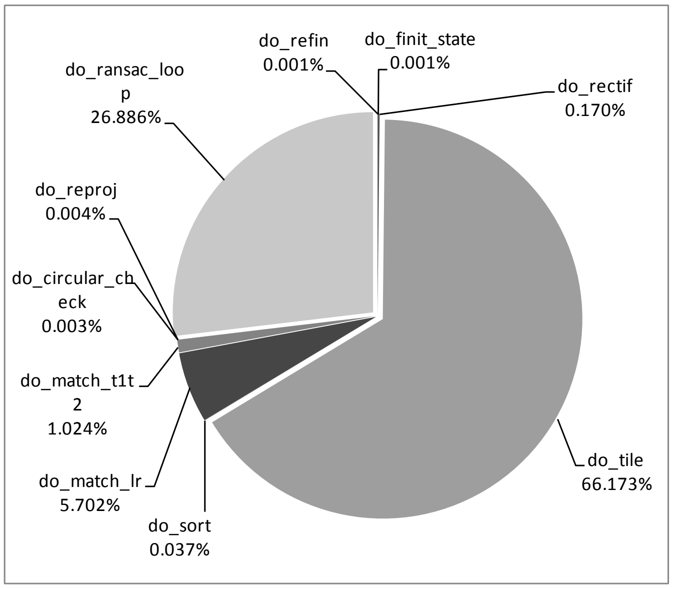


Fig. 6. Contribution of the main *StereoNav* functions to the global execution time (obtained by using hardware timers).

```
void ConvVBConst (..) { ...
for (IDXB_1U=SSTART_1U; IDXB_1U<=SEND_1U; IDXB_1U++){
   ...
   for (IDXB_0U=SSTART_0U;IDXB_0U<=SEND_0U;IDXB_0U++) {
     ... acc = 0.0F;  ...
     for (HIDXA_1U=0;HIDXA_1U<=HEND_0_1U;HIDXA_1U++) {
      for (HIDXA_0U=0;HIDXA_0U<=HEND_0_0U;HIDXA_0U++) {
        acc = u[IDXALIN_0U] * h[buf1Idx] + acc;  ...
      }  IDXALIN_0U = (uDims[0U] - hDims[0U]) + IDXALIN_0U;
     }  y[IDXBLIN_0U] = acc;
   }
  }
... // second part of the convolution
   //(with 5 FORs nested and a function call in the innermost loop
}
```

Fig. 7. Part of the C code of the *ConvVBConst* function.

5 REFLECT Approach

This section illustrates the use of aspects, design patterns and hardware/software templates for the two case studies described in the previous section.

5.1 3D Path Planning: gridIterate

For the *gridIterate* function we consider the optimizations and strategies presented in Table 2 and Table 3, respectively. For this case study we focus on data conversions from floating-point to fixed-point, partial loop unrolling of the innermost loop, and multi-dimensional arrays transformed to uni-dimensional arrays. As the data elements defining *obstacles* have values in the set {-1, 0, 1} and that the *pot* data represent real values in the range [0, 1], scaling analysis can result in an optimized fixed-point representation.

Table 2. Optimizations considered for *gridIterate*

Transf.	Description
T 1.1	Float to fixed-point representation
T 1.2	Unroll innermost loop by 2
T 1.3	Shift by powers of two promoted to wires
T 1.4	Linearization of multi-dimensional arrays.
T 1.5	Array indexing transformed as wire concatenation and wiring component
T 1.6	Code motion (loads moved from if-else conditions)

Table 3. Strategies considered for *gridIterate*

Strategy Name	Transformations					
	T1.1	T1.2	T1.3	T1.4	T1.5	T1.6
gridIt-baseline			✓	✓		
gridIt-fixed1	✓		✓	✓		
gridit-fp1		✓	✓	✓	✓	✓
gridit-fixed2	✓	✓	✓	✓	✓	✓

To convert from multi-dimensional to uni-dimensional arrays, an index such as [i][j][k] is translated to (i*Y_DIM + j)*Z_DIM + k allowing the subsequent application of operator strength reduction on the calculations for the indexing of the array variables as well as concatenation of addressing bits when the various array dimensions are aligned at specific power-of-two address boundaries.

A transformation to the *gridIterate* function considers multi- to uni-dimension transformation, code motion, and the use of a macro that can be implemented as a concatenation of wires to calculate the index of the arrays *pot* and *obstacles*. The code motion is applied based on the following explanation. In order to decrease the number of references to the *pot* array variable, the writes to *pot[i][j][k]* existent in all branches of the *if-else* construct in the code can be moved to after the *if*. This transformation also allows the parallel loads of *obstacles* and *pot* data when the two arrays are bound to different memories or to a multi-port memory. The code motion of the accesses to *pot* allows earlier scheduling of *pot* data loads. If the innermost loop is unrolled twice, we increase the impact of pipelining memory accesses, and we reuse a load to *pot* thus reducing the number of loads per two k-loop iterations.

The mapping of functions to hardware can be guided by the user through aspects. Fig. 8 illustrates a generic aspect to map a given function to a CCU identified by an input id. For instance, by associating a specific instance of this aspect as

```
map2hardware("gridIterate", 1)
```

the *gridIterate* function will be mapped to a CCU of the target architecture identified by "1". Further, the user may use conditions to make an action dependent on the value of certain attributes. For instance, the use of

```
condition: $function.no_lines < 500
```

in the aspect in Fig. 8 instructs the weaving process to map a function to hardware only if the function is less than 500 lines of code long (attributes as hardware cost can also be used).

```
aspectdef map2hardware(string $name, int $id=1)
  select A: function{name=$name}
  apply to A: map to hardware(ccu.id=$id)
end
```

Fig. 8. An aspect with an action to map a function to hardware.

5.2 Stereo Navigation: Convolutions

For the *convolution* functions we consider the optimizations and the strategies presented in Table 4 and Table 5, respectively. The convolution functions *ConvVBConst* and *ConvVBRepl* include invocations to the functions *PadBConst* and *PadBRepl*, respectively. For this second case study we consider scalar replacement, function inlining, and the specialization of the convolution functions according to the calls. This specialization is mainly dedicated to the elimination of loop headers for loops with only one iteration, as well as to the unrolling of innermost loops when their number of iterations is less than or equal to three.

Table 4. Optimizations considered for the convolution functions.

Transf.	Description
T2.1	Scalar replacement
T2.2	Function inlining
T2.3	Specialization of each call to conv
T2.4	Loop header elimination
T2.5	Loop unrolling of innermost loops with number of iterations <= 3

Table 5. Strategies considered for the convolution functions.

Function	Strategy	Transformation				
		T2.1	T2.2	T2.3	T2.4	T2.5
ConvVBConst	stg01	✓	✓			
	stg02	✓	✓	✓		
	stg03	✓	✓	✓		✓
ConvVBRepl	stg04	✓	✓			
	stg05	✓	✓	✓		
	stg06	✓	✓	✓	✓	
	stg07	✓	✓	✓	✓	✓

Fig. 9 illustrates the LARA specification of a strategy that considers function inlining, loop unrolling, and function specialization. The use of *section* (e.g., *section{"l1"}*) in the select sections of the aspects refers to specific code sections identified by pragmas included by the user in the code as *#pragma joinpoint section="l1"*. Note, however, that this is indicative and the final syntax and constructs of LARA may be slightly different.

```
import inline1;
import unroll1;

aspectdef Const_Config1
  select A: function{"harrisTile_model_step"}.section{"l1"}.
          call{"ConvVBConst"}.body;
  apply to A:
     define{$ sEnd[1U]=94, $ sEnd[0U] =94, $hEnd_0[1U]=3;
              $hEnd_0[0U] =3,  $numBSec =8, $ sEnd_0[1U] =94,
              $ sEnd_0[0U]=94, $hEnd_1[1U] =3,$ hEnd_1[0U] =3}
   optimize specialize();
  end
end
aspectdef Repl_Config1
   select A: function{"harrisTile_model_step"}.section{"l2"}.
          call{"ConvVBConst"}.body;
   apply to A:
     define{...}
     optimize specialize();
   end
end
aspectdef Repl_Config2
   select A: function{"harrisTile_model_step"}.section{"l3"}.
           call{"ConvVBConst"}.body;
   apply to A:
      define{...}
     optimize specialize();
   end
end

call unroll1("ConvVBConst");
call unroll1("ConvVBRepl");

call inline1("PadBConst");
call inline1("PadBRepl");

// two imported aspects:
aspectdef inline1(String $name) // inline functions identified bt
$name
  select: function{$name};
  apply: optimize inline();
end

aspectdef unroll1(String $name) // unroll loops if the number of
iterations is <=3
   select A: function{ $name}.loop{*};
   apply to A,B:  optimize loop_unrolling($loop);
   condition: $loop.no_iterations <= 3
end
```

Fig. 9. Examples of aspects and possible strategy for the *harrisTile_model_step* function.

6 Experimental Results

We apply the strategies outlined in Section V to the functions described in Section IV. As our design flow is not yet fully automated, the results presented here correspond to the manual application of the described aspects and strategies. We consider software versions of the functions and compare the results of running them on the PPC440 at 400 MHz against hardware versions obtained by the DWARV compilation and synthesis flow. Unless otherwise stated, the software versions are generated with the gcc compiler using the -O3 compilation option. The hardware versions are clocked at 200 MHz.

3D Path Planning: gridIterate

The use of floating-point data types for the *gridIterate* (*gridIt-baseline* and *grid-fp1*), single precision in this case, favors the use of dedicated hardware implementations. With respect to floating-point solutions, the hardware implementations achieve speedups of 2.15× and 2.83× over the software related versions for *gridIt-baseline* and *gridIt-fp1*, respectively. In the case of the fixed-point solutions (*gridIt-fixed1* and *grid-fixed2*), the hardware implementations achieve speedups of 1.05× and 5.56× over the software solutions.

Considering the FPGA resources used for different hardware implementations of the same function (*gridIterate*) the strategies used for *gridIt-fixed1* and *gridIt-fixed2* imply more hardware resources due to the presence of a 64×64 bit multiplication in the fixed-point multiplication vs. the presence of a 23×23 bit multiplication for the single precision floating-point version (*gridIt-baseline* and *gridIt-fp1*). This is reflected in the use of 2.2× the number of DSP48 and 1.23× the number of slices. The last two strategies (correspondent to *gridIt-fp1* and *gridIt-fixed2*) achieve implementations with more slices than the one using the strategy considered by *gridIt-baseline* and *gridIt-fixed1*. This is due to the fact that *gridIt-fp1* and *gridIt-fixed2* consider loop unrolling of the innermost loops by a factor of 2.

Stereo Navigation: Convolutions

For the function *ConvVBRepl* of the Stereo Navigation application, the use of strategy stg07 allows a speedup of 1.30× by the FPGA design over the software version with the same strategy. For *ConvVBConst* the FPGA design achieves speedups of 2.31× and 2.54× over the best non-specialized software implementation considered (PPC –O3) and non-specialized FPGA implementation, respectively.

The use of strategies stg05 and stg06 in the *ConvVBRepl* functions leads to a decrease in slices of 32.39% and 44.33%, respectively. For *ConvVBConst* the number of slices decreases by 7.62% when using stg03 vs. stg01 for similar clock frequencies.

7 Related Work

Compiling high-level programming languages to FPGAs is a topic that has been extensively addressed by academia and industry (see, e.g., [17] for a survey of representative approaches). However, it is understood that, due to the large gap between software and hardware, compilers for FPGAs still have a long way to go before being able to generate efficient customized architectures for complex applications. In addition, the hardware to be generated depends on non-functional requirements, which are not embedded in the code of the application and result in extensive work by the designer to explore options and to modify the code of the application.

To the best of our knowledge this is the first time an aspect-oriented approach is being used to holistically control and guide the stages of a design flow, in order to compile C applications to embedded systems implemented using FPGAs. By extending the possible join points to system artifacts, beyond possible artifacts in programs, and by applying to both those types of artifacts actions specified in a programming language, we are exposing users to powerful mechanisms to control and guide the design flow and to program strategies (mostly defining design patterns) that best suit user requirements.

Recent efforts to map computations to FPGA-based systems include the hArtes tool chain [17]. hArtes also includes as a source-to-source transformation stage the Harmonic [11] tool, and as a hardware compiler a previous version of DWARV [13]. However, the hArtes approach supports neither an aspect-oriented approach nor strategies and design patterns.

8 Conclusions

This paper presented part of the REFLECT project's approach to a design flow targeting FPGA systems. At the core of our approach is a new programming language, named LARA, allowing users to specify aspects and strategies (reflecting design patterns) that guide the design flow to meet desired non-functional requirements.

Specifically, in this paper we focused on the description of aspects and strategies to two critical functions from two avionics applications: Stereo Navigation, and 3D Path Planning. We presented experimental results of the application of selected aspects and the corresponding strategies. The results highlight the modularity and reusability of aspects and design patterns in the proposed approach, thus providing early evidence that this approach can lead to a substantial cost decrease of code maintenance while promoting design space traceability.

9 Acknowledgment

This work is partially supported by the European Community's Framework Programme 7 (FP7) under contract No. 248976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the European Community. The authors are grateful to all team members of the REFLECT project for their help and support.

10 References

[1] S. Hauck, and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon)*, Morgan Kaufmann, November 16, 2007.

[2] M. Gokhale, and P. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*, Springer, 1st Edition, Dec., 2005.

[3] REFLECT website: http://www.reflect-project.eu.

[4] J. M. P. Cardoso, et al., "REFLECT: Rendering FPGAs to Multi-Core Embedded Computing," *Book Chapter in Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*, Springer (to appear).

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect Oriented Programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, Finland. Springer-Verlag LNCS, vol. 1241, June 1997.

[6] G. Kiczales, "Aspect-Oriented Programming," in *ACM Computing Surveys (CSUR)*, special issue: position statements on strategic directions in computing research, 1996. 28(4es).

[7] Z. Petrov, K. Krátký, J. M. P. Cardoso, and P. C. Diniz, "Programming Safety Requirements in the REFLECT Design Flow," in *IEEE 9th Int'l Conference on Industrial Informatics (INDIN'2011)*, Caparica, Lisbon, Portugal, July 26-29, 2011 (to appear).

[8] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The Molen Polymorphic Processor," in *IEEE Transactions on Computers*, Nov. 2004, Vol. 53, Issue 11, pp. 1363-1375.

[9] R. Lämmel, E. Visser, and J. Visser, "Strategic programming meets adaptive programming," In *Proc. of the 2nd Int'l Conference on Aspect-Oriented Software Development (AOSD '03)*, Boston, Mass., March 17-21, 2003. ACM, New York, NY, USA, pp. 168-177.

[10] J. Gradecki, and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley, 2003.

[11] W. Luk, J. Coutinho, T. Todman, Y. Lam, W. Osborne, K. Susanto, Q. Liu, and W. Wong, "A High-Level Compilation Toolchain for Heterogeneous Systems," in *Proc. IEEE International SOC Conference (SOCC'09)*, Sept. 2009, pp. 9-18.

[12] ACE CoSy compiler development system, http://www.ace.nl/compiler/cosy.html

[13] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator," in *Proc. of the 17th Int'l Conference on Field Programmable Logic and Applications (FPL'07)*, Aug. 2007, pp. 697–701.

[14] J. M. P. Cardoso, P. Diniz, M. Monteiro, J. Fernandes, and J. Saraiva, "A Domain-Specific Aspect Language for Transforming MATLAB Programs," in *Domain-Specific Aspect Language Workshop (DSAL'2010)*, part of the 9th Int'l Conference on Aspect-Oriented Software Development (AOSD'2010), March 15-19, 2010.

[15] C. I. Connolly, J. B. Burns, R. Weiss, "Path planning using Laplace's equation," in *Proc of IEEE Int'l Conference on Robotics and Automation*, Cincinnati, OH, USA, May 1990, vol. 3, pp. 2102-2106.

[16] K. P. Valavinis, T. Herbert, R. Kollura, and N. Tsourveloudis, "Mobile Robot Navigation in 2-D Dynamic Environments Using an Electrostatic Potential Field," in *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 30, issue 2, March. 2000, pp. 187-196.

[17] J. M. P. Cardoso, P. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A Survey," in *ACM Computing Surveys (CSUR)*, Vol. 42, Issue 4, Article 13 (June 2010), 65 pages.

[18] K. Bertels, V. Sima, Y. Yankova, G. Kuzmanov, W. Luk, J. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, and A. Michelotti, "HArtes: Hardware-Software Codesign for Heterogeneous Multicore Platforms," in *IEEE Micro*, 30(5): 2010, pp. 88-97.

SESSION

REGULAR PAPERS

Chair(s)

DR. PEDRO C. DINIZ

A heterogeneous reconfigurable System-on-Chip: MORPHEUS

Florian Thoma[1], Matthias Kühnle[1], Arnaud Grasset[2], Paul Brelet[2], Philippe Millet[2], Philippe Bonnot[2], Fabio Campi[3], Nikolaos S. Voros[4], Wolfram Putzke-Roeming[5], Axel Schneider[6], Michael Hübner[1], Klaus D. Müller-Glaser[1], Jürgen Becker[1]
[1]Karlsruhe Institute of Technology, Germany
[2]Thales Research & Technology, France
[3]STMicroelectronics SRL, Italy
[4]Technological Educational Institute of Mesolonghi (consultant to Intracom Telecom SA), Greece
[5]Deutsche Thomson OHG, Germany
[6]Alcatel-Lucent Deutschland AG, Germany

Abstract— *The exponential increase of CMOS circuit complexity along the last decades has lead to two growing problems. The increasing Non-recurring Engineering (NRE) costs of ASICs or System-on-Chips are becoming only affordable to the highest volume applications. Additionally the design methodologies have not kept pace with the rising complexity leading to a rising design productivity gap. Research into reconfigurable architectures and NoC (Network-on-Chip) communication systems have shown paths for mitigating these problems for lower volume applications. In this paper, we present the European Integrated Project MORPHEUS (IST 027342). It advocates an innovative approach of heterogeneous, dynamically reconfigurable SoCs consisting of accelerators of various reconfiguration granularity connected by a NoC and supported by an integrated toolset for spatial and sequential design. The power of this approach is demonstrated with four applications from the industrial environment.*

Keywords: Dynamic reconfiguration, Reconfigurable architecture, Reconfigurable computing, heterogeneous computation, MPSoC, NoC

1. Introduction

MORPHEUS [1] copes with the challenges of rising complexity and the enlarging design productivity gap by developing a global solution based on a modular heterogeneous System-on-Chip platform providing the disruptive technology of dynamically reconfigurable computing including completed by a software oriented design flow and a consistent toolset. This three-and-half year project provided a modular silicon demonstrator in 90nm technology composed of complementary run-time reconfigurable building blocks which has been demonstrated with four complementary test cases.

The paper is organized as follows. In section 2, we define the main objectives of the MORPHEUS project. Section 3 presents related works of other teams in this field. Next, we present the developed architecture for these objectives, with emphasis on the reconfigurable components and in section 5 the prototype representing one possible implementation of this architecture. In section 6, we discuss the toolchain for compilation and design space exploration. The characteristics of the target applications are introduced in section 7, and finally, we present the conclusions of this paper in section 8.

2. Project objectives and challenges

The MORPHEUS project main objective is to make embedded system applications efficiently exploit the benefits of reconfigurable computing. The MORPHEUS platform is thus a complex System-on-Chip that exploits run-time programmability at different levels to provide a competitive computing solution. Applications are often characterised by various kinds of processing (with control dominant parts or stream processing parts, etc.). Therefore the architecture chosen for their implementation might be heterogeneous, made of several types of processing units having different structural granularity (fine grain for logic, coarser grain for arithmetic, etc).

The challenge of MORPHEUS is to master this heterogeneity and complexity thanks to an homogeneous system architecture and the appropriate toolchain. MORPHEUS' most significant innovation is thus the exploitation of the heterogeneous computational engines in the context of an integrated homogeneous system. This implies the capability of ensuring efficient communication means where communications delays at system level can be masked by computation delays in the computing units. Also the control system of such architecture through efficient hardware mechanisms is crucial and the corresponding application development tools are the key of their usability. The challenge is thus also to provide a high level language for the global platform control and its associated compiler ensuring that reconfiguration delays can be optimally be masked by computation delays.

3. Related Work

Most competing state-of-the-art SoC are based on a single CPU, or on a combined CPU and DSP enhanced with some dedicated hardware accelerators. In this systems the host CPU (central-processing unit) mainly acts as an application controlling engine (Nomadik [2], OMAP [3], and PXA [4]). MPSoC implementations differ from MORPHEUS because the processing nodes of MPSoCs provide a uniform means of computation of a single granularity, similar I/O bandwidths and access patterns. [5] and [6] describe the Pleiades approach, a reconfigurable architecture template, which was targeted at low power consumption rather than processing efficiency. It uses its NoC to connect different elements of one given reconfigurable-architecture array rather than intrinsically different, coarser reconfigurable architectures. In [7] another homogeneous multi-core runtime reconfigurable SoC is presented, where two cores are connected through a lightweight communication infrastructure. Other reconfigurable architectures are optimized for specific applications [8] or are only configurable at design time [9]. There are several languages and tools available for use with reconfigurable platforms. SystemC and SystemVerilog are mainly used for high-level descriptions of complete systems. Application specific instruction processors and accompanying tools can be developed with languages and tools like Mescal, ArchC, LisaTek, Chess/Checkers and XPRES Compiler. Design space exploration and development for coarse-grained reconfigurable hardware can be done with FELIX [10]. C-based languages like CatapultC, GARP, Mitrion-C, SA-C and Impulse C are available for parallelized applications but are limited to specific target architectures. A more stream-oriented approach is available with tools like Matlab/Simulink, Scilab, AccelDSP and languages like Array-OL, Stream-C, ASC and SNET. APIs available for different traditional languages are DRI, VSIPL++, MPI, OpenMP, OpenUH and Stream Virtual Machine. UML, sysML and SpecEdit can be used for formal specification of applications. For a more system-level approach to application mapping tools like PTOLEMY II, R-Stream and Gaspard beside the already mentioned Matlab/Simulink can be used. The projects EPICURE and RECONF offer tools targeted especially for reconfigurable platforms.

4. Architecture

4.1 Overview

One problem of most available or proposed reconfigurable processing architectures is the balancing of generality, flexibility and efficiency. MORPHEUS draws the logical conclusion to integrate three Heterogeneous, dynamically Reconfigurable processing Engines (HREs) which support different but complementary styles of reconfigurable computing, in one highly scalable platform.

Two fundamental design decisions regarding the MORPHEUS platform architecture were made at an early stage of the project and verified with a SystemC model. The first decision was to use a central controller to manage the whole platform. The second decision was the usage of the Molen paradigm [11] to control the HREs from the central processor, since it allows parallel execution of the functions on the HREs and provides a mechanism to pass parameters to the functions.

Since the data flow organization between the HREs is not fixed by the architecture, very flexible and efficient software controlled usage of the HREs is possible. In typical data stream processing applications consecutive processing steps can be mapped onto different HREs and connected in a pipeline style. Thereby, the difficult task is to find a well balanced split of the application to the pipeline stages. Alternatively the same HRE also can be used in a time-multiplex style for consecutive processing steps. A combination of both approaches allows to find a well adjusted load balance for all available HREs.

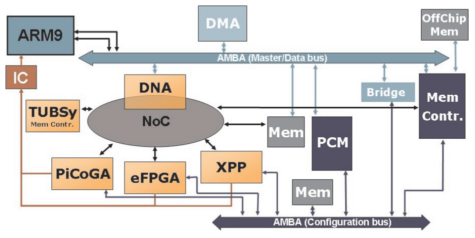


Fig. 1: MORPHEUS architecture

Furthermore, the analysis of the targeted applications clearly showed that special emphasis has to be put to a fast dynamic reconfiguration mechanism.

4.2 Structure

The MORPHEUS hardware architecture, which is depicted in fig. 1, comprises three heterogeneous, reconfigurable processing engines which target different types of computation:

- XPP from PACT XPP Technologies is a coarse-grain reconfigurable array targeting algorithms with huge computational requirements but mostly deterministic control- and dataflow.
- DREAM is based on the PiCoGA core from STMicroelectronics. It is a medium-grained reconfigurable array consisting of 4-bit ALUs and LUTs where up to four configurations may be kept concurrently and mainly targets computation intensive algorithms that can run iteratively using only limited local memory resources.
- FlexEOS from Abound Logic is an embedded Field Programmable Gate Array (eFPGA). Thus, it is a fine-grain reconfigurable device based on LUTs.

All system control is handled by an ARM926EJ-S processor. Other processing tasks can be mapped onto the ARM processor only if they do not compete for processing power or bandwidth with the primary tasks.

As the HREs in general will operate on different clock domains, they are de-coupled from the system clock domain by data exchange buffers (DEB) consisting of dual ported (dual clock) memories either configured as FIFOs or double buffers. The ARM processor, which is controlling all data transfers, has to ensure the in-time delivery of new data to the DEBs to avoid stalling of the HREs.

According to the Molen paradigm each HRE contains a set of exchange registers (XR). Through the XRs the ARM and HREs can exchange synchronization triggers (e.g. new data available or computation has finished) as well as a limited number of parameters for computation (e.g. start address of

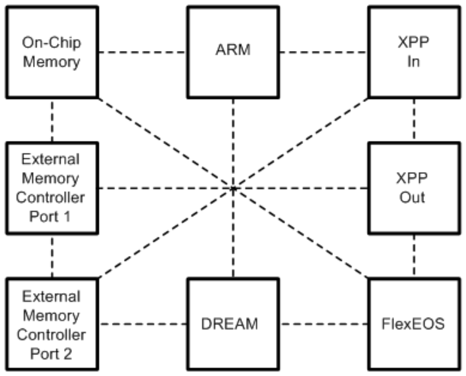

On-Chip Memory
ARM
XPP In
External Memory Controller Port 1
XPP Out
External Memory Controller Port 2
DREAM
FlexEOS

Fig. 2: MORPHEUS NoC topology

new data in the DEBs). Buffering of local data can be done in the on-chip data memory. As a third level of memory the platform uses external SRAM memory.

As dynamic reconfiguration of the HREs imposes a significant performance load on the ARM processor, a dedicated reconfiguration control manager (PCM) has been designed to serve as a respective offload-engine. The PCM analyzes which configuration is needed for the next processing steps.

All system modules are connected via multilayer AMBA busses. Separate busses are provided for reconfiguration and/or control and data access. Additionally a NoC based on ST's spidergon technology [12] with the characteristic topology has been integrated to increase bandwidth. To reduce the burden on the ARM system controller, DMA and DNA (direct network access) controllers available for loading data and configurations. Fig. 2 shows the topology of the integrated NoC where each node is connected to the neighbours and to one diagonally across the net. The main idea for optimizing the topology is to place NoC nodes with high intercommunication demand directly adjacent to one another, since a direct interconnection link exists between such nodes.

5. Chip prototype

5.1 Technology Options

The MORPHEUS chip (fig. 3) was designed and fabricated using a CMOS 90nm process of ST Microelectronics [13]. The reference voltage is 1V, although the chip was designed to operate in standard temperature conditions in the range [0.9V-1.2V].

The controlling ARM processor, the XPP, and all structures related to data and configuration transfers are implemented by Standard Cells. The design utilizes standard cell libraries supporting two different threshold voltages: specific low threshold, high speed cells are utilized only on critical paths, accounting to 11.4% of the standard cells area, while high threshold, low leakage cells were utilized in the rest to reduce power consumption. The MORPHEUS chip features a total area of 110 mm^2, 97 M transistors including L1 and L2 memory and IOs (table 1). In all, memory occupies 16% of chip area, IO 6% and Std cells 55%. It features a static power consumption of 230mW. Dynamic consumption of the devices strongly depends on the current computation and all clock domains are dynamically scalable leading to an average dynamic consumption around 700mW with local peaks above the 3W mark. Local and global IR-Drop phenomena, very common in similar high-performance architectures, are significantly limited by the multiple, asynchronous clock domains.

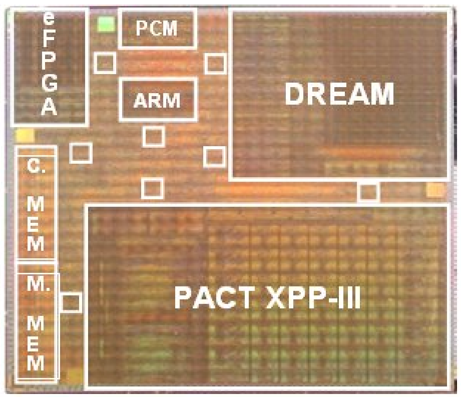


Fig. 3: Photograph of the MORPHEUS prototype

Table 1: Area Distribution in the MORPHEUS Prototype Chip

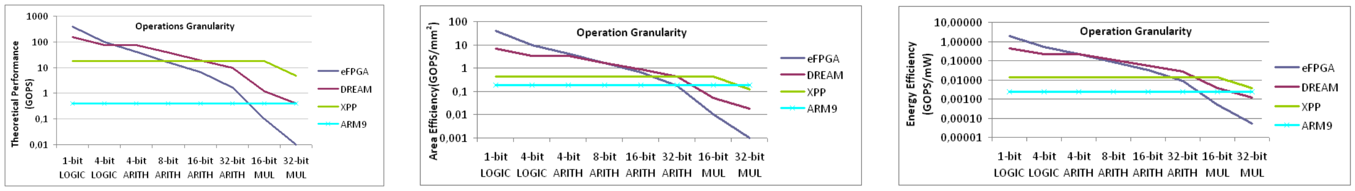
Total Chip Area [mm^2]	10.48 x 10.48
Custom Layout macros area [mm^2]	16.1 (15%)
Embedded memory (SP/DP) [Kbit]	5970 / 1961
Embedded memory area [mm^2]	17.5 (16%)
IO ring area [mm^2]	6 (5.4%)
Standard cell count [Kgates]	8507 (33%)
Overall Transistor count [Millions]	97
Standard cell density	52%
High Speed vs. Low Leakage Std. cells (area ratio)	11.4%

Timing closure in worst case conditions (wc, 0.9V, 125°C) was at 250 MHZ for the top level interconnect, 200MHZ for DREAM, 150MHz for XPP and 100MHz for the FlexEOS module. In typical conditions the respective performance figures raise to 320, 260, 180 MHz respectively. Referring to worst case conditions, these figure lead to a 12.5Gbit/sec throughput on the NoC. Choosing 16-bit operations as reference, the test chip is capable to deliver 60 GOPS at 0.6 GOPS/mm^2 and 20 GOPS/W.

5.2 Performance Assessment

The mapping of several arithmetical operations of different bit widths to the different HREs allows an evaluation of peak performance of the system. The algorithms are running local on the HREs and do not cover data transfer and synchronisation across the hole system. Fig. 4 shows how each of the computation engines offers performance that significantly depends on the operations bit-width: at low bit widths, predictably, the FlexEOS yields larger efficiency. In the 8- to 16-bit range, the efficiency of the DREAM engine is evident. After the 16-bit mark, XPP can exploit is significant computation parallelism.

The graphs clearly demonstrate the advantages enabled by the MORPHEUS approach: for every computational kernel



(a) (b) (c)

Fig. 4: Theoretical Performance of the MORPHEUS HREs

Table 2: Comparison of MORPHEUS performance metrics against alternative computation platforms

Device	Technology	GOPS	GOPS/mm^2	GOPS/W
MORPHEUS	CMOS090	60	0.6	20
Microprocessor	CMOS090	0.35	0.15	1
ASIC	CMOS090	n.a.	10	800
TU iVisual [14]	CMOS180	77	1.16	205
KAIST Vect Proc [15]	CMOS130	125	3.4	214
Philips Xetal II [16]	CMOS090	107	1.44	170
Cell	CMOS090	205	0.85	2.5
Xilinx Virtex-II 8000	FPGA	450	?	3.5
Virtex-4-SX55	FPGA	50	?	3.5

there is a component that is most suited to its needs. Applications that consist of several significantly different computation kernels will benefit most from this property as it is possible to partition and pipeline the application across all HREs. The reported performance figures classify MORPHEUS roughly in the range of vector processors in the same technology node (see table 2). On the other hand, MORPHEUS allows significant advantages against FPGAs and general purpose embedded processors using the same technology node.

6. Toolset

MORPHEUS application development is based on a number of successful tools from industry. These tools are integrated to a seamless design flow from a high level description toward target executable code. The normal drawback of heterogeneous architectures is that each component requires its own special knowledge and experience to conquer its complexity and implement a high performance solution to the requirements, which additionally hampers the partitioning of the applications. An ambition of the MORPHEUS toolset is to abstract the heterogeneity and the complexity of the architecture in such a way that software designers may program it without a deep knowledge and experience on HREs (hardware architecture, languages, programmability and so on).

The developer uses C-code for application entry. Based on this code and additional information, the toolset generates the bitstreams for the three HREs. This way the MORPHEUS toolset enables a significant productivity improvement of development on reconfigurable computing platforms. The toolset is organized in two levels.

At the top level, the user has a global view and understanding of the whole application and focuses on partitioning and mapping each function on a HRE as well as the global scheduling of the HREs, the ARM, the memory transfers and the synchronizations. Internally the programming model is based on the MOLEN paradigm [11]. It provides a solution to most of the issues that arise such as: an opcode explosion, a limitation of number of parameters, a lack of parallel execution and a modular approach. At runtime, this level is managed by the OS by scheduling the loading of proper HREs' binary code against the whole execution and the data communication.

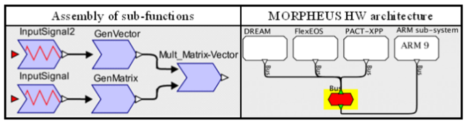
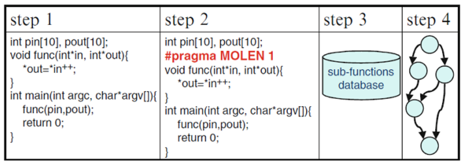


Fig. 5: Accelerated function low-level design



```
int pin[10], pout[10];
void func(int*in, int*out){
    *out=*in++;
}
int main(int argc, char*argv[]){
    func(pin,pout);
    return 0;
}
```

```
int pin[10], pout[10];
#pragma MOLEN 1
void func(int*in, int*out){
    *out=*in++;
}
int main(int argc, char*argv[]){
    func(pin,pout);
    return 0;
}
```

Fig. 6: Programming steps

At the lower level the user focuses on local optimization of the code of each accelerated function. They are designed by graphically assembling sub-functions which are described in C-code and then mapped on a set of HREs. Fig. 5 shows such an assembly of sub-functions, and the description of the MORPHEUS architecture used to map each sub-function.

Thanks to the toolset, the development of an application on MORPHEUS is nearly as easy as writing a sequential C-code for a General Purpose Processor. When implementing an application on the architecture, the designer splits it into control parts (executed on the ARM processor) and computation intensive parts (mapped on the HREs) by the steps in fig. 6:

1) The application is written in standard C-language.
2) The programmer identifies the functions that must be accelerated with a pragma in the application C-code.
3) These accelerated functions are captured inside a graphical environment, called SPEAR [17], by connecting and assembling elementary sub-functions written in C.
4) The toolset generates the bitstream for every HRE.

The SW compilation flow manages the creation of a software executable running on the main processor, directly from

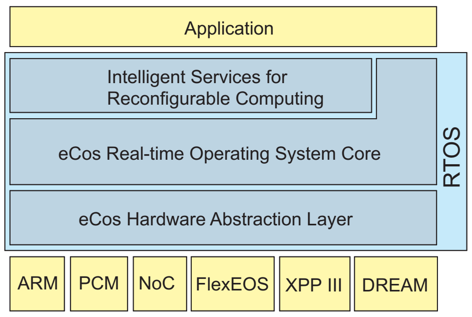

Application
Intelligent Services for Reconfigurable Computing
eCos Real-time Operating System Core
eCos Hardware Abstraction Layer
RTOS
ARM
PCM
NoC
FlexEOS
XPP III
DREAM

Fig. 7: Structure of the RTOS

the application code of the step 2. The COSY compiler [18] replaces the calls of accelerated functions in the application code with the MOLEN system calls. The RTOS has a layered structure which is shown in Fig. 7. The bottom layer is the hardware abstraction layer which provides a more uniformed access to the reconfigurable hardware and the system infrastructure with virtualized XRs. It provides also the basics for a pipeline services between the HREs.

The Spatial Design framework is the part of the toolchain dedicated to development of specific code for each HREs and consist of several tools (fig. 8). It includes programming the HREs, generating configuration bitstreams but also managing the communications to feed these accelerators with data. A Control Data Flow Graph (CDFG) format is used as an intermediate and technology independent format inside the framework. High-level synthesis techniques are used in MADEO, which acts as a back-end for the framework [19].

A second objective is to provide domain-specific models and languages to application programmers. Operations are modelled as a directed acyclic graph (fig. 9 part “a”), in a formalism called Array-OL which is well suited to represent deterministic, data intensive data-flow applications such as the kind of operations accelerated on HREs. SPEAR automatically generates a CDFG model of the accelerated function and communication parameters to feed the HRE. SPEAR enables to manage, in a coherent framework, both the HW interface of the HRE (i.e. addressing of local buffers) and the data transfers and synchronisations. The Cascade tool [20] is used to generate the CDFGs for the elementary functions. Despite the large number of tools, the toolset remains user’s friendly as it is closely integrated and automated.

To execute multi-threaded applications, their dynamic nature requires a central management. In MORPHEUS, this is provided by the Intelligent Services for Reconfigurable Computing (ISRC) [21] layer on top of an RTOS which handles the interwoven topics of scheduling and allocation. If there are multiple implementations of the requested operation available it makes a choice at run-time depending on the current status. Because of the dynamic allocations, the communication between the application and the reconfigurable units is also only indirect and handled by ISRC by programming the DMA/DNA controllers to transfer data between memory and the DEBs of the HREs. The linking

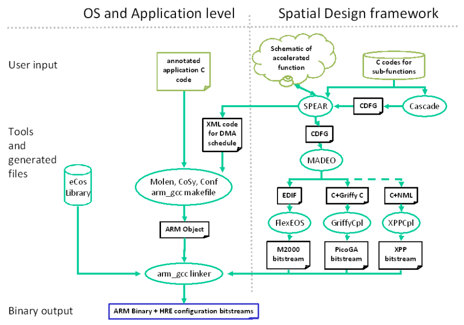


Fig. 8: Simplified internal view of the tool chain

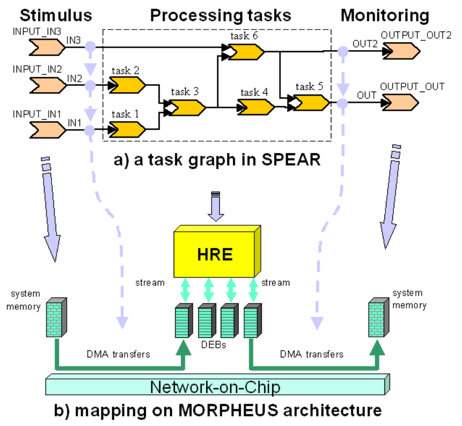


Fig. 9: Overview of an operation mapping on a HRE

of a transfer to an operation with SPEAR allows to migrate the transfer to the new HRE. The interface of ISRC extends the MOLEN directives with BREAK for synchronization with parallel operations and RELEASE for discarding no longer needed bitstreams.

In the current development status, the main limitations are located in the Spatial Design part that supports the C-language structures and data types with restrictions, and does not yet target the XPP. The configuration bitstreams generated with the PACT proprietary tools [22] can however be easily integrated into the compilation flow of the toolset.

7. Applications

Four different case studies borrowed from complementary domains with different reconfigurations needs have been used to evaluate and validate the MORPHEUS platform and toolset.

7.1 Wireless telecommunications

The emerging IEEE 802.16j standard for Mobile Broadband Wireless Access systems is the base for this application. The standard provides for a baseline PHY chain, with a large number of optional modes. A device that can reconfigure efficiently between different such modes is the main motivation for using the MORPHEUS technology. The demonstrator is part of a PHY layer implementation incorporating a word-level processing block (128-point FFT), followed by a QAM symbol demapper, capable of supporting modulation schemes ranging from QPSK to QAM64. Due to constraints of the prototype chip, it is not possible to fit both the FFT and the QAM Demapper into the DREAM. However, the capability of the MORPHEUS system for loading bitstreams for different HW accelerators at runtime offers a convenient solution.

The SPEAR tool was used for capturing the target application: one CDFG was generated for the FFT block and one for the QAM demapper. They were used by MADEO for the generation of the bitstreams for DREAM, as well as the control code running on the ARM processor, responsible for reconfiguration and for managing the data transfers to and from the DEBs. The implementation of the QAM demodulation functions on the HREs took less than a day effort for each of them and did not require any HREs knowledge. The design on the SPEAR graphical interface can easily be built and requires a few minutes. Globally a few days were necessary for the implementation of the application including C code for ARM and SPEAR capture. The application shows an improvement from 925487 cycles on pure ARM to only 19751 cycles using ARM together with DREAM.

7.2 Network routing systems

The rapid evolution of the telecommunication market forces manufacturers of high-end telecommunication equipment to develop their new products using draft versions of standards, because the standardization processes take a very long time. The risk of costly design re-spins can be lowered by providing reconfigurable SoCs which allow late hardware changes in the design process or even design changes after deployment.

Today's telecommunication networks require data rates up to 40 Gbit/s per single line, which cannot be provided by FPGAs or microcontrollers. The solution is the usage of an eFPGA macro on an ASIC. The design parts, which are considered to be uncertain, are mapped to the eFPGA, whereas the stable design parts are implemented in ASIC technology. Another problem is the distribution of the updates. Currently they are installed on-site by personnel or they could be installed remotely over dedicated channels. These solutions are not desirable because of the high cost of extra personnel or hardware required by these solutions. A third solution is the in-band download within the communication signal, which allows transmission via existing data paths. This way the reconfiguration packets can be created at a central location and broadcasted to the whole network.

A demonstrator which shows the feasibility of this approach has been developed. Ethernet was a chosen for the proof-of-concept demonstrator as it enables the usage of standard devices and is scalable down to a reasonable size. Once an Ethernet packet is received at the input of the SoC, it is first processed by a packet filter. It detects the reconfiguration packets addressed to this node and extracts it from the regular data stream. It is sent to the reconfiguration memory as well as forwarded to the regular signal processing part of the SoC

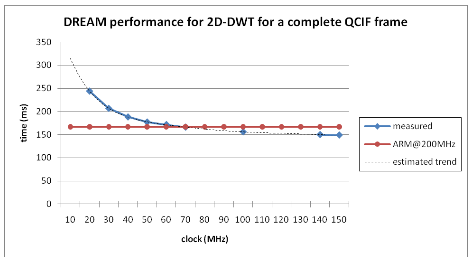


Fig. 10: DREAM performance at different clock speeds

to enable broadcasts. As soon as the reconfiguration data is complete the reconfiguration controller initiates and controls the update of the eFPGA.

7.3 Film noise grain removal application

After scanning analog films in order to digitize them, the generated digital data contains a certain kind of noise which is caused by the grain of the analog film material. The algorithm which was developed for removal of this film grain combines typical picture processing algorithms with very different processing and data transport requirements.

The main blocks of the application are motion estimation, motion compensation, and a three dimensional hierarchical wavelet transformation (DWT). Analysis of these blocks showed that motion estimation and motion compensation can be processed very efficiently on the XPP while the DWT is well suited for the DREAM. Dynamic reconfiguration of the HREs is necessary since the number and the complexity of the blocks is too high to be mapped at the same time.

The 2D-DWT is a major functional block of the DWT. Fig. 10 shows execution time over clock speed for the DREAM and shows that at ≈70 MHz DREAM reaches the speed of a 200 MHz ARM. The processing time of the film grain removal application running is approximately 600ms per frame, of which about 80ms are required for configuration. An analysis of this result shows that the main bottlenecks of the current implementation are the internal and external data transfers.

The remarkable flexibility of the MORPHEUS processing platform allows several options to improve the performance of the application. Currently, several features of the NoC are only partly used or not used at all. Additionally, [23] shows that impressive improvements in external memory bandwidth can be achieved if an advanced DDR-SDRAM controller [24] is integrated into the MORPHEUS platform architecture. Regarding the development effort, the implementation time of the film grain removal algorithm on the MORPHEUS platform with the then still experimental toolset was approximately the same as the implementation time of a current FPGA platform.

7.4 Homeland security

The goal of this test case is to demonstrate the capability of the MORPHEUS approach to address in a cost effective way applications based on intelligent cameras, i.e. systems

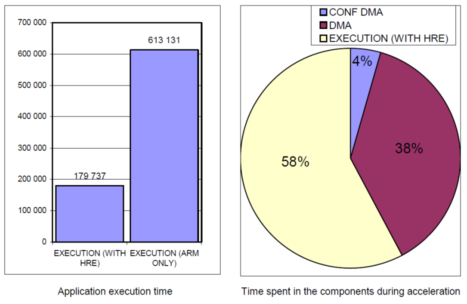


Fig. 11: Execution time of the intelligent camera application

able to automatically extract relevant information from a flow of images. A significant part of hardware requirements comes from the number crunching functions such as image enhancement, contour extraction, segmentation, objects recognition and motion detection.

The application is implemented as a sequence of operators described in C-code to enable reuse. The SPEAR tool is used to build automatically the interfaces of each operator. Writing operators and synthesis of an application is a very automated task. The toolset provides high productivity and high reuse capabilities without in-depth knowledge of the hardware. Fig. 11 indicates a speed-up of 3.4 between ARM only and ARM coupled with DREAM. It also shows that DMA communications take about the same time as the HRE execution. In a case where the DMA and the HRE work are pipelined we would gain about half of the total time.

8. Conclusion

In this paper, we gave a detailed account of aspects and outcomes of the MORPHEUS project. A modular heterogeneous, dynamically reconfigurable SoC architecture has been studied and developed. A design flow is associated to this architecture, with aims to improve the programming productivity of the platform and to shorten the development times of applications. This whole hw/sw approach for a dynamically reconfigurable platform constitutes an innovative solution for embedded computing. The feasibility and the relevance of this approach have been demonstrated through case studies. Their deployment on the chip prototype has shown good results in term of power consumption and performances of the platform. The modularity is a key advantage to easily change the architecture reconfigurable engines and adapt the toolset accordingly.

Acknowledgements

The authors would like to thank all the partners of the project consortium. This research was partially funded by the European Community's Sixth Framework Program.

References

[1] N. Voros, A. Rosti, and M. Huebner, Eds., *Dynamic System Reconfiguration in Heterogeneous Platforms: The MORPHEUS Approach*, ser. Lecture Notes in Electrical Engineering. Springer, jun 2009, vol. 40.

[2] M. Paganini, A. Div, and G. STMicroelectronics, "Nomadik®: A Mobile Multimedia Application Processor Platform," in *Design Automation Conference, 2007. ASP-DAC'07. Asia and South Pacific*, 2007, pp. 749–750.

[3] H. Mair *et al.*, "A 65-nm mobile multimedia applications processor with an adaptive power management scheme to compensate for variations," in *VLSI Circuits, 2007 IEEE Symposium on*, Jun. 2007, pp. 224 –225.

[4] L. Clark *et al.*, "An embedded 32-b microprocessor core for low-power and high-performance applications," *Solid-State Circuits, IEEE Journal of*, vol. 36, no. 11, pp. 1599 –1608, Nov. 2001.

[5] M. Wan *et al.*, "Design methodology of a low-energy reconfigurable single-chip dsp system," *The Journal of VLSI Signal Processing*, vol. 28, pp. 47–61, 2001, 10.1023/A:1008159121620. [Online]. Available: http://dx.doi.org/10.1023/A:1008159121620

[6] T. Bartic *et al.*, "Topology adaptive network-on-chip design and implementation," *Computers and Digital Techniques, IEE Proceedings -*, vol. 152, no. 4, pp. 467 – 472, Jul. 2005.

[7] G. Smit *et al.*, "Overview of the 4s project," in *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, Nov. 2005, pp. 70 –73.

[8] J. Xie *et al.*, "A reconfigurable architecture specific for the butterfly computing," in *ASIC, 2009. ASICON '09. IEEE 8th International Conference on*, Oct. 2009, pp. 83 –86.

[9] R. Gonzalez, "Xtensa: a configurable and extensible processor," *Micro, IEEE*, vol. 20, no. 2, pp. 60 –70, Mar./Apr. 2000.

[10] C. Morra *et al.*, "Felix: using rewriting-logic for generating functionally equivalent implementations," in *Field Programmable Logic and Applications, 2005. International Conference on*, Aug. 2005, pp. 25 – 30.

[11] S. Vassiliadis *et al.*, "The MOLEN polymorphic processor," *Computers, IEEE Transactions on*, vol. 53, no. 11, pp. 1363–1375, 2004.

[12] M. Coppola *et al.*, "Spidergon: a novel on-chip communication network," *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, p. 15, 2004.

[13] D. Rossi *et al.*, "A heterogeneous digital signal processor for dynamically reconfigurable computing," *Solid-State Circuits, IEEE Journal of*, vol. 45, no. 8, pp. 1615 –1626, Aug. 2010.

[14] C.-C. Cheng *et al.*, "ivisual: An intelligent visual sensor soc with 2790 fps cmos image sensor and 205 gops/w vision processor," *Solid-State Circuits, IEEE Journal of*, vol. 44, no. 1, pp. 127 –135, Jan. 2009.

[15] K. Kim *et al.*, "A 125 gops 583 mw network-on-chip based parallel processor with bio-inspired visual attention engine," *Solid-State Circuits, IEEE Journal of*, vol. 44, no. 1, pp. 136 –147, Jan. 2009.

[16] A. Abbo *et al.*, "Xetal-ii: A 107 gops, 600 mw massively parallel processor for video scene analysis," *Solid-State Circuits, IEEE Journal of*, vol. 43, no. 1, pp. 192 –201, Jan. 2008.

[17] E. Lenormand and G. Edelin, "An industrial perspective: A pragmatic high end signal processing design environment at Thales," in *Proceedings of the 3rd International Samos Workshop on Synthesis, Architectures, modelling, and Simulation*, 2003.

[18] E. M. Panainte, K. Bertels, and S. Vassiliadis, "The molen compiler for reconfigurable processors," *ACM Trans. Embed. Comput. Syst.*, vol. 6, February 2007. [Online]. Available: http://doi.acm.org/10.1145/1210268.1210274

[19] J. Cambonie *et al.*, "Compiler and System Techniques for s o c Distributed Reconfigurable Accelerators," *Computer Systems: Architectures, Modeling, and Simulation*, pp. 505–523, 2004.

[20] "Boosting software processing performance with co-processor synthesis," white paper, CriticalBlue, 2005.

[21] F. Thoma and J. Becker, "Isrc: a runtime system for heterogeneous reconfigurable architectures," in *Proceedings of the 5th International Workshop on Reconfigurable Communication-centric Systems on Chip 2010 – ReCoSoC'10*. KIT Scientific Publishing, May 2010, pp. 59–65.

[22] *PACT Software Design System XPP-IIb (PSDS XPP-IIb) - Programming Tutorial*, Version 3.2, PACT XPP Technologies, Nov. 2005.

[23] S. Whitty *et al.*, "Application-specific memory performance of a heterogeneous reconfigurable architecture," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 387–392. [Online]. Available: http://portal.acm.org/citation.cfm?id=1870926.1871020

[24] S. Whitty and R. Ernst, "A bandwidth optimized sdram controller for the morpheus reconfigurable architecture," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, Apr. 2008, pp. 1 –8.

SESSION

REGULAR SESSION - ADAPTIVE AND RECONFIGURABLE HARDWARE

Chair(s)

PROF. DAVID ANDREWS

INVITED TALKS

Design Flows and Run Time Systems For Heterogeneous Multiprocessor Systems on Programmable Chips (MPSoPCs)

David Andrews
CSCE Department
University of Arkansas
Fayetteville, Arkansas, USA
dandrews@uark.edu

***Abstract*—FPGA's have rapidly progressed through four generations and have now reached a level of maturity that allows them to be viewed as a complete multiprocessor system on programmable chip. In this invited talk, we present our approach to enable developers to guide the construction and program a heterogeneous MPSoC using standard POSIX-compatible programming abstractions. The ability to use a standard programming model is achieved by using a hardware-based microkernel to provide OS services to all heterogeneous components.**

Integration densities of FPGA's continue to follow Moore's law with emerging platform FPGAs soon to contain over 1 million LUTs. This level of transistor density will be sufficient to enable the integration of hundreds of soft programmable cores, interconnection networks, distributed memories and soft IP support components within a single FPGA. Prior to the introduction of platform FPGA's, limited integration densities prohibited the construction of a complete multiprocessor system on programmable chip. During these earlier generations, FPGA's were predominately used as accelerator co-processors for computationally intensive sections of code extracted from a single execution thread. Platform FPGA's represent an interesting enabling technology that can allow designers to switch focus from providing performance increases through tedious and platform specific accelerator point designs to constructing multiprocessor systems on programmable chip architectures that achieve performance increases through portable scalable parallel processing. As the sizes of each generation of FPGA's follows Moore's law, designers can start considering the processor instead of the transistor as least separable design unit [1]. Targeting processors instead of transistors as the lowest level design quantum promises many advantages for widening the use of FPGA technologies, lowering design costs and time to market, and increasing designer productivity. While this approach sacrifices peak performance for productivity, switching focus from low-level hardware circuit design to writing high-level parallel programs makes FPGAs accessible to the broad base of software developers who do not possess hardware design skills. Both software and hardware designers can benefit from adopting familiar software development tools and run time systems to reduce design times, and enable code portability and reuse. These should be welcome advantages within the FPGA design community that largely worked with tools and design flows that result in relatively inefficient designer productivity. This viewpoint also represents an interesting convergence with the general purpose communities movement towards multicore architectures.

Three barriers must be overcome to make this vision a reality. First, the selection of available soft IP processor components must continue to expand, and with appropriate *compiler* support. Efforts such as eMIPS and Vespa [2], [3] are increasing the selection of customizable, and vector and array programmable processors that can be integrated into a MPSoPC platform FPGA. These efforts cannot be fully exploited until associated advancements in compiler technology for heterogeneous systems are also achieved.

The second barrier is the inability to simply adopt our existing commodity operating systems, which themselves are undergoing a major paradigm shift to meet the needs of next generation homogeneous and heterogeneous multicores. The structure of our monolithic kernels were never created with scalability in mind, and integrating processors with different Instruction Set Architectures (ISA's) breaks current operating systems' abilities to provide a single uniform set of abstract services that *efficiently* execute across mixes of processor ISAs. Resolving this issue is pivotal for platform MPSoPCs as the operating system forms the foundation upon which higher level run time services and user code bases are built.

The third barrier is the lack of a standard high level programming model. Within the commercial sector several new programming frameworks are emerging specifically for heterogeneous manycores. OpenMP [4] has enjoyed some success for Symmetric Multiprocessor (SMP) global memory systems. More recently OpenCL [5] has gained popularity and is now a standard for distributed memory heterogeneous architectures. For reconfigurable systems, we are interested in extending the capabilities of the high level programming model to additionally drive the generation of a semi-custom heterogeneous MPSoPC. Third generation system level synthesis approaches made strides in generating hardware accelerators from computationally intensive portions of a high level language, such as C. However, translating a series of ALU operations into a more efficient custom circuit is orthogonal to creating an overarching multiprocessor system on chip architecture. We believe this new challenge represents a next, or forth generation of system level synthesis.

In this talk we first present the challenges that must be addressed to creating appropriate design flows and run-time systems for heterogeneous MPSoPC's. To achieve true software-like levels of productivity, the design flow and development environment for heterogeneous MPSoCs must resemble that of standard homogeneous multiprocessor systems. In our prior work called hthreads [6], [7], we have targeted the pthreads asynchronous multithreaded programming model due to it's wide acceptance and popularity. In addition to supporting standard pthreads programs, targeting pthreads allows us to support emerging models such as OpenCL that rely on an underlying pthreads implementation.

We then present our approach to enable developers to guide the construction and program a heterogeneous MPSoC using OpenCL and standard POSIX-compatible programming abstractions. We have created a new capability that allows a complete multiprocessor system on chip to be automatically created from the high level program. This relieves the designer from having to create and integrate by hand a complete set of IP components such as bus structures, I/O devices, and memory hierarchies to form the multiprocessor architecture. The architecture generated is in the form of a generic template that can be used as is, or later tuned within vendor specific tools.

Taken together, we believe a combined approach of enabling the use of high level programming models and supporting the automatic and transparent assembly of a multiprocessor system on programmable chip architecture is fundamental for the future use of dense and power FPGA's.

References

[1] S. Trimberger, "FPL 2007 Xilinx Keynote Talk - Redefining the FPGA," http://ce.et.tudelft.nl/FPL/trimbergerFPL2007.pdf, last accessed May 10, 2011.

[2] R. N. Pittman, N. L. Lynch, and A. Forin, "eMIPS, A Dynamically Extensible Processor," Microsoft Research, Tech. Rep. MSR-TR-2006-143, Oct. 2006.

[3] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors," in *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2008, pp. 61–70.

[4] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *Computing in Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[5] "OpenCL - The open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl/, last accessed May 10, 2011.

[6] D. Andrews, D. Niehaus, and P. J. Ashenden, "Programming Models for Hybrid CPU/FPGA Chips," *IEEE Computer*, vol. 37(1), pp. 118–120, 2004.

[7] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving Programming Model Abstractions For Reconfigurable Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, January 2008.

Can Run-time Reconfigurable Hardware be more Accessible?

Jim Torresen and Dirk Koch
Department of Informatics, University of Oslo, Oslo, Norway
E-mail: {jimtoer,koch}@ifi.uio.no
Web: http://www.mn.uio.no/ifi/english/research/projects/cosrecos

Abstract—*In this paper, a new project named Context Switching Reconfigurable Hardware for Communication Systems (COSRECOS) is introduced. The project started autumn 2009 and consists of applying reconfigurable hardware technology (Field Programmable Gate Arrays - FPGAs) for designing high performance run-time reconfigurable computing architectures for communication systems. The overall goal of the project is to contribute in making run-time reconfigurable systems more feasible in general. This includes introducing architectures for reducing reconfiguration time as well as undertaking tool development. Case studies by applications in network and communication systems will be a part of the project. The paper describes how we plan to address the challenge of changing hardware configurations while a system is in operation. An overview of promising initial approaches is also included.*

Keywords: Reconfigurable hardware, FPGA, Run-time reconfiguration

1. Introduction

Until the introduction of multitasking operating systems around 1985, processors would run one program at a time. The program would be uploaded at startup and be running until finished. There would be no swapping to other programs during execution of a given program. With today's multitasking operating systems, it would often be the exception not performing multitasking for software. This is in contrast to hardware which normally is static at run-time even though reconfigurable hardware is programmable at run-time. However, in this project – called *Context Switching Reconfigurable Hardware for Communication Systems (COSRECOS)*, architectures where the hardware configuration is dynamically changed (i.e. context switching) will be investigated [1].

The main contributions of the project are expected to be:

- Develop new architectures and tools to make run-time reconfiguration easier to use.
- Develop platforms for fast and reliable reconfiguration.
- Undertake case studies with focus on communication applications.

We regard that both architectures and tools would be needed to make run-time reconfigurable hardware more applicable. The project is funded by the Research Council of Norway and a number of researchers are adressing the challenges in the project. The next section gives some background information, followed by an overview of benefits and possible approaches in section III. Initial approaches are included in Section IV with Section V concluding the paper.

2. Background

Reconfigurable computing has grown to become an important and large field of research. Reconfigurable systems are designed either by using commercial Field Programmable Gate Arrays (FPGAs) or by having new FPGA-like devices developed. A survey of can be found in [2]. The most common target technology is FPGAs. In addition to fine-grained FPGA devices, several coarse-grained architectures (e.g. the PACT XPP processor array [3]) have been developed to implement reconfigurable systems. However, the COSRECOS project is targeting FPGA technology. It is substituting the configuration bitstream of the FPGA at *run-time* we think of by context switching reconfigurable hardware.

2.1 Applications

The approach of applying reconfigurable logic for data processing has been demonstrated for a number of years ago in areas such as video transmission, image recognition and various pattern-matching operations (handwriting recognition, face identification) [4]. Another area of interest is wireless systems where tremendous computational capabilities are needed to allow for high data rates in the future [5]. Automotive electronic systems are expected to gain large benefit from adaptive reconfigurable hardware [6].

A platform for network applications – called The Field-programmable Port Extender (FPX), has been implemented, see [7]. It is a generic platform with network interfaces that has been used in a wide variety of applications: route Internet packets; compress, encrypt, and buffer data; transcode motion JPEG images; and process multiple flows of video. By using FPGA hardware, rather than a microprocessor, the packet processor can perform full processing of packet payloads at Gigabit rates. The development of the FPX platform demonstrates a valuable use of FPGA technology in routers and other network equipment. The hardware of the system will evolve over time as packet processing algorithms and protocols progress, as stated in [8].

We have been conducting research on several different application areas of reconfigurable logic. An architecture for providing a fast (in the Gigabit range) network security system (called Network Intrusion Detection Systems (NIDS)) has been proposed. The detection engine (rule matching) in the open source network intrusion detection system Snort has been implemented in [9]. Further, stateful inspection is applied in NIDS in [10]. By checking the handshakes in a communication session, it provides a more advanced network checking than firewalls.

We have demonstrated the benefits of undertaking data processing in reconfigurable logic for applications like string matching [11] and image filtering [12]. An FPGA implemented processor architecture with adaptive resolution has been introduced in [13]. This allows for a variable resolution in data variables at run-time.

2.2 Architectures

For integrating several modules into an FPGA-based-system at run-time, an on-FPGA communication architecture is used to provide communication between the static part of a system and the reconfigurable modules. For simpler systems, where only one module can be exclusively placed inside a reconfigurable region (what we call *island style*), connection primitives called busmacros have been proposed by Xilinx that provide simple point-to-point connections between a static system and a reconfigurable region [14]. The vendor Xilinx is currently using dummy connection logic (called proxy logic) for interfacing reconfigurable modules [15] in their recent partial design flow. This approach is actually not a real communication architecture as it does not result in a special structure for the routing between a static system and reconfigurable modules. As a consequence, modules are bound to particular placement positions and cannot be relocated in systems that provide multiple reconfigurable regions.

A more advanced architecture has been proposed in [16]. There, a bus architecture allows to connect multiple modules in a combined shared region in a one-dimensional manner (what we call *slot style*). An on-FPGA communication architecture that considers streaming data connections between modules that can be placed in two dimensions (*grid style*) has been proposed in [17] for signal processing applications.

The ReCoBus project [18] addresses high performance, resource efficiency and flexibility for on-FPGA communication at the same time. It provides a backplane bus (called ReCoBus) for slave and master communication as well as channels for data streaming among modules that are called I/O bars. The original project was limited to Xilinx Virtex-II and Spartan-3 FPGAs but the tool has recently been extended to support the latest FPGA devices as well (see section 4).

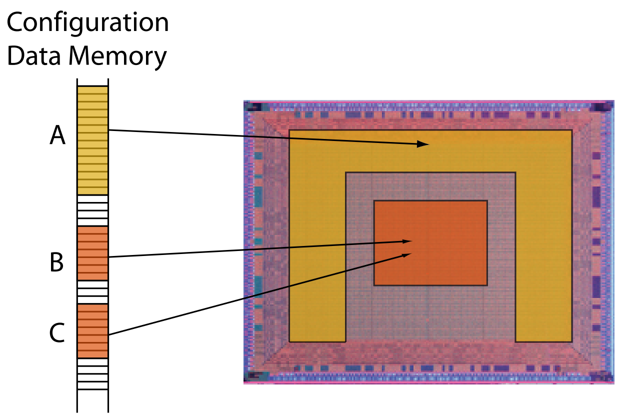


Fig. 1
ILLUSTRATION OF A RUN-TIME RECONFIGURATION OF FPGA.

3. FPGA Based Architectures

This section includes a brief overview of how FPGAs can be applied in adaptive systems [19]. We can distinguish between three different degrees of FPGA reconfiguration providing configurable computing:

- **Static:** The configuration within the FPGA is the same throughout the lifetime of the system. This means no adaptivity at run-time.
- **Upgrade:** The configuration is changed from time to time for bug fixes or functional upgrades. This represents rare reconfiguration.
- **Run-time:** A set of configurations are available which the FPGA switch between at run-time. This could provide several benefits as described below.

Most applications are implemented by applying the *static* approach – i.e. no reconfiguration. However, *upgrading* of systems have recently become more common. This allows the configuration to be upgraded when bugs are found *or* when the functionality of the system is to be changed. In the future, automatic dynamic products will probably arrive. These could *autonomously* upgrade the hardware as the environment (or data) changes or when bugs are detected in the system. One promising approach based on this idea is evolvable hardware [20].

The objectives for implementing *run-time* reconfigurable systems are:

- Space/cost/power reduction
- Speeding up computation
- Substituting data/patterns in hardware realized search filters

If not all functions in a system are needed at the same time (i.e. functions are mutually exclusive), we can substitute a part of the configuration at run-time as seen in Figure 1. Function A contains the parts of the system that always need to be present – i.e. the static part of the system.

However, part B and C are not needed concurrently and can be assigned to the *same* resources (location) in the FPGA. An example of such an application can be a multi-functional handheld device with e.g. mobile phone, MP3 player, radio and camera. For most purposes, a user would normally not apply more than one of these functions at a time. Thus, instead of having custom hardware for each function, it could be efficient having a reconfigurable system where only the *active* function is configured. This would allow for a smaller hardware device which leads to reduced cost and for some systems reduced power consumption. Such benefits are important in a competitive market.

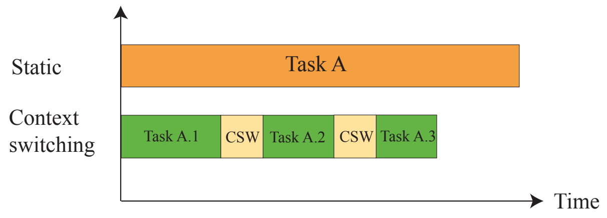


Fig. 2
ILLUSTRATION OF A RUN-TIME RECONFIGURABLE FPGA COMPARED TO A STATIC FPGA.

The application area for run-time reconfiguration for computational *speedup* is depicted in Figure 2. Swapping between successive configurations can give a hardware system a considerable throughput compared to having a general *static* FPGA configuration. If a task A can be partitioned into a set of separate sub-tasks (A.1, A.2 and A.3 in the example in the figure) to be executed one after the other, an FPGA configuration can be designed for each of them. Thus, each configuration is optimized for one *part* of the computation. During run-time, context switching (CSW) is undertaken and the total execution time for the task in the given example is reduced.

For instance, when considering a secured SSL connection, the asymmetric key exchange can be a first sub-task. The result of this first step is a session key that can then be used by the second sub-task that encodes the entire data using a symmetric cipher. By spending more resources on each particular sub- task, a higher level of parallelism is obtained, and hence, a speed-up might be achieved. The context switching time would have to be short compared to the computation time, to reduce the overhead of switching between the different configurations.

Since commercially available FPGAs do not yet provide configuration switching in one or a few clock cycles, download time is often the main obstacle against effective run-time reconfiguration. There has been undertaken some work based on run-time reconfiguration of FPGAs. The main experience seems to be that FPGAs are requiring a (too) long reconfiguration time, and we regard this as the key challenge when designing context switching systems. The granularity of the sub-tasks is critical and the amount of computation to be undertaken in each sub-task should be sufficient large to pay of the configuration overhead [21], [22]. That is, the size of the sub-tasks should be chosen so that the total processing time including context switching is minimized.

Many devices require the *complete* configuration bitstream to be downloaded in one operation. The download time then increases with the size of the device. The challenge with long download time is further addressed in the next section.

3.1 Approaches to Reducing Reconfiguration Time

The project will focus on research for developing new architectures that can reduce the problems of applying commercial FPGA technology to run-time reconfigurable computing. Rather than focusing on FPGA only as an ASIC (Application-specific integrated circuit) like accelerator for speeding up computation – as in many research projects, we will also emphasize on switching configurations at run-time. That is, replacing parts of the user logic inside the FPGA while other parts operate uninterrupted. This could also comprise storing and recovering internal states of hardware modules. By this approach, we would like to have a focus on both reduced cost and power consumption as well as additional computational speedup. However, some systems would have to be limited to one of these priorities. By developing new architectures and algorithms, we will try to come up with systems making run-time reconfigurable hardware more usable.

Challenges of run-time reconfiguration in FPGA to be addressed in the project are as follows:

- Reducing the long time required for reconfiguration
- Avoiding the system from being inactive during reconfiguration (safe and robust reconfiguration)
- Interfacing between modules belonging to different configurations
- Predictability (reliability and testability) of system operation

As introduced earlier, the main problem with switching configurations is the long reconfiguration time. Overcoming this would be one of the main objectives in the project. There is a number of different approaches available (several of them will be explored in our project):

3.1.1 Smaller Devices

Since the full reconfiguration time is less for smaller devices, reconfiguration time can be reduced by applying smaller devices. Moreover, by applying context switching, we may be able to implement a full system in a smaller device with the benefit of reduced cost and power consumption. The drawback would be that the system would have to be inactive during reconfiguration.

3.1.2 Bitstream Prefetching

An approach consisiting of hiding of the reconfiguration time by prefetching the bitstream [23].

3.1.3 Bitstream Compression

Bitstream compression would be useful for reducing memory bottlenecks.

3.1.4 Hyperreconfiguration

Faster reconfiguration can be achieved by applying a two step reconfiguration [22], [24]. The first step is used to find the reconfiguration potential of the architecture (hyperreconfiguration) followed by a second step where reconfiguration actually takes place.

3.1.5 Tuneable LUTs

Tuneable LUTs consists of having two (or more) netlists (the configurations) merged into one combined netlist. The basic idea of the tunable LUTs is that you multiplex between the two functions, but instead of having a physical multiplexer, the LUT-content is changed.

3.1.6 Overclocking

Clocking the bitstream interface at a higher speed than the maximum clock frequency specified by the FPGA vendor [25].

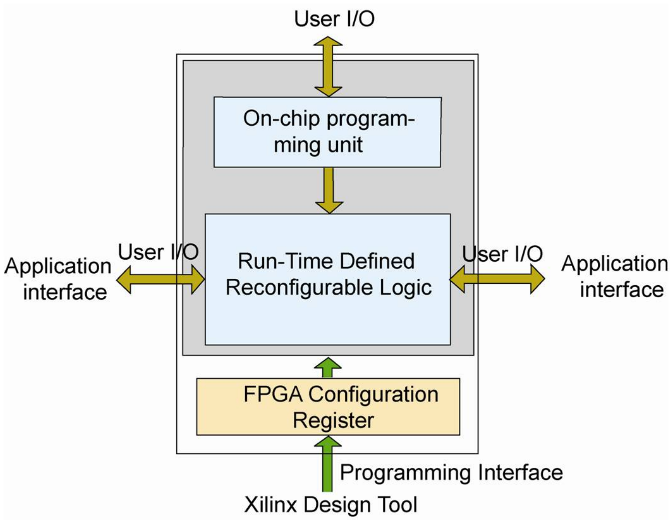


Fig. 3
VIRTUAL FPGA.

3.1.7 Virtual FPGA

Virtual FPGA is based on designing a multi-context "virtual" FPGA inside an ordinary FPGA [26] – see Figure 3. We have earlier introduced an architecture for context switching based on this idea that has been published in [27], [28]. In these papers, we report about our design of an architecture for switching between 16 different configurations in a *single* clock cycle. Such a system would never achieve as high clock frequency as a leading edge processor. However, by applying massive parallel processing, the execution time can still be less [13].

We have published a number of papers where virtual FPGA is combined with evolvable hardware [19], [29], [30], [31], [32]. The developed architectures also include a soft (MicroBlaze) or hard (PowerPC) processor core. Even though a fast processing can be achieved, the context switching architecture requires much reconfigurable resources (in that way, this architecture is prioritizing speed before cost and power consumption). Due to the fine grain structure of FPGAs, we had to focus on reconfiguring only a limited number of parameters in the designed architectures to reduce the hardware overhead. There is still a large potential for improving these systems which could be addressed in this project.

3.1.8 Partial Reconfiguration

As FPGA devices are getting bigger, the configuration bitstream becomes longer and programming time increases. Thus, run-time reconfigurable designs would benefit from having only a limited part of the FPGA being context switched by *partial reconfiguration*. This feature is available in some FPGAs where a selected number of neighboring columns are programmed. This requires detailed considerations for having no interruption at context switching [33].

Another challenge is to limit the inter spatial partition data transfer. That is, efficient communication between context switched tasks. While the first FPGAs offering partial reconfiguration required *complete columns* of the device being programmed, the more recent ones – including Xilinx Virtex-4/5, require only a *part* of each column being programmed. This makes interfacing between tasks and having uninterrupted operation easier since some rows can be used for permanent configurations. The smallest Virtex-5 device (LX30) consists of 4 rows while the largest (LX330) consists of 12 rows. Further, there have been introduced tools like PlanAhead that make partial reconfiguration easier.

It is possible to reconfigure the Virtex devices *internally* using the Internal Configuration Access Port (ICAP). This will be applied in this project where also research will be undertaken on efficient data routing and data storing between context switching tasks [34]. We have already undertaken various successful work with partial reconfiguration including change of look-up table content [35] and internal reconfiguration with ICAP by use of PlanAhead [36].

3.2 Platforms and Tools

The main barrier for applying partial run-time reconfiguration in industrial applications seems to be the lack in tools and methodologies. With the introduction of PlanAhead, the vendor Xilinx has enormously simplified the design flow for implementing reconfigurable systems but there are several open issues that have not yet been solved. For example, the island style reconfiguration scheme comprises a large waste of logic as different modules could have different resource requirements, and because modules cannot share the same region, even if their resource requirements would allow this. Furthermore, as the routing between the static system and the dynamic modules is not constrained to physical wire resources, the routing will in general differ on each implementation of the static system. Consequently, in the wake of changes in the static part, rerouting is required for all permutations of placement position and module instance (i.e., a partial module bitstream cannot be written to different positions of the FPGA). Moreover, there exist no possibility to simulate the reconfiguration of a system.

The work at the moment on reconfigurable systems are usually based on problem specific coding. Thus, a goal of this project is to come up with a tool and some general platforms that could make context switching systems more accessible for a larger number of users.

If the context switching is *not* deterministic, an operating system may be needed to schedule hardware tasks [37]. However, for many of the applications, it would be possible with a deterministic context switching. This will be our first approach since online scheduling of tasks including control of reconfigurable logic fragmentation would be much more difficult. However, to have more general computing platforms, this would become more necessary. We believe a general computing platform will make reconfigurable technology more accessible than it is today, and make it into a viable complement to processor technology. The success of multitasking in software is probably much because of the introduction of widely used operating systems like Windows and Linux. If an equivalent could be found across FPGA vendor technologies by e.g. virtual FPGA, it would probably be an important step towards more widely use of run-time reconfigurable hardware.

A part of the research will be on analyzing HW/SW partitioning and how this can be undertaken in the most efficient way. Much research related to communication technology is *either* related to implementing software *or* hardware. However, few projects are concerned about the *integration* of application software, low level software and hardware. In the industry on the other hand, much focus is given to this integration. Thus, we would in this project like to address how hardware should be designed to most effectively execute the software to be implemented. More details about the project can be found in [38].

4. Initial Approaches

The initial work which will be described below has focused on various aspects like reducing both the reconfiguration time as well as the hardware overhead for run-time reconfiguration. Further, efficient routing architectures have been introduced.

Many have been regarding the slow reconfiguration speed as the main obstacle against runtime reconfigurable hardware. Thus, we have started the project with addressing how to reduce the reconfiguration time. This is partly achieved by applying the latest FPGA technology providing higher density and speed.

However, the trend for the latest devices tend towards more logic and less routing resources that are also more irregularly arranged compared to in previous architectures. Thus, implementing an on-FPGA communication architecture has become a challenge. Thus, in [39], an on-FPGA communication architecture that is especially tailored to Xilinx Virtex-5 FPGAs is introduced. The architecture – as seen in Figure 4, contains a two-dimensional circuit switching network using dedicated I/O bars. Multiplexers in the static part perform the vertical routing while I/O bars carry out the routing in horizontal direction. This allowes for modules being integrated in a two-dimensional grid proving a data throughput of up to several GB/s between reconfigurable modules.

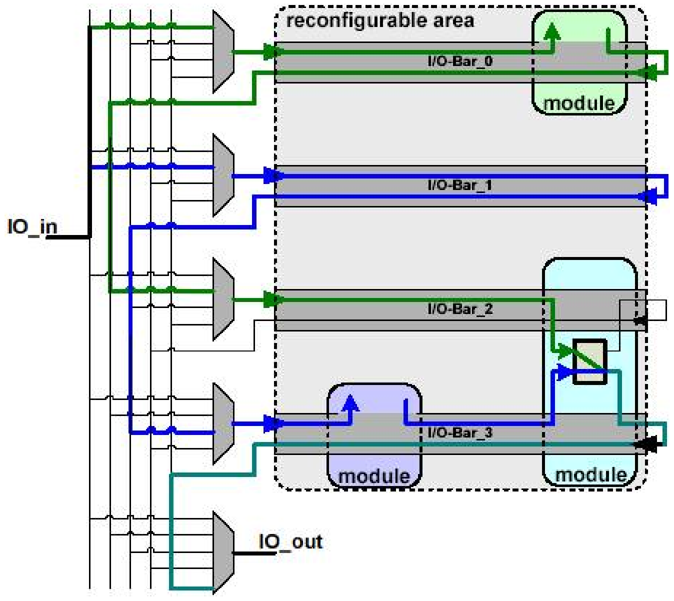


Fig. 4
TWO-DIMENSIONAL CIRCUIT SWITCHING NETWORK [39].

It will be important that reconfiguration of an FPGA does not damage the device by introducing short-circuits. Thus, we have undertaken experiments to test how vulnerable FPGAs are to short-circuits [40]. Despite the absence of tristate buffers, short-circuits in the routing fabric of recent Xilinx FPGAs may still occur. In these devices, the multiplexer based implementation of switch matrices allows for connecting more than one driver to a wire at the same time. Although not damaging the device, it was shown by a test setup that increased current consumption occured as a result of the problem. To cope with the problem, a bitstream scanner algorithm has been introduced to detect possible long term short-circuits. Furthermore, it was proven that blanking a reconfigurable region before writing a new configuration removes the occurrence of short-circuits during the reconfiguration process.

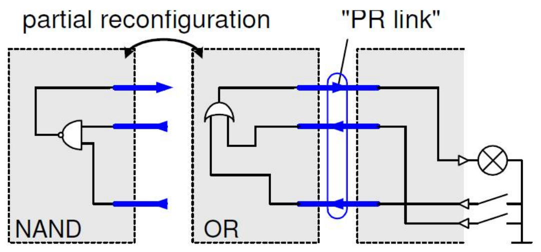


Fig. 5
PR LINK APPROACH [41].

One possible benefit of including reconfiguration is to save logic resources. However, an overhead of additional resources is often introduced for integrating reconfigurable parts with the static system. By constraining the communication resources between the static system and the partial regions, we have proposed an architecture with no logic overhead [41]. The concept has been demonstrated for a reconfigurable instruction set for a processor. It is based on having all communication with modules located in partial regions bound to dedicated links – partial reconfiguration (PR) links. However, there do not exist any constraints on routing resources in the Xilinx vendor tools but it is possible to implement *macros* to restrict the routing.

The project has a focus on applying the latest technology in the experiments and in [43] we demonstrate a system for partial run-time reconfiguration on Spartan-6 series FPGAs. Further, in [42] a two-dimensional obstacle free online-routing for run-time reconfigurable FPGA-based systems is introduced. In that work, we partitioned the FPGA routing resources into distinct sets for implementing the static system, the reconfigurable modules, and the communication between them. By carefully selecting the resources for the latter set, a circuit switching network has been directly implemented within the routing fabric (i.e., no further logic resources are used to switch the routing), such that modules can be placed freely in a two-dimensional tiled reconfigurable area.

As the size of the FPGAs has become bigger, the time needed to compile a design also increases. Having a tool to allow for physical implementation of a *part* of the device at a time would reduce the development time. Thus, a component-based based design flow is introduced in [44]. This allows for modules being directly plugged together on an FPGA without the need to run the logic synthesis or place & route for the complete system. This would also make it easier for users adding their own design to an existing design without deep knowlewdge in the working of the other parts of the system. However, such a design flow would require that the components (IP cores) are encapsulated in bounding boxes with only limited routing to other parts of the design, see Figure 6. This has been addressed by relaxing the constraint that the routing of a module has to be strictly bound into its assigned bounding box. In [44], we demonstrated that wire resources outside of such a bounding box can be used without interfering other parts of the system and while still being able to relocate modules to different positions on the FPGA.

5. Conclusion

This paper has described a new project focusing on context switching reconfigurable hardware. It will target to make such technology more applicable by introducing software tools as well as hardware architectures. Challenges include addressing reconfiguration time and making the context switching robust. The paper also introduced some of the promising results of the project so far.

Acknowledgment

This work is supported by the Norwegian Research Council funded project Context Switching Reconfigurable Hardware for Communication Systems (COSRECOS), under grant 191156V30.

References

[1] Jim Torresen and Dirk Koch. A new project to address run-time reconfigurable hardware systems. In Peter M. Athanas, Jürgen Becker, Jürgen Teich, and Ingrid Verbauwhede, editors, *Dynamically Reconfigurable Architectures*, number 10281 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[2] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proc. of Int. Conference on Design Automation and Testing in Europe - and Exhibit (DATE)*, 2000.

[3] Volker Baumgarte, F. May, Armin Nückel, Martin Vorbach, and Markus Weinhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 64–70, Las Vegas, June 2001.

[4] J. Villasenor and W.H. Mangione-Smith. Configurable computing. *Scientific American*, (6), 1997.

[5] J.M. Rabaey. Silicon platforms for the next generation wireless systems - What role does reconfigurable hardware play?. In R.W. Hartenstein et al., editors, *10th International Conference on Field Programmable Logic and Applications (FPL-2000)*, Lecture Notes in Computer Science, vol. 1896, pages 277–285. Springer-Verlag, 2000.

[6] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka. Dynamic and partial FPGA exploitation. *Proceedings of the IEEE*, 95(2):438–452, Feb 2007.

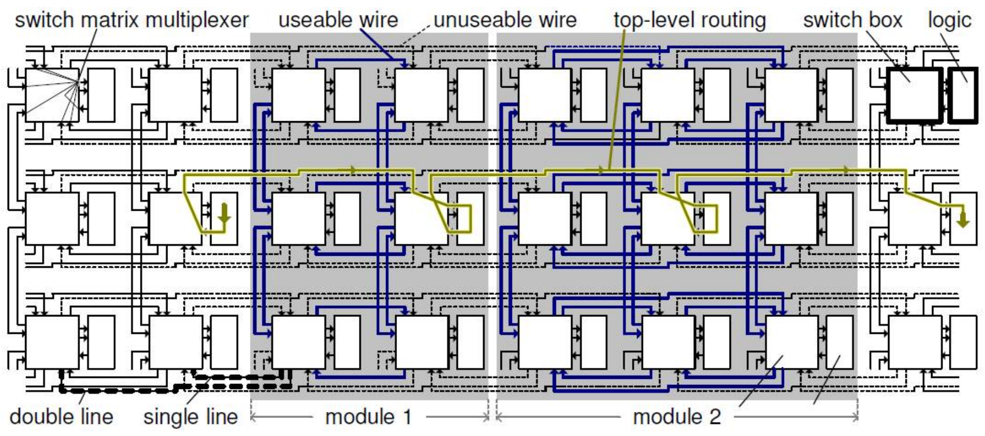


Fig. 6

Simplified FPGA architecture hosting two encapsulated modules. Because of the strict encapsulation of modules into bounding boxes, wires leaving either the static part or a particular module remain unusable. Only the reserved top-level path is allowed to cross module bounding box borders [42].

[7] J. Lockwood. *The Field-programmable Port Extender (FPX), http://www.arl.wustl.edu/projects/fpx*, 2010.

[8] J.W. Lockwood. Evolvable internet hardware platforms. In *Proc. of the Third NASA/DoD Workshop on Evolvable Hardware*, pages 271–279, 2001.

[9] S. Li, J. Torresen, and O. Soraasen. Exploiting reconfigurable hardware for network security. In *11th Annual IEEE Symp. on Field Programmable Custom Computing Machines (FCCM'03)*. IEEE, 2003.

[10] S. Li, J. Torresen, and O. Soraasen. Exploiting stateful inspection of network security in reconfigurable hardware. In *Field-Programmable Logic and Applications: 13th International Conference on Field Programmable Logic and Applications (FPL-2003)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[11] G. Nilsen, J. Torresen, and O. Soraasen. A variable word-width content addressable memory for fast string matching. In *Proc. of 22nd Norchip Conference*, pages 214–217. IEEE, 2004.

[12] J. Torresen, J. W. Bakke, and L. Sekanina. Efficient image filtering and information reduction in reconfigurable logic. In *Proc. of 22nd Norchip Conference*, pages 63–66. IEEE, 2004.

[13] J. Torresen and J. Jakobsen. An FPGA implemented processor architecture with adaptive resolution. In *Proc. of 1st NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2006)*. IEEE, 2006.

[14] Patrick Lysaght, Brandon Blodget, Jeff Mason, Jay Young, and Brendan Bridgford. Invited Paper: Enhanced Architecture, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *Proceedings of the 16th International Conference on Field Programmable Logic and Application (FPL)*, pages 1–6, Aug 2006.

[15] Xilinx Inc. Partial Reconfiguration User Guide, December 2009. Rel 11.4.

[16] Jens Hagemeyer, Boris Kettelhoit, Markus Koester, and Mario Porrmann. Design of Homogeneous Communication Infrastructures for Partially Reconfigurable FPGAs. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, USA, Jun 2007.

[17] P. Sedcole et al. Modular dynamic reconfiguration in Virtex FPGAs. *IEE Proceedings - Computers and Digital Techniques*, 153(3):157–164, May 2006.

[18] D. Koch, C. Beckhoff, and J. Teich. ReCoBus-builder - a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs. In *Proceedings of Int. Conf. on Field-Programmable Logic and Applications*, Heidelberg, Germany, Sep. 2008, pages 119 – 124.

[19] J. Torresen and K. Glette. Improving flexibility in on-line evolvable systems by reconfigurable computing. In *Evolvable Systems: From Biology to Hardware. Seventh International Conference, ICES'07*, volume 4684 of *Lecture Notes in Computer Science*, pages 391–402. Springer-Verlag, 2007.

[20] J. Torresen. An evolvable hardware tutorial. In *Proc. of the 14th International Conference on Field Programmable Logic and Applications (FPL 2004)*, pages 821–830. Springer Verlag, LNCS 3203, 2004.

[21] Y. Agarwal et al. Solving fracture mechanics problems using reconfigurable computing. In *Proc. of Int. Conf. on Reconfigurable Computing and FPGAs, ReConFig'04*, 2004.

[22] S. Lange and M. Middendorf. Hyperreconfigurable architectures and the partition into hypercontexts problem. *Journal of Parallel and Distributed Computing*, 65(6):743–754, 2005.

[23] J. Torresen. Reconfigurable logic applied for designing adaptive hardware systems. In *Proc. of the International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet (SSGRR'2002W)*. Scuola Superiore G. Reiss Romoli, 2002.

[24] S. Lange and M. Middendorf. Hyperreconfigurable architectures for fast runtime reconfiguration. In *Proc. of 2004 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM04), Napa Valley, USA*, 2004.

[25] C. Claus, R. Ahmed, F. Altenried, and W. Stechele. Towards rapid dynamic partial reconfiguration in video-based driver assistance systems. In *Reconfigurable Computing: Architectures,Tools and Applications*, volume 5992 of *Lecture Notes in Computer Science*, pages 55–67. Springer Berlin / Heidelberg, 2010.

[26] L. Sekanina and R. Ruzicka. Design of the special fast reconfigurable chip using common FPGA. In *Proc. of Design and Diagnostics of Electronic Circuits and Systems - IEEE DDECS'2000*, pages 161–168, 2000.

[27] J. Torresen and K.A. Vinger. High performance computing by context switching reconfigurable logic. In *Proc. of the 16th European Simulation Multiconference*, pages 207–210. SCS Europe, June 2002.

[28] K. A. Vinger and J. Torresen. Implementing evolution of FIR-filters efficiently in an FPGA. In *Proc. of the 2003 NASA/DoD Workshop on Evolvable Hardware*, 2003.

[29] K. Glette, J. Torresen, M. Yasunaga, and Y. Yamaguchi. A flexible on-chip evolution system implemented on a Xilinx Virtex-II Pro device. In *Proc. of Evolvable Systems: From Biology to Hardware. Sixth International Conference, ICES 2005. Volume 3637 of Lecture Notes in Computer Science*, pages 66–75. Springer-Verlag, 2005.

[30] K. Glette, J. Torresen, M. Yasunaga, and Y. Yamaguchi. On-chip evolution using a soft processor core applied to image recognition. In *Proc. of the First NASA /ESA Conference on Adaptive Hardware and Systems (AHS 2006)*, pages 373–380. IEEE Computer Society, 2006.

[31] K. Glette, J. Torresen, and M. Yasunaga. An online EHW pattern recognition system applied to face image recognition. In M. Giacobini et al., editor, *Applications of Evolutionary Computing, EvoWorkshops2007: EvoCOMNET, EvoFIN, EvoIASP, EvoInteraction, EvoMUSART, EvoSTOC, EvoTransLog*, volume 4448 of *Lecture Notes in Computer Science*, pages 271–280. Springer-Verlag, 2007.

[32] K. Glette, J. Torresen, and M. Yasunaga. Online evolution for a high-speed image recognition system implemented on a Virtex-II Pro FPGA. In *The Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*. IEEE, 2007.

[33] D. Lim and M. Peattie. *Two flows for partial reconfiguration: module based or small bit manipulation, Application Note 290*. Xilinx, 2003.

[34] N.P. Sedcole, P.Y.K. Cheung, G.A. Constantinides, and W. Luk. On-chip communication in run-time assembled reconfigurable systems. In *Proc. of IC-SAMOS*. IEEE, 2006.

[35] H. Kawai, M. Yasunaga, K. Glette, and J. Torresen. An adaptive pattern recognition hardware with dynamic partial reconfiguration. 2008. In preparation.

[36] G.A. Senland. *Design of an Architecture for Evolvable Hardware based on Internal Reconfiguration of an FPGA (in Norwegian)*. University of Oslo, 2008. Master thesis.

[37] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. on Computers*, 53(11):1393–1407, Nov 2004.

[38] D. Koch and J. Torrresen. Advances in component-based system design and partial run-time reconfiguration. In *Proc. of Dagstuhl Seminar 10281*, 2010.

[39] Dirk Koch, Christian Beckhoff, and Torresen Jim. Fine-grained Partial Runtime Reconfiguration on Virtex-5 FPGAs. In *18th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010)*, pages 69–72. IEEE Computer Society, May 2010.

[40] Christian Beckhoff, Dirk Koch, and Jim Torresen. Short-Circuits on FPGAs caused by Partial Runtime Reconfiguration. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 596–601, Milan, Italy, August 2010.

[41] Dirk Koch, Christian Beckhoff, and Jim Torresen. Zero Logic Overhead Integration of Partially Reconfigurable Modules. In *23rd Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 103–108. ACM, September 2010.

[42] Dirk Koch, Christian Beckhoff, and Jim Torresen. Obstacle-free Two-dimensional Online-Routing for Run-time Reconfigurable FPGA-based Systems. In *Proceedings of International Conference on Field-Programmable Technology (ICFPT'10)*, Beijing, China, 2010. IEEE. to appear.

[43] Dirk Koch, Christian Beckhoff, and Jim Torresen. Demo Paper: Advanced Partial Run-time Reconfiguration on Spartan-6 FPGAs. In *Proceedings of International Conference on Field-Programmable Technology (ICFPT'10)*, Beijing, China, 2010. IEEE. to appear.

[44] Dirk Koch and Jim Torresen. Routing Optimizations for Component-based System Design and Partial Run-time Reconfiguration on FPGAs. In *Proceedings of International Conference on Field-Programmable Technology (ICFPT'10)*, Beijing, China, 2010. IEEE. to appear.

SAHA: A Self-Adaptive Hardware-Software System Architecture for Ubiquitous Computing Applications

Pao-Ann Hsiung and Chun-Hsian Huang
Department of Computer Science and Information Engineering,
National Chung Cheng University, Chiayi, Taiwan−621, ROC
Email: {pahsiung,huang}@cs.ccu.edu.tw

Abstract— *Ubiquitous computing enables services and devices to be dynamically adapted to changing conditions. System adaptivity becomes a key requirement in providing possibly better system performance. Most existing ubiquitous computing systems either only support software adaptation or limit the usage of reconfigurable hardware designs as conventional hardware devices. As a result, system adaptation and performance are quite restricted. To provide a more robust system adaptation, we propose a self-adaptive hardware-software system architecture (SAHA) that consists of service suppliers, hardware adapter, system manager, observer, and reconfigurable hardware architecture. SAHA supports both hardware preemption and hardware virtualization within a complete self-aware system adaptation mechanism such that the utilization of system resources is enhanced and better performance is provided for ubiquitous computing applications. Experiments with a ubiquitous computing service for information encryption demonstrate that SAHA can reduce the turnaround time by at least* 22.04% *of that required by using the conventional method.*

Keywords: Dynamic Partial Reconfiguration, Hardware Virtualization, Hardware Preemption, Ubiquitous Computing

1. Introduction

Ubiquitous computing provides human being with a more perfect life. Under natural interaction, ubiquitous computing enables information processing to be thoroughly integrated into everyday living, without being aware that it is doing so. In such a ubiquitous computing environment, not only services and devices can be dynamically adapted to changing environments, but contexts and preferences of users can be also switched seamlessly.

To support dynamically changing and unpredictable ubiquitous computing applications, system adaptivity becomes a key requirement in providing better system performance. Most existing dynamic adaptive approaches to ubiquitous computing [1]–[3] focus on adapting software services and applications, while hardware devices only passively support the changing software applications. In such a hardware architecture, hardware functions can be only configured once at design-time so that new hardware functions cannot be reconfigured at run-time, which leads to the inefficient use of hardware resources. Further, this restricts the adaptivity of ubiquitous computing system to changing environment conditions, and system performance is thus restricted by the underlying hardware. Though several research works [4], [5] have adopted the dynamic reconfiguration capability of FPGA [6] to support hardware adaptation, the hardware functions in these approaches [4], [5] are still managed as conventional hardware devices. This means that once the device node of a hardware function has been opened by a software application, the hardware function cannot be accessed by other software applications at the same time, though it is not always accessed by the software application that opens its device node. As a result, the utilization of reconfigurable hardware resources and the enhancement of system performance are still limited. To be able to not only adapt on-demand hardware/software functionalities but also provide better *Quality-of-Service* (QoS), ubiquitous computing applications require a more robust and effective infrastructure that can autonomously adapt its software and hardware functions to meet the dynamic requirements of various environmental situations.

In this work, we try to solve the above adaptivity and performance problems in the current existing ubiquitous computing systems [1]–[5] by proposing a *Self-Adaptive Hardware-software system Architecture* (SAHA). Figure 1 gives an example for illustrating the practicability of the proposed SAHA. In this ubiquitous computing environment, each family member can use his/her own electronic devices to monitor the house, watch the live television programs, and listen to the voice messages recorded in the home telephone via SAHA. To ensure the QoS and the information security, all data will be compressed, encrypted, or both in SAHA and then transferred to family members. Computing-intensive functions, such as *Discrete Cosine Transform* (DCT), cryptographic, and hash functions are implemented as hardware devices to provide better performance. Further, SAHA can adapt on-demand its hardware and software functionalities to different service requirements, based on the network services and electronic products used by family members.

Figure 1 shows an ideal blueprint to apply SAHA to ubiquitous computing environments. However, it must solve several issues related to ubiquitous computing, including 1) how to make hardware functionalities adaptable? 2) how to

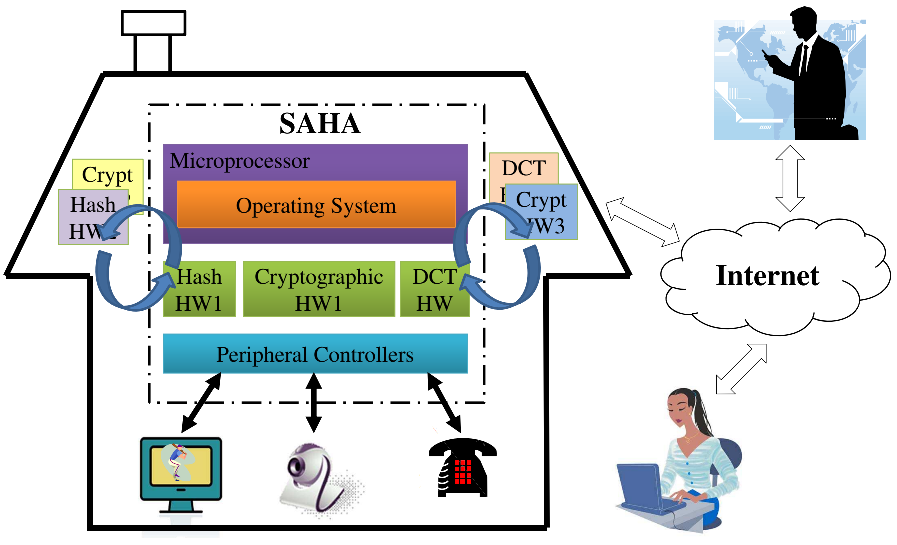


Fig. 1: Ubiquitous Computing Environment

make hardware adaptation more efficient? 3) how can the utilization of hardware devices be maximized to serve more users? 4) when must hardware functionalities be adapted to different user requirements?

To make hardware functionalities adaptable, the dynamic partial reconfiguration technology [6], [7] of FPGA devices is adopted in SAHA to perform computing-intensive functions. Hardware functions can be dynamically configured on-demand into the FPGA device, thus making the utilization of hardware logics in a FPGA device more efficient. To make hardware adaptation efficient, SAHA integrates our previously proposed hardware preemption technology [8], as a result of which high-priority hardware functions can interrupt low-priority hardware functions that have been configured in the FPGA. Thus, the utilization of hardware logic resources can be significantly enhanced. To maximize the utilization of hardware devices to serve more users, SAHA also integrates our previously proposed hardware virtualization technology [9], and thus a hardware device is virtualized such that it can be accessed by more than one user at the same time. To determine when the hardware functionalities must be adapted, SAHA provides a robust ubiquitous computing infrastructure. It includes service suppliers to interact with users, a hardware adapter to manage hardware functionalities on an FPGA device, an observer to be aware of the characteristics of user applications, and a system manager to manage all system adaptations. With the changes in environment conditions and user requirements, SAHA can effectively and autonomously adapt its hardware/software functions to support better performance.

Our previous works [8], [9] have introduced the detailed designs of hardware preemption and virtualization mechanisms, respectively, and this work will focus on the design of SAHA for ubiquitous computing applications. The rest of the article is organized as follows. Section 2 discusses the related research works. Section 3 introduces the design of SAHA, while Section 4 introduces how the hardware virtualization and preemption techniques can be realized in SAHA to support system adaptation. The experimental evaluation and analyses are given in Section 5. Finally, Section 6 concludes this work.

2. Related Work

Within a ubiquitous computing environment, computing devices are aware of changes in their environment, and thus they need to automatically adapt to environmental changes for satisfying user requirements. Odyssey [10] was a typical ubiquitous computing approach that supported application-aware adaptation to adjust the quality of accessed data to match available resources. However, due to the lack of support for coordination between the adaptation policies, its notification approach may lead to inefficient solutions. To more efficiently support the capability of adaptation, Efstratious et al. [1] proposed an architecture that could support adaptive context-aware applications. However, a main constraint in the approach was that the infrastructure only notified applications about the environmental changes, and then application themselves needed to trigger the adaptive mechanism. Instead of the passive application adaptation, Ghim et al. [2] proposed a reflective approach to dynamic adaptation that could perform adaptation operation triggered by changes in the policy and context, while users and applications could trigger adaptation using policy documents. However, hardware functions in the existing research works [1], [2], [10] cannot be adapted to different requirements, which thus restricts system adaptation and performance

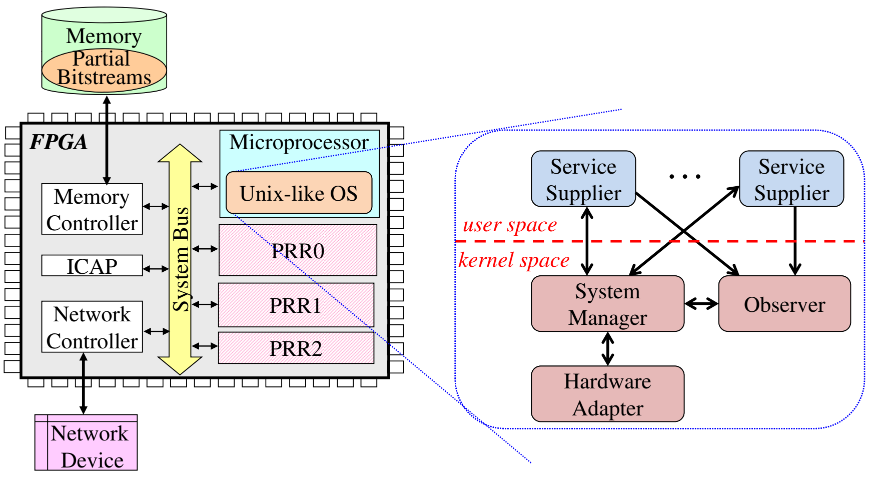


Fig. 2: Self-Adaptive Hardware-Software System Architecture

enhancement.

To support hardware adaptation, Danek et al. [4] proposed self-adaptive networked entities (SANE) to build pervasive computing architectures. The SANE design included an observer to monitor computation process. Furthermore, a computing engine could be on-demand reconfigured as different hardware functions at runtime to adapt to different system requirements, by using the partial dynamic reconfiguration technology [6]. Lagger et al. [5] also proposed a self-reconfigurable pervasive platform for cryptographic applications. They compared a full-software design with a coprocessor design that had an FPGA device which could be partially configured with different cryptographic hardware cores. Compared to the former, the performance of the latter was significantly enhanced due to dynamic reconfiguration techniques. However, these existing approaches to hardware adaptation [4], [5] still managed reconfigurable hardware functions as conventional hardware devices. As a result, the enhancement of system performance using partial reconfiguration technology is still limited, which makes the utilization of reconfigurable hardware functions inefficient.

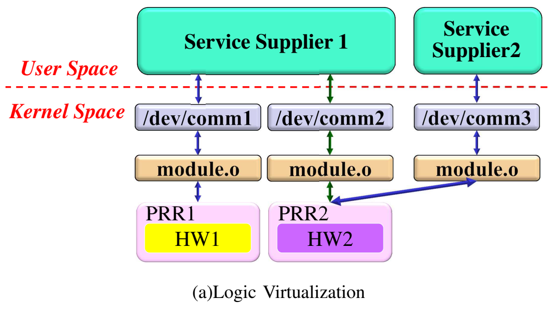
In this work, SAHA provides a robust ubiquitous computing infrastructure that includes service suppliers, a hardware adapter, an observer, a system manager, a reconfigurable hardware architecture. Instead of focusing on only software adaptation [1], [2], [10], SAHA further supports hardware adaptation by adopting the partial reconfiguration technology. Different from the existing approaches to hardware adaptation [4], [5] that manage reconfigurable hardware functions as conventional hardware devices, SAHA integrates hardware preemption [8] and hardware virtualization [9] for providing more efficient system adaptation to changing ubiquitous computing applications. At the same time, system performance can be further enhanced through system adaptation.

3. Self-Adaptive Hardware-Software System Architecture

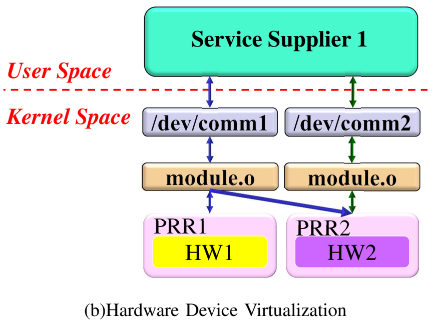
To realize hardware adaptation, we implement the hardware architecture of SAHA in an FPGA device, instead of an ASIC device. In general, not all hardware functions are always accessed by the software applications. Using the FPGA devices, hardware functions can be configured on-demand at runtime, without integrating them into the system at design time. As a result, though the total amount of logic resources required by all hardware functions exceeds that available in the FPGA, SAHA can still support a larger number of hardware functions by using the capability of hardware adaptation. Furthermore, because a large part of computing-intensive functions in the current SAHA implementation are cryptographic and hash functions, the parameter-specific architecture of the FPGA device also enables SAHA to provide better performance [11].

The SAHA design consists of a microprocessor, an FPGA device, a network device, and an off-chip memory, as shown in Figure 2. The hardware architecture of SAHA is implemented by using the *Early Access Partial Reconfiguration* (EA PR) design flow [6], in which several *Partial Reconfigurable Regions* (PRRs), such as `PRR0`, `PRR1`, and `PRR2` as shown in Figure 2, are implemented in the FPGA. They can be dynamically reconfigured as different hardware functions to meet the requirements for different service suppliers. All reconfigurable hardware functions are encapsulated as partial bitstreams and stored in an off-chip memory. Further, the hardware architecture of SAHA contains an *Internal Configuration Access Port* (ICAP) controller to configure the partial bitstreams into the FPGA.

The microprocessor runs a unix-like *Operating System* (OS) that includes service suppliers, a hardware adapter, a system manager, and an observer. Their detailed introductions are given in the following sections.



(a)Logic Virtualization



(b)Hardware Device Virtualization

Fig. 3: Hardware Virtualization Mechanism

3.1 Service Supplier

The service supplier is a software application that interacts with a user, while it interacts with hardware functions for data capture and processing. Each service supplier has a priority, and thus the system manager can decide when it starts to serve its user, based on the current ubiquitous computing environmental conditions. Further, similar to the interactions between software applications and hardware devices in a conventional embedded OS, service suppliers in SAHA also interact with reconfigurable hardware functions through the device nodes. As a result, though SAHA supports system adaptation, the generality in accessing the hardware device design is still not sacrificed. Further, because all software and hardware adaptations in SAHA are performed underlying the kernel level, service suppliers can serve their users in a natural interaction method, without any adjustment in them.

3.2 Hardware Adapter

The hardware adapter is used to manage the reconfigurable hardware architecture. When a service supplier requests a hardware function that is not configured in the FPGA, the hardware adapter thus loads the corresponding partial bitstream to the ICAP controller for configuring the required hardware function into the FPGA. The hardware adapter also records which hardware functions are now configured in the PRRs and which service suppliers are accessing them. As a result, the system manager can inquire about the information to estimate which system adaptation, such as hardware reconfiguration, hardware virtualization, and hardware preemption, is better for all ubiquitous computing applications. Further, the hardware adapter needs to support the system manager for performing the hardware preemption mechanism. The details of the hardware preemption mechanism in SAHA will be introduced in the next section.

3.3 System Manager

The system manager is the core of SAHA that coordinates all system adaptations. It not only manages all data transfers within the SAHA design, but also supports the hardware virtualization and preemption techniques.

Two types of hardware virtualizations, namely logic virtualization and hardware device virtualization, are supported by the system manager, as described in the following. 1) In the many-to-one logic virtualization as shown in Figure 3(a), a required hardware function (`HW2`) is virtualized such that it can be accessed by two service suppliers, `Service Supplier 1` and `Service Supplier 2`, through the device nodes `comm2` and `comm3`, respectively. Here, the data synchronization is controlled by the system manager. This many-to-one mapping between more than one service supplier and a configured hardware function increases the utilization of a hardware function. 2) In the hardware device virtualization as illustrated in Figure 3(b), the kernel module that is linked to and drives a specific hardware function (`HW1`) can be dynamically re-linked to another hardware function `HW2`. Without unnecessarily copying data between the *user space* and *kernel space*, the processing results of `HW1` can be directly transferred to `HW2` through the kernel module. This one-to-many mapping is a seamless reconfiguration of the underlying hardware, without any change to the software. Thus, using the hardware virtualization technique, multiple service suppliers can serve their users under the illusion of full access to the same reconfigurable hardware function through their own device nodes, while a service supplier can also access two or more reconfigurable hardware functions. Therefore, for changing ubiquitous computing environments, SAHA can adapt to more user requirements, while it can provide better performance.

Though the partial reconfiguration technique enables parts of an FPGA device to be reconfigured as different hardware functions at runtime, the occupied logic resources of a configured hardware function can be released only after it finishes its execution. This would incur significant time overhead, and thus degrade system performance. However, for ubiquitous computing applications, the support for seamless services is very important. As a result, SAHA integrates the hardware preemption technique to further enhance the utilization of hardware logic resources per unit time. When the hardware preemption mechanism is triggered, the system manager requests the hardware adapter to send a swap-out

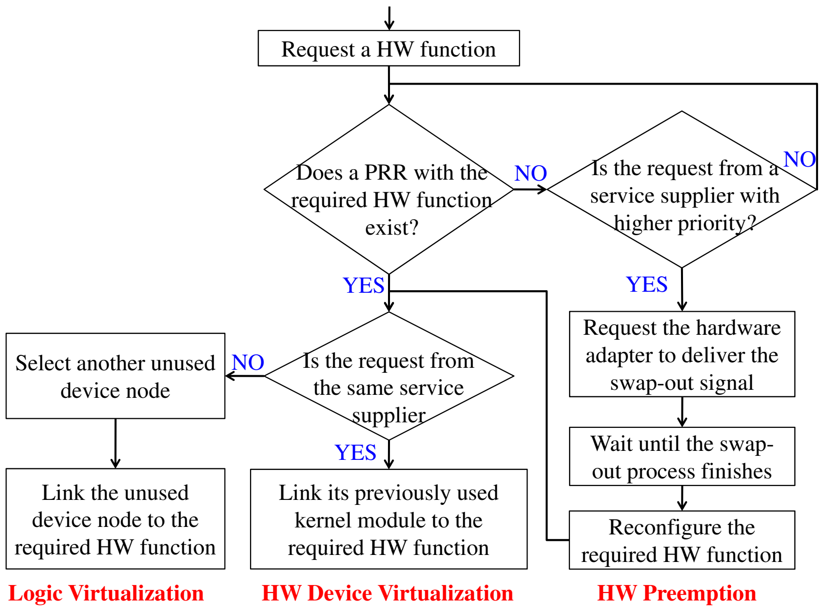

Request a HW function
Does a PRR with the required HW function exist?
NO
Is the request from a service supplier with higher priority?
NO
YES
YES
Request the hardware adapter to deliver the swap-out signal
Wait until the swap-out process finishes
Reconfigure the required HW function
Select another unused device node
NO
Is the request from the same service supplier
YES
Link the unused device node to the required HW function
Link its previously used kernel module to the required HW function
Logic Virtualization
HW Device Virtualization
HW Preemption

Fig. 4: Self-Aware System Adaptation

signal to a hardware function that is being accessed by a service supplier with lower priority. After all the context data of the swap-out hardware function are saved in an off-chip memory, the hardware adapter reconfigures the required hardware function into the corresponding PRR. Then, it notifies the system manager that the hardware function can be accessed. Here, the context data are the collections of the state registers and data registers. When the swap-out hardware function is reconfigured into the FPGA and all its context data are restored, it can continue execution from the state in which it was swapped out.

As for the determination of when hardware virtualization and preemption mechanisms are used due to system adaptation, the system manager needs to collaborate with the observer. The detailed introduction will be given in the next section.

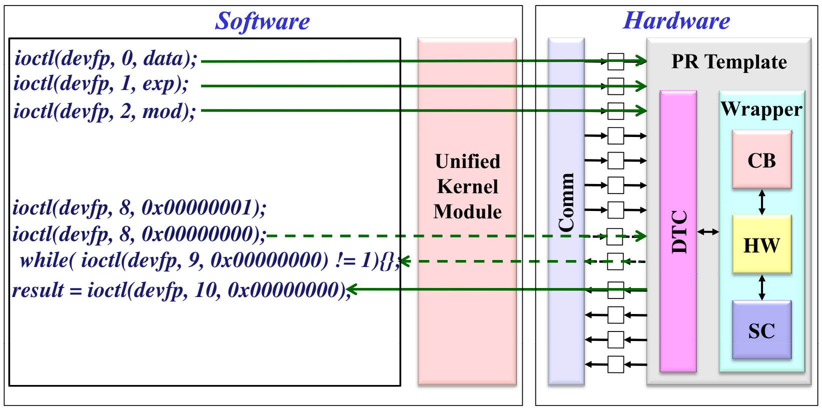
3.4 Observer

Both the system manager and the hardware adapter provide the capability of system adaptivity for SAHA. As to when system adaptation is triggered, it mainly depends on the observer. Thus, the observer plays a key role to make SAHA self-aware to adapt to ubiquitous computing environments. The observer is aware of the characteristics of currently running applications. As shown in Figure 4, it collaborates with the system manager to decide which of the three proposed techniques, namely logic virtualization, hardware device virtualization, and hardware preemption.

When a service supplier requests for a hardware function, the observer first asks the system manager whether the required hardware function has been configured in a PRR. If not, the observer determines what the priority level of the service supplier is, and then it queries the system manager whether the priority of this service supplier is higher than any service supplier that is accessing reconfigurable hardware functions. In our current implementation of SAHA, the priority levels of all service suppliers need to be preset during system initialization by users. Otherwise, SAHA is based on a first-come-first-served basis. If the service supplier does not have higher priority, it will wait until the required hardware function is configured into the FPGA. Otherwise, the hardware preemption mechanism is invoked, such that the system manager requests the hardware adapter to start the swap-out process. After the swap-out process finishes, the required hardware can be configured into the FPGA. Then, the observer continues to collaborate with the system manager to decide which hardware virtualization technique will be used.

When the required hardware function is already configured, the observer checks whether the request is received from the same service supplier. If not, the system manager invokes the logic virtualization to dynamically link another unused device node to the required hardware function. Otherwise, the system manager invokes the hardware device virtualization to dynamically link the previously used kernel module to the PRR with the required hardware function, and thus the processing results of the previous hardware function can be directly transferred to the requested hardware function. Note that, using the hardware device virtualization, when a pair of device node and kernel module is linked to only one hardware function, the final processing results are thus transferred back to the service supplier.

Through the cooperation between the service suppliers,


Software
ioctl(devfp, 0, data);
ioctl(devfp, 1, exp);
ioctl(devfp, 2, mod);
ioctl(devfp, 8, 0x00000001);
ioctl(devfp, 8, 0x00000000);
while(ioctl(devfp, 9, 0x00000000) != 1){};
result = ioctl(devfp, 10, 0x00000000);
Unified Kernel Module
Hardware
Comm
PR Template
DTC
Wrapper
CB
HW
SC

Comm: Communication Interface Component; DTC: Data Transformation Component;
CB: Context Buffer; SC: Swap Controller

Fig. 5: Unified Communication Mechanism

the hardware adapter, the system manager, and the observer, SAHA can provide a robust infrastructure that is capable of self-aware adaptation to ubiquitous computing applications. As to what types of ubiquitous computing applications are suited to SAHA, the related analysis will be given in Section 5.1.

4. Implementation of Hardware-Software Adaptation

Besides the support of reconfigurable hardware architecture as introduced in Section 3, this section will introduce how the hardware virtualization and preemption techniques can be realized in SAHA to support system adaptation.

To support the hardware virtualization mechanism, both the interfaces of device drivers and reconfigurable hardware functions in SAHA must be unified, such that service suppliers and reconfigurable hardware functions can be many-to-one and one-to-many mapped. Therefore, we further propose a unified communication mechanism in SAHA to standardize the hardware/software communication interface, as shown in Figure 5. To minimize user design efforts, in our design environment, a user-designed hardware function can be integrated with a 3-tier interface, including (a) a partially reconfigurable (PR) template [12], (b) a reusable wrapper design, and (c) a communication interface component.

The PR template consists of eight 32-bit input data signals, one 32-bit input control signal, four 32-bit output data signals, and one 32-bit output control signal. It also contains an optional *Data Transformation Component* (DTC) for unpacking incoming data and packing outgoing data based on the I/O registers sizes in the hardware functions, and a reusable wrapper design to support hardware preemption technique. The wrapper architecture in Figure 5 mainly consists of a context buffer (`CB`) to store context data and a swap controller (`SC`) to manage the swap-out and swap-in activities.

Two types of reusable wrapper designs, namely *Last Interruptible State Swap* (LISS) wrapper and *Next Interruptible State Swap* (NISS) wrapper, are supported by SAHA, as described in the following. 1) The LISS wrapper stores the hardware context at each interruptible state, thus the hardware function can be swapped out from the last interruptible state whenever there is a swap request. It can be used for hardware functions whose context data size is less than that of the context buffer, as a result of which all context data can be stored in the context buffer using a single data transfer. 2) The NISS wrapper requires the hardware function to execute until the next interruptible state, store the context, and then swap out. It can be used when the context data size is larger than that of the context buffer, where the process of storing the context data into buffer and reading into memory is repeated and controlled by the hardware adapter. The communication interface component is used to connect the PR template to the system bus. Further, the processing results can be buffered in the communication interface component until the service supplier reads them.

Within the software part, different from the traditional device driver designed for a specific hardware function, a unified kernel module is designed to only interact with the fourteen 32-bit signals of the PR template. All the interactions between service suppliers and reconfigurable hardware functions are through the `ioctl` system calls of the unified kernel module and implemented in a hardware control library. Thus, a new user-designed hardware function needs to be only integrated with a PR template and included its control methods into the hardware control library. As a result, SAHA can be easily extended to support more types of ubiquitous computing applications, without being only restricted to our current implemented applications.

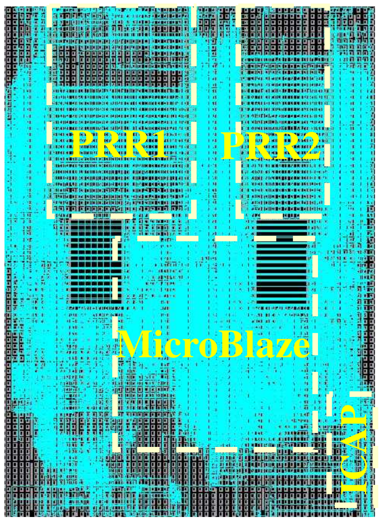


Fig. 6: FPGA Implementation

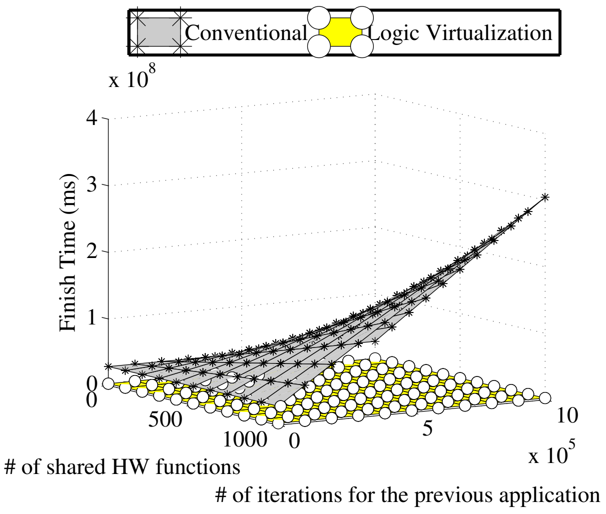


Fig. 7: Logic Virtualization and Conventional Method

5. Experimental Evaluation

The current SAHA design was implemented on the Xilinx ML310 platform [13] with a Virtex II Pro FPGA device, in which the PetaLinux embedded OS [14] ran on a Xilinx MicroBlaze soft-core processor [15] as shown in Figure 6. Due to the resource limitations of our current implementation, only a large `PRR1` and a small `PRR2` are implemented in the FPGA. In this section, we will first analyze the hardware virtualization and preemption techniques in SAHA, and then give a case study that SAHA itself adapts to ubiquitous computing applications using both hardware preemption and hardware virtualization.

5.1 Performance Analysis

In the following sections, we will discuss the impact on system performance, respectively, using hardware virtualization and hardware preemption.

5.1.1 Hardware Virtualization

To analyze the impact on system performance using hardware virtualization in SAHA, we first measure the time required for each of the following basic operations. The average amounts of time to insert a kernel module, to open a device node and to close it are 830, 0.253, and 0.048 milliseconds, respectively. The average amounts of time to write a 32-bit data to and to read a 32-bit data from the kernel space are 0.022 and 0.022 milliseconds, respectively. The average computing time for processing a 128×64 pixel image using the RSA32, RSA64, RSA128, RC6, CRC32, CRC64, and CRC128 hardware functions is 0.026 milliseconds, while the average time to read a 32-bit data from a hardware function and then write it to another hardware function through the unified kernel module is 0.0028 milliseconds.

As introduced in Section 3.3, through logic virtualization, a service supplier can access a hardware function that has been accessed by another previous service supplier, as long as the previous service supplier is not using it, irrespective of whether it is released or not. Without logic virtualization, the hardware function cannot be accessed until the previous service supplier releases it. Thus, using the conventional method in the best case, the waiting time would be equal to the total processing time of the shared hardware functions in the previous service supplier. Based on our measurement of the time required by each basic operation, Figure 7 compares logic virtualization and the conventional access method by looking at the finish time of a service supplier that has some hardware functions previously accessed by another service supplier. Here, the numbers of shared hardware functions and iterations to access the shared hardware functions in the previous service supplier are set ranging from 0 to $1,000$ and from 0 to $1,000,000$, respectively. We can observe that the time required by using the logic virtualization becomes lesser and lesser compared to the time required by using the conventional access method, when the numbers of shared hardware functions and iterations to access the shared hardware functions in the previous service supplier gradually increases. This is because logic virtualization reduces significantly the waiting time for the shared hardware functions. Thus, performance improvement using the logic virtualization becomes more and more prominent with an increase in the total processing time of shared hardware functions in the previous service supplier.

Using hardware device virtualization, the processing results (data) of a hardware function can be directly transferred to another hardware function through the unified kernel module. Thus, hardware device virtualization is used to save time by avoiding repeated data transfers between the OS

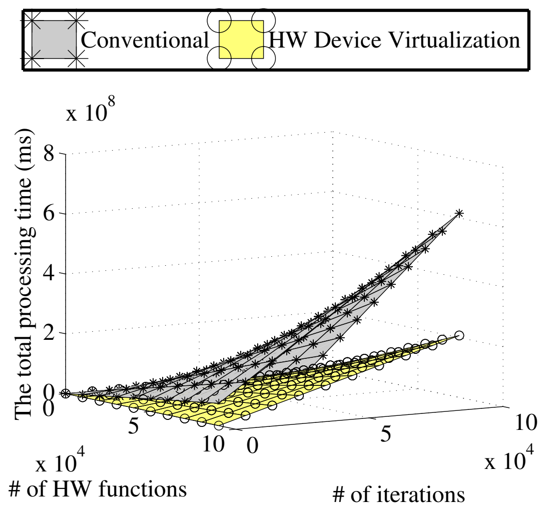


Fig. 8: HW Device Virtualization and Conventional Method

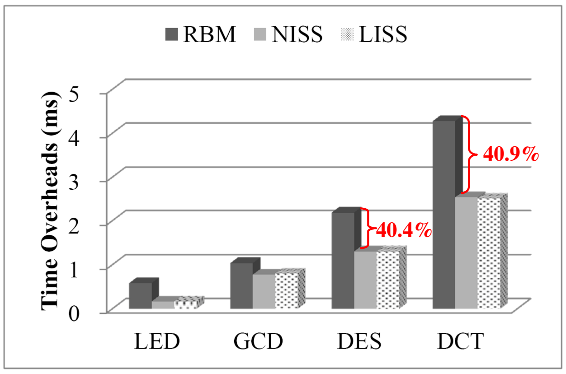


Fig. 9: Experiments on Hardware Preemption

kernel and the user level. Based on our measurement of the time required by each basic operation, Figure 8 shows the total processing time of a service supplier using the hardware device virtualization technique and the conventional method. Here, the number of hardware functions to be sequentially accessed by the service supplier and the number of iterations to access the hardware functions are both set ranging from 0 to $100,000$. We can observe that the time required by using the hardware device virtualization technique becomes lesser and lesser compared to the time required by using the conventional method, when the numbers of hardware functions and iterations to access the hardware functions gradually increase.

From the above results, we can clearly observe that the hardware virtualization technique allows greater performance improvement, compared to the conventional method, when a service supplier requires more and more iterations to access hardware functions. This also shows that the hardware virtualization technique will play a key role to help system adaptation and provide better performance, especially for ubiquitous computing applications, such as multimedia applications and the cryptographic applications, that usually must process a larger amount of information and data.

5.1.2 Hardware Preemption

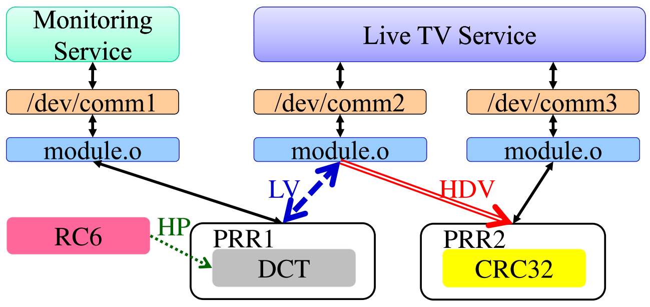
In our current implementation, the software processing time is much more than the hardware processing time. To clearly show the contribution to hardware preemption in SAHA, we directly compared it with another reconfiguration-based method (RBM) [16]. Here, RBM requires readback support from the reconfigurable logic such as state extraction from the readback stream and manipulation of the bitstreams for context restoring.

Figure 9 shows the time overheads in swapping out and swapping in for the LED controller, the *Greatest Common Divisor* (GCD), the *Data Encryption Standard* (DES), and DCT hardware functions. We can observe that the hardware preemption mechanism in SAHA performs better than RBM. For the more complex DES and DCT hardware functions, our method can reduce the swap time by 40.4% and 40.9%, respectively, of that required by RBM. We are thus saving much time, which is very important for supporting seamless ubiquitous computing services. Even though additional reconfiguration time is required, the hardware preemption mechanism in SAHA would enable more service suppliers to adapt to changing environmental conditions and achieve higher system performance.

5.2 Case Study using both HW Preemption and Virtualization

The impacts on performance using the hardware preemption technique and the hardware virtualization technique have been discussed in Section 5.1. In this case study, SAHA will serve two ubiquitous computing applications, including a monitoring service and a live television service. Note that this experiment does not use the stream buffering technology and mainly shows performance improvement in SAHA, and thus the amounts of time to capture data from the peripherals and transfer data to users on the network are omitted. As shown in Figure 10, the live television service requests for processing fifty images using the RC6 and CRC32 hardware functions. However, the DCT and CRC32 hardware functions are now configured in `PRR1` and `PRR2`, respectively, where the DCT function required by the low-priority monitoring service has already executed for 55,000 ms and `PRR1` is selected to configure the required RC6 function.

Without the hardware preemption technique, the RC6 hardware function will be configured in `PRR1` only after the DCT function finishes. According to our experimental results, it needs 13,122.8 ms exclusive of the previous execution time of 55,000 ms to finish the current DCT execution and 177 ms to configure the RC6 function. Note



Priority: Live TV Service > Monitoring Service
HP: Hardware Preemption; LV: Logic Virtualization;
HDV: Hardware Device Virtualization;

Fig. 10: Case Study using both Hardware Preemption and Virtualization

Table 1: Comparison on Processing Time

Case	System Adaptation		Time (sec)	Improvement
	Preemption	Virtualization		
A	No	No	109.696	
B	Yes	No	96.575	11.96%
C	No	Yes	98.635	10.08%
D	Yes	Yes	85.514	22.04%

that here we assume that the current monitoring service (with requirement for the DCT hardware function) can also be preempted after one DCT execution. If this is not the case, a much greater delay will be incurred without the hardware and software preemption. However, using the hardware preemption technique, it needs only 1.2687 ms for the NISS wrapper or 1.2552 ms for the LISS wrapper to swap out the DCT function, and 177 ms to configure the RC6 function.

After hardware and software preemption, the required RC6 hardware function is configured into `PRR1`. Using the conventional method, the live television service must now wait until the currently executing monitoring service closes the device node `comm1` that is linked to `PRR1`, and then the live television service opens the same device node `comm1` to access `PRR1` with the RC6 hardware function. However, using the logic virtualization, another pair `comm2` of device node and kernel module can be simultaneously used to connect to `PRR1` and the live television service starts accessing `PRR1` with the RC6 hardware function more quickly. The time saved by not waiting for another monitoring service to close and open the device node `comm1`, namely logic virtualization, is around 0.301 seconds according to our experimental results.

Finally, using the hardware device virtualization, the processing results (output data) of the RC6 hardware function in `PRR1` can be directly transferred to the CRC32 hardware function in `PRR2` through the kernel module in the pair `comm2` of device node and kernel module without going back and forth between the OS kernel and the user levels, as introduced in Section 3.3. Table 1 gives the total processing time required for finishing the live television service, where Cases A, B, C, and D represent the four scenarios that either use or do not use hardware preemption, and/or hardware virtualization. Case A is the conventional method in a Unix-like embedded system, thus it is taken as the baseline method. The fifth column of Table 1 shows the percentage of performance improvement compared to the baseline method Case A. We can observe that system performance can be significantly enhanced, when hardware preemption (Case B) or hardware virtualization (Case C) is used. Here, the performance improvement using hardware preemption results from the configuration of RC6 without waiting until the monitoring service completes the compression of an image. Thus, more improvements can be achieved with hardware preemption, when SAHA receives the request for preempting the DCT hardware function earlier. Employing both hardware preemption and hardware virtualization (Case D), SAHA can reduce the processing time by around 22.04% of that required by using the conventional method. As a result, system adaptation in SAHA can provide not only seamless service but also more efficient infrastructure support for the services in embedded and ubiquitous applications.

6. Conclusion

To provide a more robust ubiquitous computing infrastructure, we propose a self-adaptive hardware-software system architecture (SAHA) that solves the issues related to system adaptation, as described in Section 1. SAHA also integrates the hardware preemption and virtualization techniques and has a complete self-aware system adaptation mechanism to provide better performance for ubiquitous computing applications. Our experimental results also demonstrate that not only system resources can be effectively utilized, but system performance can be also improved significantly, when ubiquitous computing applications are served by SAHA.

References

[1] C. Efstratiou, K. Cheverst, N. Davices, and A. Friday, “An Architecture for the Effective Support of Adaptive Context-Aware Applications,” in *Proc. of the 2nd International Conference on Mobile Data Management (MDM 2001)*. Springer, January 2001, pp. 15–26.

[2] S.-J. Ghim, Y.-I. Yoon, and J.-W. Choe, “A Reflective Approach to Dynamic Adaptation in Ubiquitous Computing Environment,” in *Proc. of the International Conference on Networking Technologies for Broadband and Mobile Networks (ICOIN 2004)*. Springer, February 2004, pp. 75–82.

[3] J.-Z. Sun, “Adaptive Determination of Data Granularity for QoS-Constraint Data Gathering in Wireless Sensor Networks,” in *Proc. of the 2009 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing (UIC-ATC 2009)*. IEEE Computer Society, June 2009, pp. 401–405.

[4] M. Danek, J.-M. Philippe, P. Honzik, C. Gamrat, and R. Bartosinski, “Self-Adaptive Networked Entities for Building Pervasive Computing Architectures,” in *Proc. of the 8th International Conference on Evolvable Systems: From Biology to Hardware (ICES 2008)*. Springer, September 2008, pp. 21–24.

[5] A. Lagger, A. Upegui, E. Sanchez, and I. Gonzalez, “Self-Reconfigurable Pervasive Platform for Cryptographic Application,” in *Proc. of the 16th IEEE International Conference on Field Programmable Logic and Applications (FPL06)*. IEEE CS Press, August 2006, pp. 777–780.

[6] Xilinx, “Early Access Partial Reconfiguration User Guide, UG208,” 2006.

[7] P.-A. Hsiung, M. D. Santambrogio, and C.-H. Huang, *Reconfigurable System Design and Verification*. CRC Press, USA, ISBN: 978-1420062663, 2009.

[8] C.-H. Huang and P.-A. Hsiung, “Software-Controlled Dynamically Swappable Hardware Design in Partially Reconfigurable Systems,” *EURASIP Journal on Embedded System*, vol. 2008, article ID 231940, 11 pages, doi:10.1155/2008/231940.

[9] ——, “Hardware Resource Virtualization for Dynamically Partially Reconfigurable Systems,” *IEEE Embedded Systems Letters*, vol. 1, no. 1, pp. 19–23, May 2009.

[10] B. Noble, “System Support for Mobile, Adaptive Applications,” *IEEE Personal Communications*, vol. 7, no. 1, pp. 44–49, February 2000.

[11] T. Wollinger and C. Paar, “How Secure are FPGAs in Cryptographic Applications,” in *Proc. of the 13th IEEE International Conference on Field Programmable Logic and Applications (FPL'03)*. Springer Verlag, September 2003, pp. 1–3.

[12] C.-H. Huang, P.-A. Hsiung, and J.-S. Shen, “UML-based hardware/software co-design platform for dynamically partially reconfigurable network security systems,” *Journal of Systems Architecture (JSA)*, vol. 56, no. 2-3, pp. 88–102, February 2010, (doi: 10.1016/j.sysarc.2009.11.007).

[13] Xilinx, “ML310 User Guide,” 2007.

[14] PetaLogix, “PetaLinux,” http://www.petalogix.com/.

[15] Xilinx, “MicroBlaze Processor Reference Guide, Embedded Development Kit, EDK 8.2i - UG081 (v6.3),” August 2006.

[16] H. Kalte and M. Porrmann, “Context Saving and Restoring for Multitasking in Reconfigurable Systems,” in *Proc. of the 15th IEEE International Conference on Field Programmable Logic and Applications (FPL'05)*. IEEE CS Press, August 2005, pp. 223–228.

SESSION

REGULAR PAPERS

Chair(s)

PROF. DAVID ANDREWS

Evaluating Expression Trees in Hardware

Lars Middendorf[1] **and Christophe Bobda**[2]
[1]Institute of Computer Science, University of Potsdam, Potsdam, Germany
[2]College of Engineering, University of Arkansas, Fayetteville, Arkansas, USA

Abstract— *We propose expression trees as an universal format for communication and as an executable specification for work-items. The presented application-specific instruction-set processor (ASIP) interprets a stream of expressions received from the main processor to speed-up expensive floating-point operations. Due to our term rewriting approach, long pipelines, as they are required for floating-point operations, can be realized without explicit scheduling. In contrast to RISC processors, a register file is not necessary because the arguments can be taken directly from the stream. Similar to DSP filters, only a short shift-register is necessary to remember a set of previous values. Complex examples from the field of computer graphics show the usefulness of our approach.*

Keywords: Expression Trees, ASIP, Streaming, FPGA

1. Introduction

Due to decreased time-to-market and rising requirements on functionality and performance, a co-design of hardware and software components is essential. Since both parts depend on each other, the design of a stable interface at the beginning of the development process guarantees interoperability and verifiability. In addition, implementation details like the topology of the network and the partitioning of the system may be influenced by these design decision.

Usually, an abstraction based on tasks is used to describe the interaction between modules on both sides [1]. While a uniform set of similar modules is easier to program, often specialized processing units like digital signal processors (DSP) or custom hardware blocks are utilized to meet performance goals. Especially in the case of manycore systems, the communication between these modules still remains an open problem. A shared bus or network-on-chip (NoC) only provides the infrastructure for data exchange, but it usually does not define a protocol to distribute tasks across processing units [2]. An intermediate layer, supporting the creation of threads in hardware and software, hides this complexity and creates a consistent view of the system. In this case, it is possible to build operating system services like the scheduler in hardware to reduce latency [3] [4]. Hence, even late in the development cycle, tasks can be transferred into hardware without adapting the clients.

While smaller systems can use shared memory for synchronization, larger systems rely on message-passing as a scalable communication method [5]. Instead of flat packets, we propose to use expression trees for distributing work-items in a manycore system with the following considerable advantages:

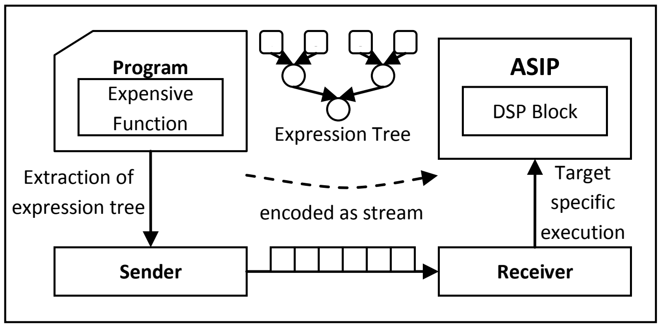


Fig. 1: Expressions tree used for communication.

Expression trees offer an advantageous approach for decoding tasks, since they cannot only encapsulate function calls but also arbitrary program fragments. Similar to the decorated abstract syntax tree built during compilation, our expressions contain all information necessary for code generation or interpretation, so that they provide a self-contained and portable encapsulation of both data and instructions. Furthermore, they are mostly independent of both the source language and the target architecture (Fig. 1).

There are node types for primitive operations and functions, so that micro-threads and course grained tasks can be described using the same technique. As a result, communication costs can be reduced if possible while being still able to compose tasks dynamically. Due to this flexibility, parts of an application can be moved gradually into hardware and accelerated as necessary. Unlike C programs which are expensive to move between processors due to side effects and external references, expression trees can carry the current execution environment by capturing free variables in a closure and are therefore much better suited to transfer tasks between processing units.

The rest of the paper is organized as follows: In section 2, we will look at related work in the area of stream processing and expression trees. In the following section, the concept of our processor is described in detail and a formal specification is given. After that, we transfer this approach into a hardware design which is then tested in an example system-on-chip (SoC) to accelerate graphic applications. Finally, the last section is used for discussion and further ideas.

2. Related Work

Expression trees already have been used successfully as an intermediate format for the exchange of program fragments between different environments. The *Language Integrated Query* (LINQ) employs expression trees to embed queries, which are similar to SQL, in C# or Visual Basic programs. If the compiler encounters an assignment to a variable of a special type, it does not generate code, but instead stores the tree of the expression directly. At runtime, the tree is then transformed it into real query and sent to the database server. It is also possible to modify and compile expressions dynamically [6].

In fact, programs for stack-based languages are post-order serialized expression trees. They are often used as an intermediate representation for stack-oriented environments like the Java VM or the .Net IL, which is not surprising since a stack machine provides both a compact though powerful and portable abstraction. Also certain programming languages like *Forth*, *Joy*, *Factor* and *PostScript* consist almost completely of operations modifying a stack [7].

However, there is the second possibility to execute a stack-based program by *term rewriting* [8]. Nodes containing only literal values as inputs are replaced successively by the result of the arithmetic operation. In contrast to the stack-based evaluation, the tree is visited using breadth-first search instead of depth-first order. Hence, all nodes at the same level are processed in the same pass. Since there cannot be any interdependencies between them, the temporary results from one step are not needed until the next iteration. As a result, it is feasible for a hardware implementation to use pipelining if the tree is wide enough to hide the latency.

The *Queue Processor* has been designed as an accelerator to speed-up the execution of a basic block, which is usually the body of an expensive loop [9]. As a pure queue machine, it also has some restrictions: An instruction can only read from the end of the queue, so that it cannot refer to arbitrary arguments. Therefore, the data flow graph of the program fragment should be planar without crossing edges, which means that variables should be optimally accessed continuously. The queue processor supports special commands to duplicate and swap the items at the end of the queue, so that it can execute all code, but often half of the instructions in a program are used to reorganize the queue. However, all calculations can be optimally pipelined, so that this approach is also usable for expensive and high-performance computations. ur work is similar to the queue processor, since we will also focus on a pipelined execution. However, the queue processor supports only basic arithmetic operations and is limited to data flow graphs. Contrary, our system deals with trees that can also contain function calls and definitions. Further, the use of positional parameters avoids the expensive reordering on the stream as it is the case for the queue processor.

3. Expression Trees

In this section, we will define the format of the expression trees and specify a set of rules which are later used for hardware-based evaluation.

3.1 Expressions

Our system consists of a tree grammar G and a set of rules R that provide a mapping of one tree to another. The grammar determines the set of possible node types, which are described later in this section. A tree is executed by applying all possible rules iteratively until there is no more match. A rule normally reduces nodes of the tree and replaces them by their result, but there are also rules that expand definitions and create new branches. It is assumed that the rules are well-defined, so that there is only at most one matching rule for a given tree and that no rule is the identity function. A program is a set of rules extending the predefined grammar. Let T be the language generated by G, so that $t \in T$ is a tree. Further, let f_R be a function that applies all matching rules of R and returns a modified tree.

$$f_R : T \to T$$

For example, R may contain a rule that replaces $A + B$ by the result of the addition if A and B are constants. Then $f_R(1 + 2)$ is evaluated to $f_R(3)$, while $f_R(1 + x)$ cannot be reduced. In general, the function $f_R(t)^n$ is the temporary result of t after n iterations. If we choose a discrete metric to compute the difference between two trees, the final result can be described as the limiting value:

$$F(t) = \lim_{n\to\infty} f_R(t)^n$$

Our goal is to implement the function f_R in hardware. Since expression trees are immutable and contain all necessary information, global memory writes can be avoided and the execution can be parallelized beyond simple data parallelism.

3.2 Encoding

We choose to serialize the trees into sequences because a data stream can be processed efficiently in hardware and does not require random memory accesses. A node of the tree is described by a token $(type, value)$ which contains a type and an optional argument. For instance, the equation $(A \cdot B) + (C \cdot D)$ is described by the sequence $AB \cdot CD \cdot +$. Figure 2 shows the evaluation of this tree by repeated application of f_R. The individual rules are explained later. A post- or pre-order representation is advantageous, because it does not require parentheses to encode operator precedence, so that the overall storage can be reduced. Further, we preferred a depth-first traversal over a breadth-first, to allow for efficient substitution of sub-trees. In a depth-first scheme, a sub-tree always corresponds to a single range of tokens that

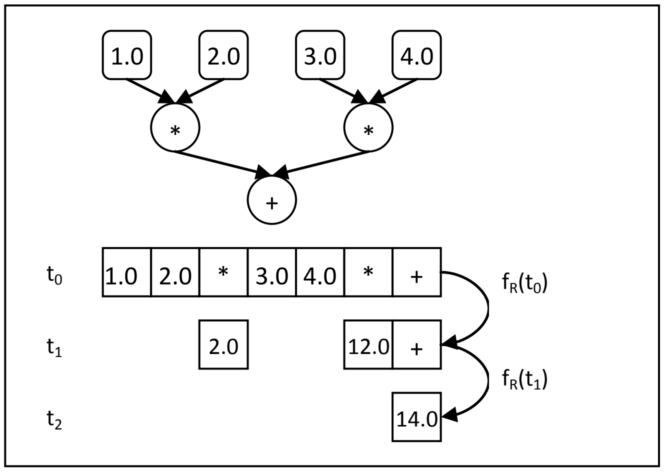


Fig. 2: Evaluation of an expression tree using term rewriting.

Table 1: Token types.

Token	Description
CONST(value)	Literal value
SYMBOL	Dereference a symbol
UNARY(operator)	Unary operation like $-$,
BINARY(operator)	Binary operation like $+$,$*$
DEF(name) ENDDEF	Definition of symbol $< name >$
IF, ELSE, ENDIF	Begin/End of a conditional block
FUNC(args) , ENDFUNC	Duplicate value
ARG(index)	Argument placeholder

can be replaced on-the fly while the tokens pass through a hardware component. The rules for arithmetic evaluation are applied on trees whose children are either constants or have been reduced to constants. In depth-first order, these trees consist of two or three consecutive elements, so that the internal storage of the hardware component is minimized. Table I lists the possible token types which are explained in the next sub-sections.

3.3 Arithmetic

Since our processor should accelerate floating-points calculations, our system needs to represent arithmetic expressions. We first allow all meaningful functions like addition, subtraction or multiplication. Later, we will adapt this set to an actual hardware implementation. For instance, it could be possible to realize expensive calculations like square-roots directly in hardware and represent them as unary expressions.

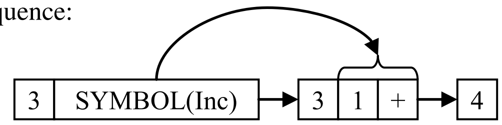
Constant tokens belong to the terminal symbols of our grammar and represent scalar literal expressions like 1.0 or 2.0. A sequence is evaluated iteratively by applying a set of replacement rules. Since unary and binary operators only work on numbers, they expect constant tokens directly before the operator. Otherwise, the function cannot execute and leaves the stream unmodified. Figure 2 shows a more complex example that is evaluated in two separate steps. In the first pass, only the inner multiplications are ready. The addition depends on intermediate results and has to wait for the second pass. Although all non-trivial expressions require multiple passes for complete evaluation, there are no data dependencies within a pass, making it possible execute these operations in a pipeline.

3.4 Definitions

To compose more complex functions efficiently, parts of an expression must be saved for later reuse. Further, we also want to limit the amount of data sent to reduce communication costs. For example, a matrix that is used to transform an array of vectors should not be retransmitted for every element in the array. Therefore, we need a mechanism to look-up expressions by a symbolic name. The following example shows the definition of a symbol using a pair of *DEF/ENDDEF* tokens.

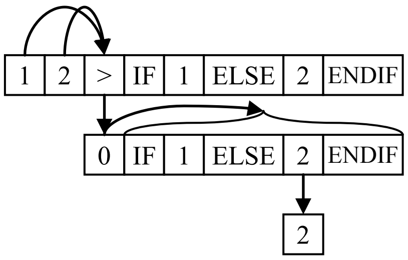
DEF(Inc)	1	+	ENDDEF

Later, the *SYMBOL* token can be used to insert the stored sequence:



3.5 Conditionals

Similar to definitions, conditional expressions are also implemented by a pair of begin and end markers (*IF/ENDIF*). Depending on the value of the token directly before this block, the content is either discarded or passed though. The *ELSE* token can appear inside the block and reverses this logic. In the diagram, the condition is evaluated to 0, so that the *if*-branch is omitted while the *else*-branch is kept.



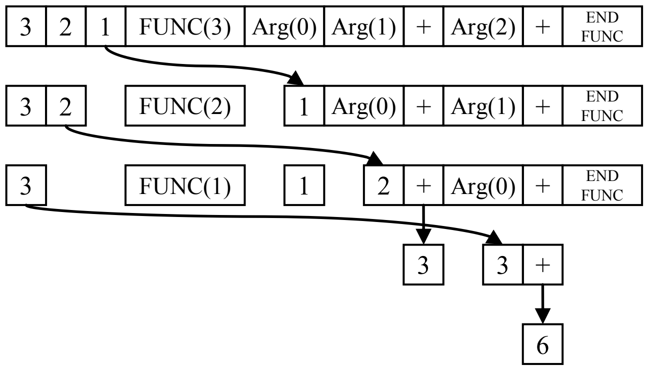
3.6 Functions

For term rewriting, the operands of an arithmetic expression must be placed immediately before the corresponding token, so that arguments and operators are interleaved. Contrary, code and data become separated when employing symbols to store functions for re-usability. Hence, a mechanism that re-combines different parts of a sequence is required. A function block (*FUNC(n)/ENDFUNC*) works like a template. It provides a mapping from a list of arguments to a sequence of tokens. Inside the block, an *ARG(i)* token marks the positions where the argument i should be inserted.

For example, the following function calculates the sum of three parameters:

FUNC(3)	Arg(0)	Arg(1)	+	Arg(2)	+	ENDFUNC

To evaluate this function, the formal parameters are bound to actual values. The binding rule replaces the first argument of the function by the first constant before the *FUNC* token. The result is again a function containing one argument less, so that also partial evaluation (*currying*) is supported. After the last argument has been applied, header and end marker are removed, leaving a constant function on the stream. The incremental evaluation is illustrated by the following example that passes the arguments $(1, 2, 3)$ to the function defined above:


3 2 1 FUNC(3) Arg(0) Arg(1) + Arg(2) + END FUNC
3 2 FUNC(2) 1 Arg(0) + Arg(1) + END FUNC
3 FUNC(1) 1 2 + Arg(0) + END FUNC
3 3 +
6

4. Expression Tree Processor

We will present an application-specific hardware accelerator that implements the rules specified in section 3.

4.1 Overview

Figure 3 shows the internal structure of our stream processor which resembles the function f_R defined in the previous section. The instruction set of this processor corresponds to the set of term rewriting rules. Each rule is implemented as a separate hardware module that analyzes and modifies the stream if necessary. The modules are connected in a pipeline, so that different parts of the stream can be replaced in parallel. A single iteration through this pipeline corresponds to one invocation of f_R.

An expression is received as a list of tokens from the host processor via a FIFO interface. Each token is represented as a packet with a size of 33 bit (Fig. 4). If bit 32 is zero then the rest of the word is a 32-bit constant value. Otherwise, the bits 31 to 24 describe the opcode and the lowest 24 bit store additional arguments. For example, the *DEF* token needs the address of the new symbol. After being passed through all modules, the resulting stream of expressions is written back to the output FIFO.

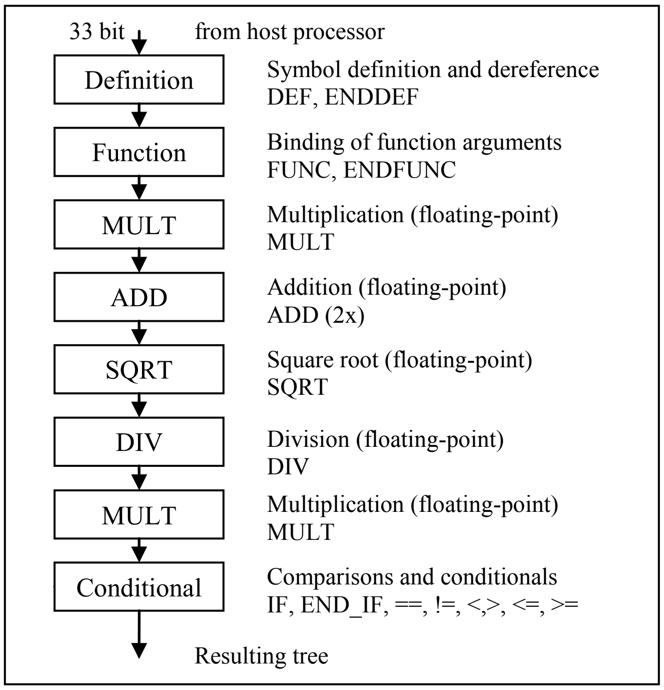

33 bit
from host processor
Definition
Symbol definition and dereference
DEF, ENDDEF
Function
Binding of function arguments
FUNC, ENDFUNC
MULT
Multiplication (floating-point)
MULT
ADD
Addition (floating-point)
ADD (2x)
SQRT
Square root (floating-point)
SQRT
DIV
Division (floating-point)
DIV
MULT
Multiplication (floating-point)
MULT
Conditional
Comparisons and conditionals
IF, END_IF, ==, !=, <,>, <=, >=
Resulting tree

Fig. 3: Structure of the Stream Processor.

4.2 Pipeline

In the ALU of a conventional processor, mostly all operations are executed each clock cycle and a multiplexer selects the desired result. Our design arranges the functions in a chain, so that intermediate results can be passed directly to the next stage without being written into a register file. As a consequence, all modules could operate in one cycle leading to a potentially much higher utilization of the arithmetic units.

To match as many rules as possible in one step, the order of operations in the expression should correspond to the order of modules. As a result, the optimal configuration may depend on the target application. For computer graphics, we usually deal with geometric calculations which involve a larger amount of linear functions. As a consequence, we choose to place the multiplication in front of the addition, so that a dot product can be calculated in one pass. Square-root and division are located at the end, since they are scalar functions that often take the result of linear functions. For communication, a simple bus, containing the token (33 bit) and a valid bit (1 bit), is employed, so that the same concept can be also adapted to other applications by reordering or replacing stages.

4.3 Symbols

The definition and insertion of symbols is handled by the symbol-table which is located in the *Definition* stage. It has a small RAM that is used to store the sequence of tokens associated with a symbol. On this level, the name of a symbol corresponds to the start address in the RAM. The *ENDDEF* is also written into the RAM to mark the end of a symbol sequence.

4.4 Conditionals

The conditional stage is used to implement the *IF/ELSE/ENDIF* and comparisons rules. It either discards or copies the token, depending on current condition. While the comparisons are pipelined, the *IF* itself only needs to test against zero. As a result, multiple nested *IF*-blocks can be processed in a single iteration by storing the previous condition on a small stack. It takes only two bits per nested *IF* to store the enable flag for the *if*-branch and the *else*-branch, so that a distributed RAM is sufficient to buffer a reasonable amount of conditions.

5. Results

To test the concept of streaming expression trees, we implemented the processing unit as part of a SoC on a Virtex4-FX20 from Xilinx (Fig. 5). The expression processor is connected to a PowerPC 405 via the *Fabric Co-processor Bus* (FCB). The PowerPC does not contain a floating-point unit, so that all floating-point operations are evaluated in software. Although the PowerPC runs at 300MHz and our ASIP is limited to 100MHz, we should expect an adequate speed-up. Our processing unit took 3432 Slices, 12 DSP48s and 1 BRAM (symbol-table). We used the Xilinx CoreGenerator to build the floating-point modules.

5.1 Empirical Tests

To evaluate the performance of our design, we measured some generic functions on an array of 4D vectors. For all functions, both a pure software solution and a hardware-accelerated version have been implemented. Table 2 shows the results of these tests with an array size of 512 vectors. In the first test, both vectors are added. The second test calculates the dot product of four-component vectors and a second constant vector. This is in particular interesting for graphic processing, since most external parameters like light directions or matrices remain the same for millions of invocations. At third, we evaluate the speed of vector normalization which requires two iterations. In the first iteration, the length of the vector is calculated and in the second step, the components are scaled by this value. Currently, the hardware does not support looping, so that we have to use the PowerPC to feed-back the values manually. The fourth test calculates the product of a constant matrix and a vector. Every vertex of a scene must be transformed into the camera-space of the viewer by a matrix-vector-multiplication, so that this operation should be fast. In all vector tests, we attained an increase in performance. The large number of vectors makes sure that there are enough overlapping operations. Since the pipeline is quite deep, we might have to wait for the result of the hardware, so that we expect a non-proportional scaling between software and hardware performance. Figure 6 shows the performance of the vector normalization test for the first 20 vector counts. A linear regression reveals a constant offset of $4\mu s$ for the hardware version. Surprisingly, the software variant has also some fixed costs, which might be caused by a cache-miss at the first time the floating-point library routines are executed.

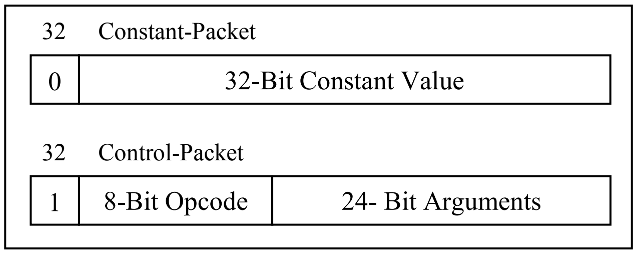


Fig. 4: The packet format of the hardware component.

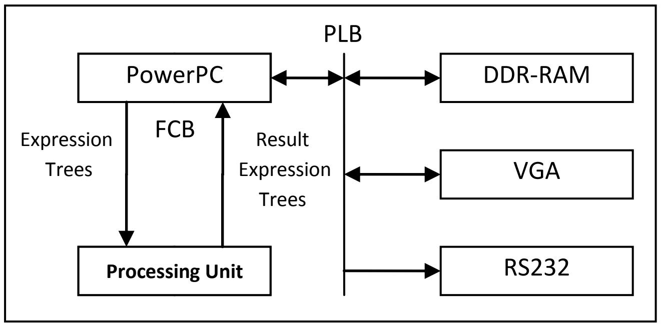


Fig. 5: System-on-Chip used as a test environment.

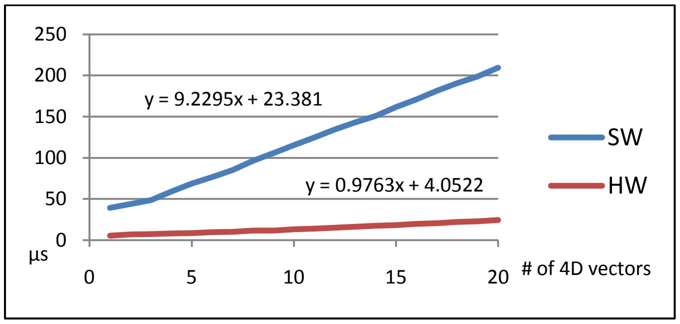


Fig. 6: Performance of Vector-Normalization.

5.2 Triangle Rendering

We implemented a triangle-rendering application in C++ that supports the rasterization of flat-shaded triangles. It consists mainly of two parts. First, the coordinates are transformed from the global word-space into the local camera-space and projected onto the screen. After that, the triangles are rasterized with a constant color, which requires only integer math. Therefore, we choose to accelerate the transformation of coordinates and normal vectors. As a result,

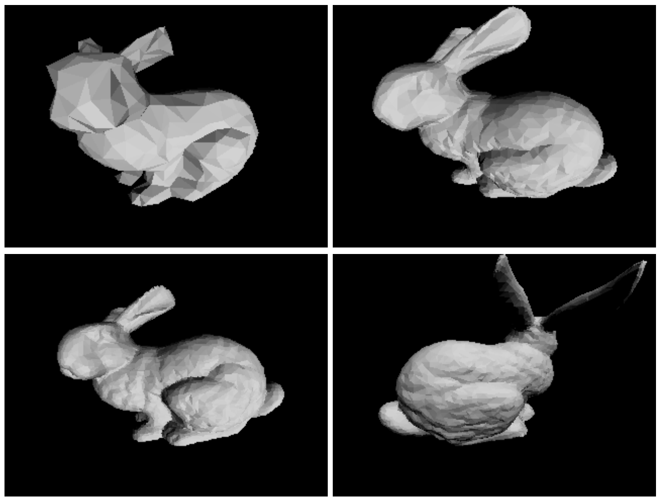


Fig. 7: Screenshots of Rendering Scenes.

Table 2: Time per frame (software and hardware rendering).

Test	SW	HW	SW/HW
Vector Tests(512)			
Vector Addition	2936 ţs	445 ţs	6.60
Dot-Product	1862 ţs	164 ţs	11.36
Normalization	4860 ţs	570 ţs	8.53
Vector-Matrix-Multiplication	8373 ţs	520 ţs	16.10
Triangle-Rendering			
Bunny 1K	59 ms	44 ms	1.34
Bunny 5K	173 ms	100 ms	1.73
Bunny 10K	315 ms	170 ms	1.85
Raytracing			
Plane without lighting	4200 ms	1890 ms	2.22
Plane, 1 Light	5752 ms	3433 ms	1.68
Plane, 1 Sphere, 1 Light	7827 ms	5483 ms	1.44
Plane, 1 Sphere, 1 Light	8892 ms	6868 ms	1.29
Plane, 2 Spheres, Shadows	10661 ms	8954 ms	1.19
Plane, 3 Spheres, Shadows	13279 ms	11773 ms	1.13
Plane,3 Spheres, Shadows, Reflection	15641 ms	13879 ms	1.13

we can use more and smaller triangles so that the low-quality flat-shading becomes less relevant. We tested three versions of the Stanford Bunny [1] consisting of 1K, 5K and 10K triangles. Figure 7 shows screenshots of these models at a resolution of 320x240. The factor between hardware and software, listed in Table 2, is much lower than the speed-up of the generic tests. This is causes by the fact, that we used the ASIP only for some parts of the program. Our solution focuses on interoperability and allows for a gradual move from software to hardware. It can be seen, that the test with 10K triangles achieves a higher performance increase than the scene with 1K triangles. The fixed costs of the pipeline are irrelevant in this case, because we send the vertices in blocks of 512 elements. Therefore, the work of the CPU must depend mainly on the number of visible pixels, which remains nearly the same.

5.3 Raytracing

Raytracing takes the opposite approach to rasterization. Instead of drawing the pixels of the triangles directly, we create a line from the camera through every pixel and calculate the nearest intersection of the scene with this line. Special effects like mirrors or shadows can be implemented by tracing additional rays. Further, we are not limited to triangles, but can use every primitive that supports an intersection test. On the other hand, ray-tracing is expensive on a system without an FPU, because most calculations involve floating-point operations. We measured the rendering time of a test scene consisting of a plane with a checkerboard pattern and up to three spheres with and without lighting (Fig. 8). In the last test, the spheres are reflective. Some parts of the intersection equations, vector normalizations and matrix-vector-multiplication have been moved to the ASIP.

[1]http://www-i8.informatik.rwth-aachen.de/old-site/teaching/ss05/praktikum_sfx

Let g be a ray and E be a plane defined by the following equations:

$$g(t) = S + t \cdot R$$

$$E = \{(x, y, z\} \in R^3 : Ax + By + Cz + D = 0\}$$

Then, the intersection of g and E is given by:

$$t = -\frac{A \cdot S_x + B \cdot S_y + C \cdot S_z + D}{A \cdot R_x + B \cdot R_y + C \cdot R_z}$$

The result t marks the position on the line g at which ray and plane intersect. Although the expression is rather complex, it can be evaluated almost completely in a single pass. In our case, the final minus is calculated on the CPU, since inverting a floating-point number is cheap. First, the numerator of the fraction is encoded as:

A	S_x	*	B	S_y	*	+	C	S_z	*	D	+	+

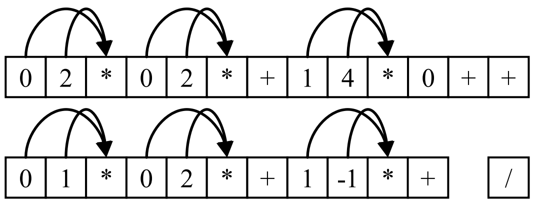
The denominator corresponds to the sequence:

A	R_x	*	B	R_y	*	+	C	R_z	*	+

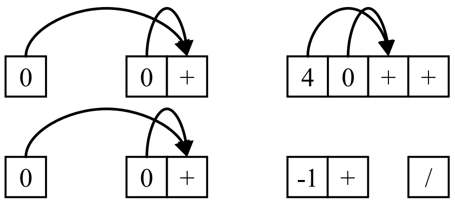
Both are divided by appending the division token:

A	S_x	*	B	S_y	*	+	C	S_z	*	D	+	+
A	R_x	*	B	R_y	*	+	C	R_z	*	+		/

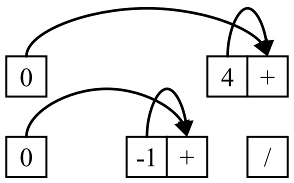
To demonstrate the individual steps of the calculation, we insert real values. The structure of the flattened tree is illustrated by arrows. The plane should be aligned to the Z-axis and go through the origin. Hence, we set (A, B, C, D) to $(0, 0, 1, 0)$. The ray starts at the point $(2, 2, 4)$ and has the direction $(1, 2, -1)$. The following sequence is sent to the device:



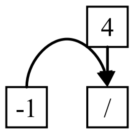
In the first stage of the ALU, the multiplication rule is applied, which leads to the following intermediate result:



After that, the sequence passes the first adder stage, so that the addition can be evaluated:



The second adder stage reduces the remaining additions:



The final result is left on the stream: $\boxed{-4}$

We verify the result by inserting 4 into the equation of the ray: $g(4) = (2, 2, 4) + 4 \cdot (1, 2, -1) = (6, 10, 0)$.

Obviously, the intersection $g(4)$ is also on the plane. Performance results can be also found in Table 2: Contrary to the triangle rendering, the speed-up is lower for the more complex scenes. The intersection tests for primary rays, which start directly at the camera, can be easily overlapped by processing a set of pixels in parallel. The secondary rays for shadows and reflections have not been pipelined completely, so that we can see a performance degradation. As a result, our solution can be used to improve the speed of existing programs, but for optimal performance, the program should be adapted.

6. Conclusion

We have shown that expression trees can be used to communicate work-items between hardware and software modules. In addition, serialized expressions trees are evaluated in a hardware pipeline by applying local replacement rules.

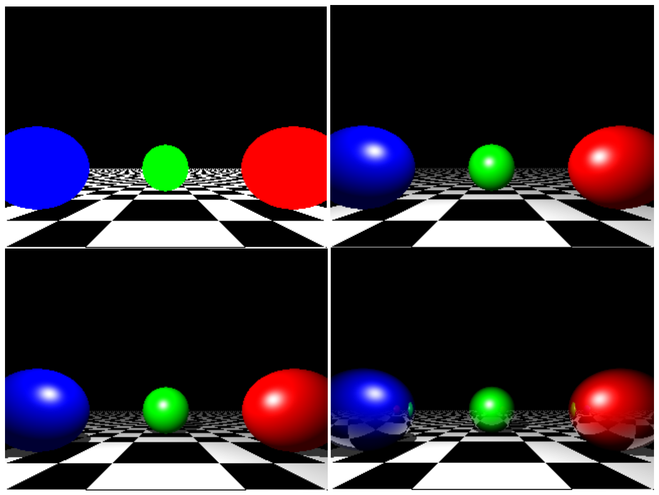


Fig. 8: Screenshots of Raytracing Scenes.

Thus, our stream processor could successfully accelerate the performance of two compute-intensive rendering programs. In contrast to a fixed accelerator built for a special purpose, the expression tree processor can be adapted to different applications. The format of the expression trees allows us to describe program fragments in a platform-independent way, making this technique most useful for heterogeneous environments. However, to employ the full capabilities of our system, we need a compiler that extracts the trees automatically from a high-level language.

References

[1] A. A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *Computer*, vol. 38, pp. 63–69, 2005.

[2] L. Benini and G. De Micheli, "Networks on chips: A new soc paradigm," *Computer*, vol. 35, pp. 70–78, January 2002.

[3] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews, "Enabling a uniform programming model across the software/hardware boundary," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 89–98.

[4] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden, "Programming models for hybrid fpga-cpu computational components: A missing link," *IEEE Micro*, vol. 24, pp. 42–53, July 2004.

[5] P. Mahr, C. Lorchner, H. Ishebabi, and C. Bobda, "Soc-mpi: A flexible message passing library for multiprocessor systems-on-chips," *Reconfigurable Computing and FPGAs, International Conference on*, vol. 0, pp. 187–192, 2008.

[6] G. M. Bierman, E. Meijer, and M. Torgersen, "Lost in translation: formalizing proposed extensions to c#," *SIGPLAN Not.*, vol. 42, pp. 479–498, October 2007.

[7] S. Pestov, D. Ehrenberg, and J. Groff, "Factor: a dynamic stack-based programming language," in *Proceedings of the 6th symposium on Dynamic languages*, ser. DLS '10. New York, NY, USA: ACM, 2010, pp. 43–58.

[8] M. von Thun, "A rewriting system for joy," 1996, available from the author.

[9] H. Schmit, B. Levine, and B. Ylvisaker, "Queue machines: Hardware compilation in hardware," in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 152–.

Data-Transfer-Aware Memory Allocation for Dynamically Reconfigurable Accelerators in Heterogeneous Multicore Processors

Y. Ohbayashi, H. M. Waidyasooriya, M. Hariyama and M. Kameyama
Graduate School of Information Sciences, Tohoku University
Aoba 6-6-05, Aramaki, Aoba, Sendai, Miyagi,980-8579, Japan
Email: {y-ohba@kameyama., hasitha@kameyama., hariyama@, kameyama@}ecei.tohoku.ac.jp

Abstract—*Accelerator cores in low-power heterogeneous multicore processors have multiple memory modules to enable parallel data access. Recent low-power processors contain address generation units (AGUs) for fast address generation. To reduce the core-area, small functional units such as adders and counters are used in AGUs. Such small functional units make it difficult to implement complex addressing patterns without duplicating the data among multiple memory modules. The data-duplication wastes the memory capacity and increases the data transfer time significantly. This paper proposes a method to remove the memory duplication and to increase the degree of parallelism. To verify the effectiveness of this method, we use window-based media processing which is widely used in many applications. According to the evaluation, the proposed method reduces the total processing time by 14% to more than 85% compared to the previous works.*

Keywords: Heterogeneous multicore, memory allocation, dynamic reconfiguration, multi-context FPGA

1. Introduction

Recently, there is a huge demand for the media processing on mobile devices such as mobile phones, cameras and vehicles. The media processing applications are getting very complicated due to the addition of many different tasks such as filtering, matching, object detection, etc. This has increased the demand for a large computational capability and also a low power consumption. Using heterogeneous multicore processors is a very promising way to meet these demands. They contain different cores such as CPUs and accelerators in the same chip. To reduce the power consumption and to integrate many cores in a single chip, the area of each core is reduced. Therefore, each core has a very small memory capacity and a small number of processing elements (PEs). Moreover, many accelerators are dynamically reconfigurable so that the same resources are shared in different time slots. If the tasks of an application are correctly mapped to the most suitable core, all the cores work together to increase the overall performance. Examples of heterogeneous multicore processors are [1] and [2].

One major problem in the heterogeneous multicore processors is the data-transfer bottlenecks. Many heterogeneous multicore processors contain a hierarchical memory structure as shown in Fig.1. It contains a large memory module (global memory module) placed outside the accelerator core and several small memory modules (local memory modules) placed inside the accelerator core. Since the memory capacities of the local memory modules are small, data are copied from the global memory module to the local memory modules many times. To access the local memory modules, the accelerator core employs address generation units (AGUs) as shown in Fig.1. To decrease the area of the accelerator, the AGUs contain simple hardware units such as adders and counters. Therefore the AGUs implement only the most common memory access patterns such as linear access and stride access. The relationship between the memory address and the control step (or time) is called the “addressing function”. Many media processing applications contain very complex memory access patterns. To implement such access patterns using simple AGUs, the same data should be transferred many times into different addresses of the local memory modules. Transferring the same data multiple times is called “data-duplication problem”. Due to this, the amount of data transferred to the local memory modules is increased and the data-transfer time becomes much larger than the computation time for many applications.

To solve this problem, address-function-constrained mem-

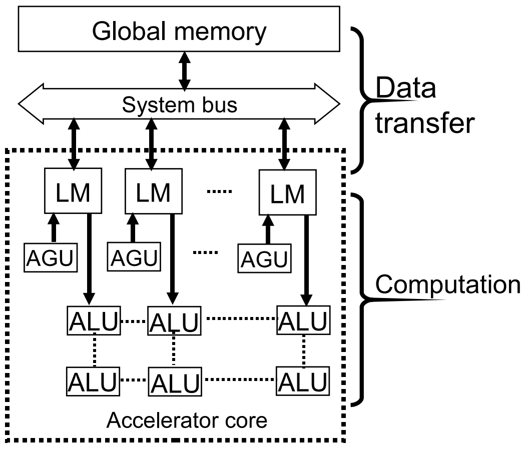


Fig. 1: Hierarchical memory structure

ory allocations are proposed recently [3], [4], [5]. A memory allocation for FIR filters and their implementation on a dynamically reconfigurable accelerator is proposed in [5]. However, the applications are limited to FIR filters and it does not consider a processing time minimization. A memory allocation for window-based media processing under addressing function constraint is proposed in [3] and [4]. The method in [3] and [4] allocates the data in the local memory modules in such a way that they can be accessed using simple addressing functions. Although this method reduces the data-transfer time significantly, it cannot completely solve the data-duplication problem. Therefore, the data-transfer time is more than half of the total processing time for many applications.

This paper proposes a temporal memory allocation for window-based media processing. This work is distinguished from the earlier works by considering both the spatial and the sequential relationships between the data and the memory addresses, while the conventional memory allocation [4] considers only the spatial relationship. In the proposed memory allocation, only the required data are transferred to the local memory modules. The data are allocated to the local memory addresses that contain obsolete data which are already processed and are not required in later calculations. Therefore, the transferred data amount is minimized and the data-transfer time is reduced. It uses both pixel-parallelism and window-parallelism simultaneously, so that the computational time is also reduced. To verify the effectiveness of this method, we use window-based media processing which is widely used in many applications such as stereo matching [6], filter computations, etc. It involves with a large amount of data and also requires complex memory access patterns. According to the evaluation, the proposed method reduces the total processing time by 14% to more than 85% compared to the method in [4].

2. Heterogeneous multicore processor architecture

In this paper, we use the heterogeneous multicore processor called “RP1” proposed in [2]. It has 4 CPU cores and 2 FE-GA (Flexible Engine/Generic ALU Arrays) accelerator cores. All the cores are connected by a bus called “SuperHyway”. An off-chip SDRAM is connected to the processor through the SuperHyway. Figure 2 shows the architecture of the FE-GA accelerator. It has an array of 32 PEs called “ALU” cells and “MLT” cells. The FE-GA has 10 local memory modules of 4KByte each. Each memory module has AGUs as shown in Fig.2.

The AGUs are very useful since they significantly decrease the addresses generation time. Since AGUs do the address generation, the PEs are used only for the data processing and that decreases the processing time. To reduce the area of the accelerator core, AGUs contain only simple

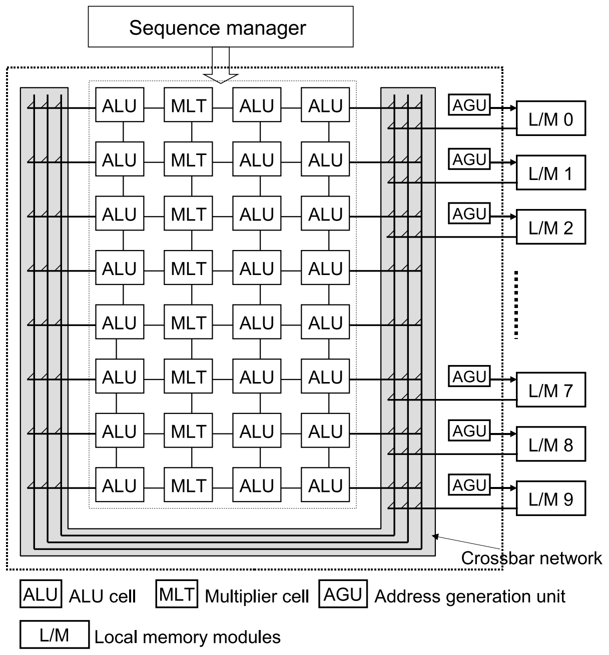


Fig. 2: Block diagram of FE-GA

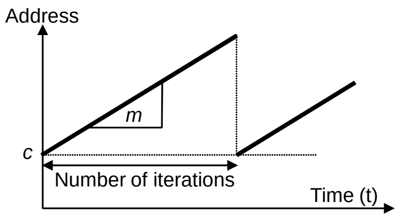


Fig. 3: Addressing function

hardware such as adders and counters. Therefore, the number of addressing patterns generated in AGUs is limited to the most common addressing patterns. The relationship between the time and the control step (clock cycle) is called the “addressing function”. In FE-GA, the addressing functions are limited to a repetition of linear functions as shown by Eq.(1).

$$A_M = m \times t - m \times N_I \times \left\lfloor \frac{t}{N_I} \right\rfloor + c \qquad (1)$$

The parameters m, t, c and A_M are the *address increment*, the *control step* (or clock cycle), the *base address* and the address of memory module M respectively. The parameter N_I is called the *number of iterations* and it determines how many clock cycles this addressing function works. After the addressing function executes for N_I clock cycles, the address returns to the *base address* as shown in Fig.3. In each context of the FE-GA, we need to set these 3 parameters: m, c and A_M. Therefore, it is possible to change these parameters dynamically to access different parts of the memory. Note that, although Eq.(1) contains division and multiplication, we do not require any multipliers or dividers to implement this addressing function. The multiplications and divisions are done by repeated additions. We have not included the details of implementing the addressing function, since it is not in the scope of this paper.

The FE-GA contains 256 contexts which are dynamically reconfigurable. The sequence manager shown in Fig.2 controls the dynamic reconfiguration. In each context, we can change the operations in PEs, AGUs and also the interconnection network. The dynamic reconfiguration can be used to dynamically change the addressing function to generate more complex addressing patterns. In this paper, we change the parameter c dynamically to access different address ranges from the memory. This modified addressing function is given by Eq.(2).

$$A_M = m \times t - m \times N_I \times \left\lfloor \frac{t}{N_I} \right\rfloor + c\,(t) \qquad (2)$$

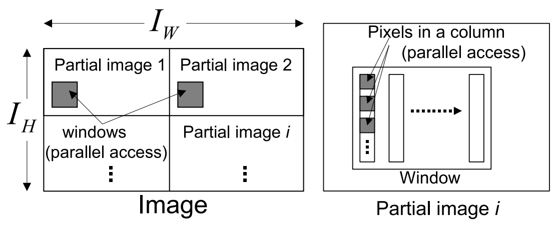
3. Memory allocation

3.1 Targeted application : Window-based media processing

This paper proposes a memory allocation for window-based media processing to reduce the data-transfer time. The window-based media processing is used in many applications such as stereo matching [6], optical flow extraction [7], scale-invariant feature transformation (SIFT) [8], histogram of oriented gradients (HOG) [9], matrix processing, filtering, etc. It processes the data in blocks and those blocks are called windows. Windows are overlapped each other so that there is a huge shared area between two windows. Therefore, it is very important to maximize the data sharing to minimize the amount of data-transfers. At the same time, it is important to allow parallel data access while satisfying the address function constraint. We consider both pixel-parallelism and window-parallelism for the data access.

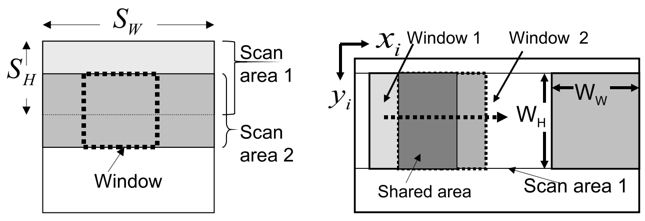
3.2 Window-parallel pixel-parallel scheduling

This section explains how the image data are arranged and how those are accessed in window-based media processing. Since the local memory modules in the accelerator have a limited capacity, we partition the image into many small partial images as shown in Fig.4(a). The windows belong to multiple partial images are accessed in parallel. This parallelism is called window-parallelism. The pixels in a window are accessed in pixel-parallel column-serial manner as shown in Fig.4(b). The data in a column are accessed in parallel and this parallelism is called pixel-parallelism. Figure 5 shows how the windows inside the partial image i are accessed. Each partial image contains multiple scan areas as shown in Fig.5(a). After the first scan area is accessed the next scan area, which is one pixel bellow, is accessed. The data in scan areas are accessed by sliding a window from left-to-right as shown in Fig.5(b).



(a) Window-parallel access (b) Pixel-parallel access

Fig. 4: Window-parallel and pixel-parallel access



(a) Scan area access sequence (b) Window access of scan area 1

Fig. 5: Window access inside the partial image i

3.3 Temporal memory allocation

Temporal memory allocation considers both the spatial and the sequential relationships between the image data and the memory address. In the proposed memory allocation, the spatial relationship is the relationship between the image data coordinates and the address A_M in the local memory module M. The sequential relationship is the relationship between the image data coordinates and the time sequence that the data are allocated. This section defines the temporal memory allocation of a partial image. The size of the partial image and the degree of parallelism is given.

The objective of the memory allocation is to find a feasible solution for the partial image i that satisfies the following 3 conditions.

Condition 1 :	Given degree of parallelism (window-parallelism W_P and pixel-parallelism P_P are explained in section 3.2) P_P and W_P must satisfy the relationship given by Eqs.(3) and (4) where, C_M is a natural number and W_H is the window height. The maximum degree of parallelism available in the accelerator is P_{max}
Condition 2 :	No data-duplication for a partial image (Data-duplication is explained in section 1)
Condition 3 :	Addressing function (Equation (2) is explained in section 2)

$$P_P \times C_M = W_H \qquad (3)$$

$$P_P \times W_P \leq P_{max} \tag{4}$$

To explain the temporal memory allocation, we first discuss the spatial relationship. The spatial relationship between the data and memory address is defined by Eqs.(5) and (6). Equation (5) gives the memory module number M that the pixel (x_i, y_i) is allocated.

$$M = y_i \text{ MOD } P_P + (i \text{ MOD } W_P) \times P_P \tag{5}$$

Note that, we give a number starting from 0 to all the local memory modules. The degree of pixel-parallelism, the degree of window-parallelism, the horizontal and the vertical coordinates of the partial image i are denoted by P_P, W_P, x_i and y_i respectively. Equation (6) gives the address A_M of the memory module M that the pixel with the coordinates (x_i, y_i) is allocated.

$$A_M = x_i \times C_M + \left\lfloor \frac{y_i \text{ MOD } W_H}{P_P} \right\rfloor \tag{6}$$

The sequential relationship between the pixel data coordinates and the time sequence is discussed in this paragraph. The data transfer based on the proposed memory allocation is done sequentially. When the processing of one scan area is finished, the data of another scan area is transferred. To maximize the data sharing, only the difference between two scan areas is transferred. Moreover, the new data are overwritten to the memory addresses with obsolete data which are not required for further processing. This method minimizes the data-transfer time since there is no data-duplication. It also optimizes the memory capacity of the CRAMs since new data are overwritten to the memory addresses that have obsolete data.

Figure 6 shows the sequential data transfer. In each sequence, the data belong to a scan area is transferred to the local memory modules. In the first sequence, all the pixel data belongs to the first scan area are transferred. The coordinates of the data are given by Eq.(7).

$$\begin{aligned} 0 \leq x_i < S_W \\ 0 \leq y_i < S_H \end{aligned} \tag{7}$$

In the second sequence, only the difference of the first and second scan areas is transferred. The coordinates of the transferred data are given by Eq.(8).

$$\begin{aligned} 0 \leq x_i < S_W \\ S_H \leq y_i < S_H + 1 \end{aligned} \tag{8}$$

Similarly, the difference between two scan areas is transferred in each sequence. This process continues for all the scan areas in a partial image.

The following example explains the proposed memory allocation. Figure 7(a) shows a partial image with the size 5×6 that have 3 scan areas. The data in the scan areas are accessed by a window of size 4×4. Figure 7(b) shows the data transfers based on the temporal memory allocation. In

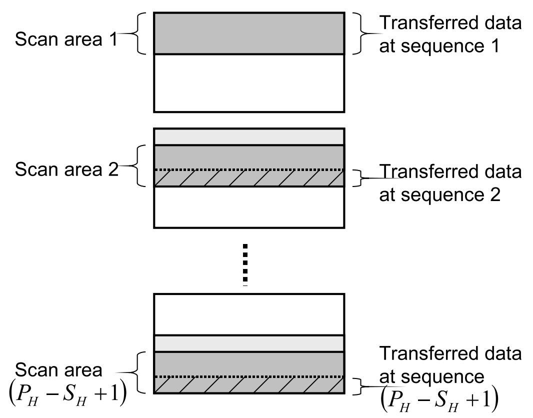
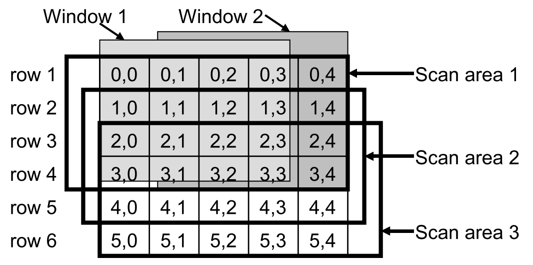
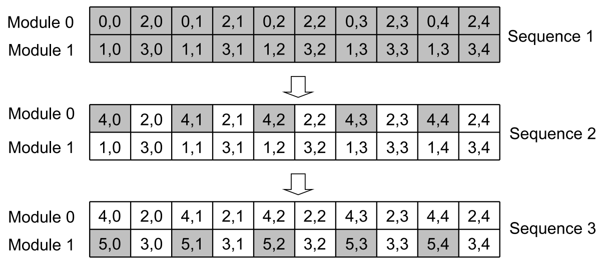

Scan area 1
Transferred data at sequence 1
Scan area 2
Transferred data at sequence 2
Scan area $(P_H - S_H + 1)$
Transferred data at sequence $(P_H - S_H + 1)$

Fig. 6: Temporal memory allocation



(a) Partial image and scan areas



Sequence 1

Module 0	0,0	2,0	0,1	2,1	0,2	2,2	0,3	2,3	0,4	2,4
Module 1	1,0	3,0	1,1	3,1	1,2	3,2	1,3	3,3	1,3	3,4

Sequence 2

Module 0	4,0	2,0	4,1	2,1	4,2	2,2	4,3	2,3	4,4	2,4
Module 1	1,0	3,0	1,1	3,1	1,2	3,2	1,3	3,3	1,4	3,4

Sequence 3

Module 0	4,0	2,0	4,1	2,1	4,2	2,2	4,3	2,3	4,4	2,4
Module 1	5,0	3,0	5,1	3,1	5,2	3,2	5,3	3,3	5,4	3,4

(b) Sequential data transfer

Fig. 7: Example of the memory allocation

the first sequence, all the data belong to the scan area 1 is transferred to the two memory modules. In the sequence 2, the difference between the scan areas 1 and 2, that is the row 5 in the partial image shown in Fig.7(a), is transferred. Since the scan area 2 does not contain the data in the row 1 of the partial image, those data are overwritten by the data of row 5. Similarly, in the sequence 3, the difference between the scan areas 2 and 3 is transferred. The data of row 2 are overwritten by the data of row 6.

The following equations show the pixel-parallel scheduling (discussed in section 3.2) of the allocated data that belongs to a scan area.

$$x_i = \left\lfloor \frac{t_S}{C_M} \right\rfloor - \left\lfloor \frac{t_S}{W_W \times C_M} \right\rfloor (W_W - 1) \tag{9}$$

$$y_i \text{ MOD } W_H = (t_S \text{ MOD } C_M) \times P_P + b \quad \text{such that } b \in \{0, 1, \cdots, P_P - 1\} \tag{10}$$

Equations (9) and (10) give the relationships between the time (or the control step) and x_i, y_i coordinates respectively. Note that, since multiple pixel data of a single column are accessed in parallel, Eqs.(9) and (10) shows multiple y_i coordinates for a single x_i coordinate for the same t_S. Also note that, the time t_S is the time of accessing the scan area S. The value t_S is different for each scan area. When the window-parallelism W_P is given, we access W_P partial images in parallel. In each partial image, pixel-parallel data access is done according to Eqs.(9) and (10).

3.4 Satisfaction of the memory allocation conditions

This section shows that the proposed memory allocation satisfies the 3 conditions explained in section 3.3. According to Eq.(5), the data in rows are allocated to P_P number of memory modules. Therefore, the pixel-parallelism can be achieved. The data in partial images are allocated to W_P sets of P_P memory modules. Therefore, the window-parallelism is achieved. This satisfies the condition 1 explained in section 3.3.

Since only the difference between two scan areas are transferred, there is no data-duplication. This satisfies the condition 2. Note that, the condition 2 (no data-duplication) is satisfied only for a partial image. Depend on the manner that the image is partitioned, the partial images can be overlapped. Therefore, a small amount of data has to be duplicated among multiple partial images.

To obtain the addressing function required for the memory access, we substitute Eqs.(9) and (10) to Eq.(6). After that, we get Eq.(11).

$$\begin{aligned} A_M = C_M \times \left\lfloor \frac{t}{C_M} \right\rfloor - \\ C_M \times (W_W - 1) \times \left\lfloor \frac{t}{W_W \times C_M} \right\rfloor + \\ \left\lfloor \frac{((t \text{ MOD } C_M) \times P_P + b) \text{ MOD } W_H}{P_P} \right\rfloor \end{aligned} \tag{11}$$

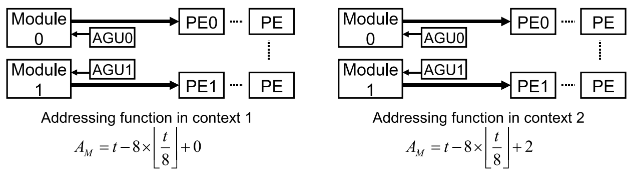
Equation (11) can be rewritten as Eq.(14) using Eqs.(12) and (13).

$$C_M \times \left\lfloor \frac{t_S}{C_M} \right\rfloor + (t_S \text{ MOD } C_M) = t_S \tag{12}$$

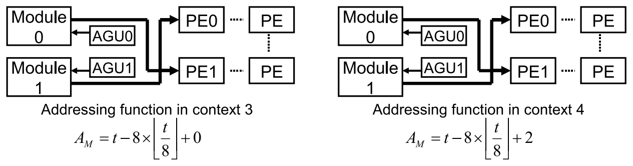
$$\frac{b}{P_P} < 0 \quad \text{since } b < P_P \tag{13}$$

$$\begin{aligned} A_M = t_S - W_W \times C_M \left\lfloor \frac{t_S}{W_W \times C_M} \right\rfloor + \\ C_M \times \left\lfloor \frac{t_S}{W_W \times C_M} \right\rfloor \end{aligned} \tag{14}$$

Equation (14) is the addressing function of this memory allocation which gives the relationship between the control step t_S and the memory address A_M. We substitute m for 1, N_I for $C_M \times W_W$ and $c(t)$ for $C_M \times \left\lfloor \frac{t_S}{W_W \times C_M} \right\rfloor$ in Eq.(14). Then, Eq.(14) equals Eq.(2) and it shows this memory allocation satisfies the condition 3.


Module 0
AGU0
Module 1
AGU1
PE0
PE
PE1
PE
Addressing function in context 1
$A_M = t - 8 \times \left\lfloor \frac{t}{8} \right\rfloor + 0$
Addressing function in context 2
$A_M = t - 8 \times \left\lfloor \frac{t}{8} \right\rfloor + 2$

(a) Computation of scan area 1 and 3


Module 0
AGU0
Module 1
AGU1
PE0
PE
PE1
PE
Addressing function in context 3
$A_M = t - 8 \times \left\lfloor \frac{t}{8} \right\rfloor + 0$
Addressing function in context 4
$A_M = t - 8 \times \left\lfloor \frac{t}{8} \right\rfloor + 2$

(b) Compution if scan area 2

Fig. 8: Example of the data access and computation

Figure 8 shows an example of implementing the addressing function using different contexts in the dynamically reconfigurable accelerator. This example uses the memory allocation result given in Fig.7. According to Fig.7, each scan area contains two windows. The computations of the first and second windows of the scan area 1 is assigned to the contexts 1 and 2 respectively as shown in Fig.8(a). Note that, the base address of the addressing function is different in the contexts 1 and 2. Similarly, the computation of the scan area 2 is done by contexts 3 and 4. The interconnection network between the memory modules and PEs in scan area 2 is different from that in scan area 1. The addressing functions and the interconnection network of the scan area 3 are the same as those in scan area 1. Therefore, we reduce the number of contexts by assigning the computations of scan area 3 to the contexts 1 and 2.

4. Evaluation

We use Hitachi RP1 heterogeneous multicore processor [2] for the evaluation. The details of the processor architecture is explained in section 2. This evaluation is done by using one SH-4A CPU core and one FE-GA accelerator core. The total processing time consists of three major components: the data-transfer time, the computation time and the control time as shown in Fig.9. Note that the control-time refers to the initialize of accelerator and initial configuration load as shown in Fig.9.

Figure 10 shows the components of the total processing time in the proposed method and the method in [4]. The image size is 640×480. In the proposed method, the optimal values of P_W, P_H, P_P and W_P that minimize the total processing time are used. In Fig.10, the horizontal axis shows the window sizes. For each window size, the

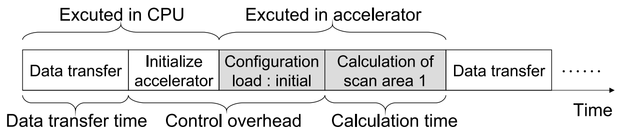


Fig. 9: Components of the total processing time

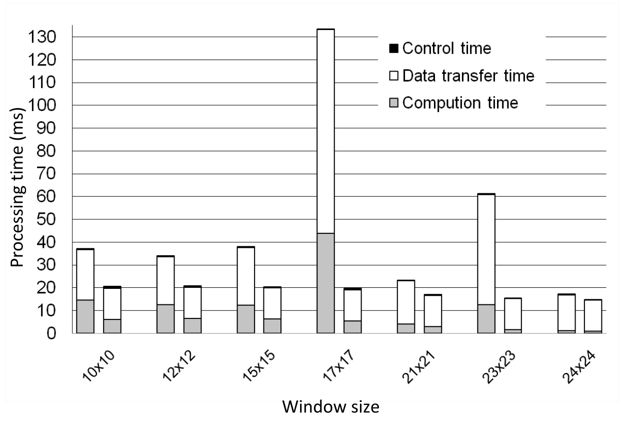


Fig. 10: Comparison of the processing time components

processing time of the conventional method [4] and the proposed method are shown by the bars to the left and to the right respectively. According to the results, the data-transfer time is reduced by 13% to more than 84%. This is due to the proposed memory allocation with no data-duplication. The computation time is also reduced by 15% to more than 87%. We explain this using Fig.11. As shown in Fig.11(a), the method in [4] uses only the pixel-parallelism so that the degree of parallelism is 3. As shown in Fig.11(b), the proposed method uses both pixel and window parallelisms simultaneously. Therefore, P_P is 2, W_P is 2 and the total degree of parallelism is increased to 4. Higher degree of parallelism reduces the computation time.

The data-transfer time is reduced significantly when the window-sizes contain prime numbers such as 17×17. In such window sizes, the degree of pixel-parallelism P_P is 1. For such small values of P_P, the conventional method [4] has a high degree of data-duplication. However, the proposed method does not have any data-duplication. Therefore, the data-transfer time and the total processing time are decreased significantly. In real world media processing, we find applications such as gauss filters that have the window sizes with prime numbers. Therefore, the proposed method is very useful in such applications.

Table 1: Total processing time comparison

Example	Total processing time		
	method in [4] (ms)	proposed method (ms)	reduction (%)
$W_W = W_H = 10$	37.04	20.50	44.66
$W_W = W_H = 12$	34.06	20.86	38.77
$W_W = W_H = 15$	38.07	20.49	46.19
$W_W = W_H = 17$	133.54	19.64	85.29
$W_W = W_H = 21$	23.37	16.92	27.58
$W_W = W_H = 23$	61.29	15.43	74.82
$W_W = W_H = 24$	17.30	14.77	14.58

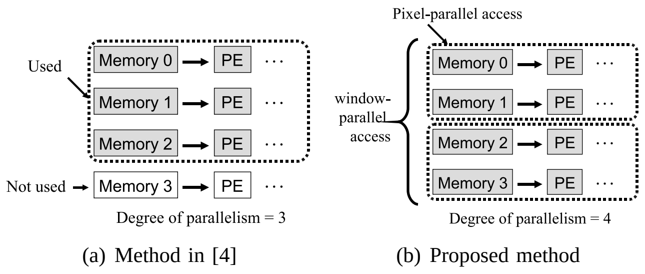


Fig. 11: Degree of parallelism

Table 1 shows the measured total processing time in the proposed method and the method in [4]. Although the control time is increased as shown in Figure 10, the total processing time is decreased by 14% to more than 85% in the proposed method. The reasons for this are the smaller computation time due to the higher degree of parallelism and the smaller data-transfer time due to not duplicating the data.

5. Conclusion

This paper proposed a temporal memory allocation for window-based media processing. In the proposed memory allocation, only the required data are transferred to the local memory modules. Therefore the data are not duplicated unlike the conventional approaches. Therefore, the data-transfer time is reduced significantly. The proposed memory allocation allows us to use both the pixel-parallelism and the window-parallelism. Therefore, the computation time is reduced. The proposed method has a larger control overhead compared to that of the conventional method [4], so that the control time is increased. However, the reduction of the data-transfer time and the computation time is much larger than the increase in the control time. As a result, the total processing time is reduced by 14% to more than 85%.

References

[1] H. Kondo, M. Nakajima, N. Masui, S. Otani, N. Okumura, Y. Takata, T. Nasu, H. Takata, T. Higuchi, M. Sakugawa, H. Fujiwara, K. Ishida, K. Ishimi, S. Kaneko, T. Itoh, M. Sato, O. Yamamoto and K. Arimoto, “Design and Implementation of a Configurable Heterogeneous Multicore SoC With Nine CPUs and Two Matrix Processors”, IEEE Journal of Solid-State Circuits, Vol.43, No.4, pp.892-901, 2008.

[2] H. Shikano, M. Ito, M. Onouchi, T. Todaka, T. Tsunoda, T. Kodama, K. Uchiyama, T. Odaka, T. Kamei, E. Nagahama, M. Kusaoke, Y. Nitta, Y. Wada, K. Kimura, H. Kasahara", “Heterogeneous Multi-Core Architecture That Enables 54x AAC-LC Stereo Encoding”, IEEE Journal of Solid-State Circuits, Vol.43, No.4, pp.902-910, 2008

[3] H. M. Waidyasooriya, M. Hariyama and M. Kameyama, “Acceleration of Optical-Flow Extraction Using Dynamically Reconfigurable ALU Arrays”, International Conference on Engineering of Reconfigurable Systems and Algorithms, pp.291-294, 2009
[4] H. M. Waidyasooriya, M. Hariyama and M. Kameyama, “Memory Allocation for Window-Based Image Processing on Multiple Memory Modules with Simple Addressing Functions”, IEICE Trans. Fundamentals, Vol.E94-A, NO.1, pp.342-351, 2011
[5] R. Tamura, M. Honma, N. Togawa, M. Yanagisawa, T. Ohtsuki and M. Satoh, “FIR Filter Design on Flexible Engine Generic ALU Array and Its Dedicated Synthesis Algorithm”, IEEE Asia Pacific Conference on Circuits and Systems, pp.701-704, 2008.
[6] M. Hariyama, H. Sasaki, and M. Kameyama, “Architecture of a Stereo Matching VLSI Processor Based on Hierarchically Parallel Memory Access”, IEICE Trans. Inf. and Syst., Vol.E88-D, No.7, pp.1486-1491, 2005
[7] S. Lee, M. Hariyama and M. Kameyama, “An FPGA-Oriented Motion-Stereo Processor with a Simple Interconnection Network for Parallel Memory Access”, IEICE Trans. Inf. Syst., Vol.E83-D, No.12, pp.2122-2130 2000.
[8] D.G. Lowe, “Object recognition from local scale-invariant features”, IEEE International Conference on Computer Vision, pp.1150 - 1157, Vol.2, 1999.
[9] N. Dalal and B. Triggs “Histograms of oriented gradients for human detection”, IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp.886 - 893, Vol.1, 2005.

Placement and Routing Algorithm for Pipeline Architecture

M.Koezuka, A.Kuroda, K.Funaoka, H.Matsuzaki, T.Yoshikawa, and S.Asano
Corporate R&D Center Toshiba Corporation, Kawasaki, Kanagawa, JAPAN
E-mail: {mayuko.koezuka, akira3.kuroda, kenji.funaoka, hidenori.matsuzaki, takashi.yoshikawa, shigehiro.asano}@toshiba.co.jp

Abstract— *We present an algorithm that places and routes processing expressed in the form of a data flow graph in pipeline architecture. Pipeline architecture, which reduces wiring compared with array architecture, is a focus of attention as architecture that achieves small area and low power consumption. However, pipeline architecture has low degree of freedom of placement and routing (P&R) because it has far fewer data paths. Therefore, it is difficult to apply P&R algorithm for array architectures to pipeline architectures, because array architecture has high degree of freedom of P&R. The P&R algorithm for typical array architecture includes Simulated Annealing (SA). We propose a high-speed and high-solution-discovery-rate P&R algorithm for pipeline architecture that has low degree of freedom of wiring. In the evaluation, comparison of the proposed P&R algorithm (P_P&R) and SA revealed the superiority of P_P&R. Moreover, we verified the performance of P_P&R in the pipeline architecture, and showed the validity of the proposed functions.*

Keywords: pipeline, placement, routing, pruning

1. Introduction

Recently, high-definition and multifunctional mobile computers have been developed. These battery-powered mobile computers require large processing power with low power consumption and small area. Dynamically Reconfigurable Architecture (DRA) has been proposed to meet these needs in recent years [1]. A typical structure of DRA includes array architecture and pipeline architecture. Arithmetic Logic Units (ALUs) of array architecture are arranged in an array as shown in Fig.1, and connect interactively with the neighboring ALUs. Array architecture has many data paths compared with pipeline architecture and high degree of freedom of P&R because data can be transmitted to an arbitrary ALU, but the area grows and power consumption increases. On the other hand, ALUs of pipeline architecture connect ALUs as shown in Fig.2, and data may flow to the pipeline. Pipeline architecture has small area though it has low degree of freedom of P&R because wiring is less than array architecture wiring, and so destination ALUs are limited. In this paper, we solve the P&R problem for a pipeline architecture that has few data paths, small area and low power consumption. Architectures that use pipeline architecture include PipeRench [3], Sanyo's Reconfigurable

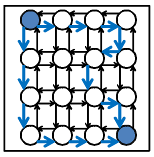


Fig. 1: Array Architecture

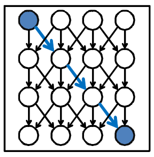


Fig. 2: Pipeline Architecture

Architecture [7] and FlexSword(TM) [8] [9]. The P&R algorithm changes depending on whether it is applied to array architecture or pipeline architecture, because the algorithm's purpose differs depending on the type of architecture. The purpose of P&R in array architecture is to find the P&R solution that has the shortest data path. However, if any P&R solution is found, the search can be finished in pipeline architecture because the path length of pipeline architecture, which has few data paths, is dependent on the number of pipeline stages in any P&R. Therefore, P&R algorithms of array architecture usually employ metaheuristic algorithms, of which SA is a typical example [2] [4]. Metaheuristic algorithms can only use architecture that has high degree of freedom of P&R, such as array architecture [6]. On the other hand, full search algorithms have been used for pipeline architecture that has very low degree of freedom of P&R. However, pipeline architecture that has high degree of freedom of P&R, has been considered in recent years [3]. Search time of this pipeline architecture becomes huge when using a full search algorithm, because it has many P&R patterns and search space may increase. [5]. Moreover, the P&R discovery rate decreases when a metaheuristic algorithmis applied, because degree of freedom of the operation placement is low. For pipeline architecture that has not only low degree of freedom, but also high degree of freedom, we propose P&R algorithm, which has high speed and high discovery rate of P&R compared with existing algorithms.

In section 2, we model the pipeline architecture. Section 3 explains the proposed P&R algorithm. In Section 4, we compare the proposed algorithm and metaheuristic algorithm, and evaluate the effectiveness of the proposed functions. Moreover, we evaluate general versatility in the general pipeline architecture with the proposed algorithm. Finally, section 5 concludes our paper and refers to future work.

2. Architecture

In this section, the target pipeline architecture is modeled. In the pipeline architecture, the P&R degree of freedom is

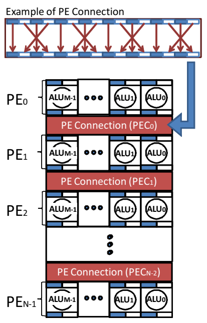


Fig. 3: Architecture Model

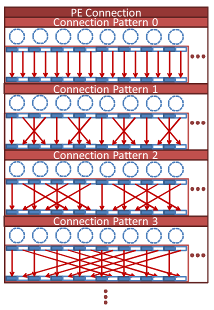


Fig. 4: PE Connection

one of the items that influences the search time. So we propose a pipeline architecture that transfers data only one way along the pipeline and has a number of different types of one-way data path patterns. Because data transfer is limited to one way along the pipeline, the area of this pipeline architecture is smaller than that of the pipeline architecture that has high P&R degree of freedom. And, since different types of one-way data path patterns can be selected, operation groups can be placed and routed more flexibly than in the case of the pipeline architecture that has low P&R degree of freedom. Fig.3 shows the structure of the modeled pipeline architecture. The modeled pipeline architecture consists of N Processing Elements (PEs) and $N-1$ PE Connections (PECs) that transfer data between PEs. It inputs data to PE_0, and outputs the operation result from PE_{N-1} at the end. Each PE has M Arithmetic Logic Units (ALUs). Each ALU of PE_i inputs the two data through PEC of PE_{i-1}, and outputs the two operation results. ALUs, which constitute PEs, and PECs are discussed in more detail below.

2.1 ALU

In addition, ALUs can execute different operations simultaneously. The condition of two inputs is applied because most of the arithmetic and logical operations require two inputs. Therefore, all the operation results can be inputs of ALUs in the following pipeline stages by setting the output results to be two. Each ALU can input two data and output the operation result.

2.2 PE Connection (PEC)

PEC is used for data transfer between adjacent PEs. It can change the order of output data from ALUs. PEC has S data transfer pattern. Fig.4 shows a sample of PEC pattern that is data transfer pattern of PEC.

PEC pattern consists of data paths that transfer data from ALU of PE_i to ALU of PE_{i+1} as shown in Fig.4. Note that each ALU must connect to the ALU of neighboring PE. Each PEC can select either of S patterns.

3. Algorithm

The proposed algorithm is intended to place and route data flow graph for the pipeline architecture modeled in the previous section. We define the data flow graph used by the proposed algorithm.

In the data flow graph, nodes express operations, and edges express data dependences. We use the directed graph, which is a kind of data flow graph, to express input and output data. Moreover, we use the acyclic graph, which is a kind of data flow graph, because the target architecture is pipeline and it is not possible for a output data that has already been calculated to become an input of the operation. In short, the data flow graph is expressed by Directed Acyclic Graph (DAG). The graph is a binary tree because ALU has two inputs and two outputs. The binary tree nodes of the same path length are less than M, which is the ALU number of each PE, and critical path length of the binary tree is less than N, that is the number of pipeline stages. DAG that meets these requirements is hereinafter referred to as Binary DAG (BDAG). BDAG is the input of the proposed algorithm. Fig.5 shows an example of BDAG in the case that the number of PE is $N = 5$ and the number of ALU is $M = 8$.

We consider the possible application to the P&R problem of existing algorithms, based on the features of defined BDAG and the features of the pipeline architecture model. If we decide P&R by using full search algorithm the search area depends on the target architecture though we can search all P&R solutions. Since the pipeline architecture targeted in this paper has very large search area, the calculation amount is huge and finding the P&R solution in realistic time becomes difficult. Eq.(1) is the equation of $Pattern\_num$, which is the number of routing patterns.

$$Pattern\_num = {}_{M*(N+1)}P_{V_{num}} * S^N \tag{1}$$

M expresses the number of ALUs in each PE, N expresses the number of PECs, V_{num} expresses the number of nodes in BDAG, and S expresses the number of PEC patterns in each PEC. The execution time of searching P&R solution becomes more than several years, because the number of solutions that can place and route is very small compared with the number of P&R patterns calculated by Eq.(1). This is understood from the exponential growth in P&R search time with increasing M, N, and S.

On the other hand, metaheuristic algorithms, of which SA is a typical example, are applicable to a wide search area by repeatedly searching the random P&R patterns and their neighboring solutions. In short, metaheuristic algorithm is suited to the array architecture whose P&R solution, such as that shown in Fig.1, has various paths. However, the pipeline architecture has a small chance of finding a correct solution from the random P&R solution because this architecture has many constraints, which is evident from Fig.1 and Fig.2. In Fig.1 there are many paths between two arbitrary arithmetic units, but in Fig.2 there are very few paths or no paths between two arbitrary arithmetic units. In short, the pipeline architecture has very few P&R solutions, whereas P&R solutions are easily found in the case of array architecture. Therefore, using constrained conditions of the pipeline architecture model, we propose an algorithm that searches only P&R patterns close to the P&R solution.

As a result, the proposed algorithm can search P&R solution faster than metaheuristic algorithm. And, we certainly find the correct solution if P&R solution exists, because we search all P&R patterns close to P&R solution.

The proposed P&R algorithm in the pipeline architecture model is executed in the order of PreProcess, Pruning, P&R Search and PostProcess. The outline of each phase is described as follows:

1) PreProcess
   In this phase, the path length of all paths of BDAG between input and output is normalized to the number of PE (N).
2) Pruning
   In this phase, the execution time is cut by selecting a node that needs search and reducing P&R solutions.
3) Placement and Routing Search (P&R Search)
   In this phase, we search only the selected nodes of BDAG at Pruning. ALU placement in each node and PEC pattern in each PEC are decided by using the graph dependency.
4) PostProcess
   In this phase, placement of nodes other than the selected nodes is decided and final P&R result of the pipeline architecture model is generated.

In the following, we explain P&R Search, which is the main phase of the proposed P&R algorithm. Then, we explain Pruning, which is an important phase for reducing search time and improving P&R possibility in the proposed P&R algorithm.

3.1 P&R Search

At the P&R Search phase, placing of each node and PEC pattern of each PE are decided based on the data dependency relationship of BDAG. Placing of node and PEC pattern are decided per PE, and they are decided alternately. Hereinafter, Placement and Routing Search is described step by step, assuming that input BDAG is already normalized as shown in Fig.5. In normalized BDAG, each node has its own PE (row) and it is only necessary to decide the place of ALU (column). PE is arranged along the pipeline, as PE_0, PE_1 and PE_2.

If there is no place of node or PEC pattern that satisfies the data dependences of Normalized BDAG, the previously decided place of node must be relocated or the previously decided PEC pattern has to be exchanged. This method is called backtrack search. The decision method for choosing PEC pattern and Node Placement is described below. This method can start search from either PE_0 or PE_{M-1}, but our description of the algorithm is based on the assumption that it starts from PE_{M-1} in this case.

PEC pattern of PEC_i can be decided by the place of

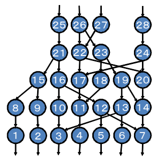


Fig. 5: BDAG

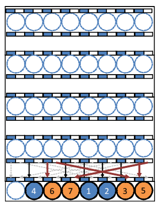


Fig. 6: PEC

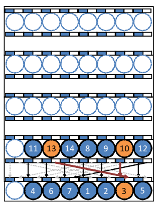


Fig. 7: PE

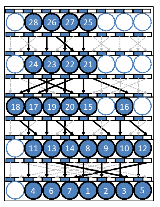


Fig. 8: P&R

node in PEC_{i+1}, which has already been allocated in the previous step. If the operation of ALU_{IN} or ALU_{OUT} does not correspond to the parent node or the child node of BDAG, prune this search area that is regarded as impossible for P&R. Fig.6 shows a sample of P&R search using the pipeline architecture. Fig.5 shows the input BDAG. The number of PEs of this pipeline architecture is $N = 5$ and the number of ALUs in each PE is $M = 8$. In Fig.6, PEC_3 is decided by PE_4 and its nodes are decided randomly. When no PEC pattern can be applied, PEC_i must be searched using backtrack search after replacing the nodes of PE_{i+1}.

Node Placement is done so that the PEC pattern chosen in the previous step satisfies the dependence of Normalized BDAG. Place of node in PE_i is decided by place of in-place node in PE_{i+1} and PEC_i. The node of PE_i is placed along routing of PEC_i, which is already decided. Fig.7 shows a result of the ALU place of the nodes in PE_3. For this pruning, placement patterns can be significantly reduced.

Fig.8 shows a result of PEC pattern and Node Placement decided from PE_4 to PE_0. P&R in the pipeline architecture is realized by deciding Node Placement and PEC pattern alternately from one side to the other. Here, search time has been reduced by pruning PEC pattern and Node Placement pattern using sibling relationship and parent-child relationship of BDAG. Eq.(2) shows $Place\_Pattern$, which is the rate of reduction of P&R patterns.

$$Place\_Pattern = 2^m / {}_M P_m \tag{2}$$

m expresses the number of operation nodes requiring placement of each PE. 2^m expresses the number of P&R patterns of each PE after pruning of PEC pattern. ${}_M P_m$ expresses the number of P&R patterns of each PE for which P&R patterns were not pruned. Fig.6 shows the case of reducing the number of P&R patterns from 4 to 1. As a result, the search time is greatly reduced.

3.2 Pruning

P&R Search subsection explains reduction of the search P&R pattern that meets conditions in the case of the pipeline architecture. In the Pruning subsection, we explain the proposed higher-speed algorithm. Therefore, two optimizations for speeding up the algorithm, named Search Node Selection and Search Direction Decision, are performed in the Pruning phase.

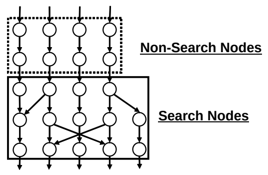


Fig. 9: Non-Search Nodes, Search Nodes

3.2.1 Search Nodes Selection

Search Nodes Selection aims to reduce the search patterns by restricting the nodes mapped in the P&R Search phase. It is unnecessary to search the nodes of BDAG that can be allocated by any Shuffle Pattern. Search patterns can be reduced by dividing all nodes into Search Nodes that require search and Non-Search Nodes that do not require search. Non-search nodes are allocated in Post-processing at tiny calculation cost. This division can be performed by looking at the number of input-output edges of each node. When the number of input edges is 1 and the number of output edges of all nodes in PE_i is 1, all nodes of PE_i become Non-Search Nodes. The other nodes become Search Nodes. Fig.9 shows an example of Non-Search Nodes and Search Nodes of BDAG. In this case, the problem of the pipeline architecture has been reduced from 5-column to 3-column.

3.2.2 Search Direction Decision

Search Direction Decision aims to reduce the search time by discovering PE_i that has less initial searching space.

In regard to Placement and Routing Search, we explained the method for advancing the search from PE_i to PE_{i-1}. However, this algorithm can also search from PE_i to PE_{i+1}. In addition, the number of total search patterns varies with starting PE since the number of search patterns can be greatly decreased by pruning search patterns initially. Therefore, we reduce the number of search patterns by starting the search from the PE that has the fewest search patterns. The selection method is described as follows.

PE_i with the smallest i in PEs is assumed to be PE_a and PE_i with the biggest i is assumed to be PE_b. The index number of search is calculated by Eq.(1) for both $x = a$ and $x = b$. PE_x that has smaller value of $Pattern(PE_x)$ is selected as initial PE for searching.

$$Pattern(PE_x) = NodeNum_x / EdgeNum_x \qquad (3)$$

Here, the $NodeNum_x$ expresses the number of nodes in PE_x, and $EdgeNum_x$ expresses the number of output edges. The smaller the number of nodes, the smaller the search pattern of Node Placement is. Conversely, the smaller the number of output edges, the more the search patterns of PEC pattern increase.

However, the search beginning with PE calculated from this expression is not always correct. It can be seen that some BDAG cases finish the P&R search faster than search from the direction calculated by the expression Eq.(1). Then, when fixed time passes, it is necessary to introduce a method of restarting the search from the direction opposite to the search beginning with PE calculated by the expression Eq.(1). As a result, this method has the potential to speed up P&R searching.

4. Evaluation

In this section, we compare P&R searching algorithm using SA with the proposed algorithm in terms of the execution time and the P&R discovery rate solution. Next, we perform efficacy evaluation of the proposed pruning functions. Lastly, we evaluate the execution time of our P&R algorithm with various pipeline architectures. This comparison experiment is designed to verify whether the proposed P&R algorithm can be applied to pipeline architecture even if its architectural constitution scales. Random BDAGs used for these verifications are generated by the indicator called Complexity calculated based on the number of edges and the number of nodes. To ensure reliable verification, we use a test set composed of randomly generated BDAGs that differ in terms of Complexity. Eq.4 shows the expression for calculating Complexity. Eq.5 shows the composition of the test set.

$$Complexity(G) = (V_{num} * E_{num})/(M * N) \qquad (4)$$

$$\begin{aligned} test\_set = & set(0.00 < complexity \le 0.25) \\ & \cup\, set(0.25 < complexity \le 0.50) \\ & \cup\, set(0.50 < complexity \le 0.75) \\ & \cup\, set(0.75 < complexity \le 1.00) \end{aligned} \qquad (5)$$

$Complexity(G)$ shows Complexity of BDAG (G). V_{num} expresses the number of nodes of G and E_{num} expresses the number of edges of G. Thus, Complexity is the utilization rate of nodes and edges of the BDAG in the number of ALUs M and the number of pipeline stages N. $set(a < complexity \le b)$ expresses the set of 25 BDAGs that is $a < complexity \le b$. Therefore, a $test\_set$ is the set of 100 BDAGs. It has already been shown that BDAGs of the $test\_set$ are able to place and route. When we evaluate pipeline architecture of a different scale, the validity of the evaluation using Complexity is assured.

In the evaluation, we determine the switch time parameter of the search direction $T_{switch} = 600 seconds$. SA P&R algorithm that is compared with the proposed P&R algorithm is described below. Afterwards, we report comparative evaluation of the execution time between SA P&R algorithm and the proposed P&R algorithm and perform efficacy evaluation of the pruning functions. And lastly, we report the general versatility evaluation in the general pipeline architecture with the proposed P&R algorithm.

4.1 SA_ P&R Algorithm

SA is an algorithm that searches a much better P&R solution by generating a random solution and searching for the local neighborhood solution of the previous solution. If

Table 1: Comparison parameter : ALU

	M=2	M=4	M=8
P_P&R	99%	99%	99%
SA_P&R	98%	56%	5%

the local neighborhood solution is better than the previous solution, the local neighborhood solution is replaced as the basic solution. The basic solution is the basis of the next local neighborhood solution. Even if the local neighborhood solution is not improved from the previous solution, it is replaced as the current solution with a certain probability. This probability gradually falls as the search advances. As a result, we can prevent convergence to the local optimum solution in the early search stage.

In the searching P&R, if we search for placement of the BDAG nodes, the BDAG edges, and PEC pattern simultaneously, the solutions will be discrete. Consequently, SA algorithm that searches the local neighborhood solution is unsuited to P&R search. Therefore, we devise the SA algorithm that searches only placement of the BDAG. In this case, PEC pattern of all PEC in the target pipeline architecture is already decided. All combinations of PEC pattern in the pipeline architecture are executed in linear sequence.

We can compare the result to performance evaluation of the proposed P&R search algorithm by constructing the SA P&R search algorithm.

4.2 Comparative Evaluation

In this subsection, we compare the discovery rates of P&R solutions between SA P&R search algorithm (SA_P&R) and the proposed P&R search algorithm (P_P&R). In the evaluation, we use a $test\_set$ that consists of 100 BDAGs. The P&R discovery rate solution shows the rate of resolved BDAGs that can find the P&R solution out of 100 BDAGs ($test\_set$) in $1,200 seconds$. To compare various pipeline architectures, we use three parameters, namely, the number of ALUs in each PE (M) and the number of PEs (N), and the number of PEC patterns (S). The evaluation result of each parameter is shown below.

Table1 shows the P&R discovery rate in which the number of ALUs (M) is changed. In this evaluation, the number of PEs is $N = 4$ and the number of PEC patterns is $S = 2$. As a result, in the case of $M = 2$, both P_P&R and SA_P&R could solve most BDAGs. In the case of $M = 4$ and $M = 8$, the P&R discovery rate does not decrease in P_P&R. However, the P&R discovery rate decreases remarkably as the number of ALUs increases in SA_ P&R.

Next, Table2 shows the P&R discovery rate in which the number of PEs (N) is changed. In this evaluation, the number of ALUs is $M = 8$ and the number of PEC patterns is $S = 4$. As a result, even if the number of pipeline stages is increased to $N = 10$, more than 59% of BDAGs are resolved in the case of P_P&R. However, in the case of SA_P&R, the P&R discovery rate, which is less than 50% for $N = 2$,

Table 2: Comparison parameter : PE

	N=2	N=4	N=6	N=8	N=10
P_P&R	99%	74%	59%	62%	59%
SA_P&R	45%	15%	4%	3%	0%

Table 3: Comparison parameter : PEC pattern

	S=2	S=4	S=8
P_P&R	95%	75%	55%
SA_P&R	5%	15%	6%

decreases as N increases and no P&R solution for $N = 10$ is found. P_P&R is more applicable to the increase of N than SA_P&R.

Finally, Table3 shows the P&R discovery rate in the case that the number of PEC patterns (S) is changed. In this evaluation, the number of ALUs is $M = 8$ and the number of pipeline stages is $N = 4$. As a result, the P&R discovery rate is low for SA_P&R regardless of the value of S. In contrast, the P&R discovery rate is over 50% regardless of the value of S, which is as good as other evaluation results for P_P&R. Moreover, the P&R discovery rate of $S = 4$ for P_P&R is higher than $S = 2, 8$ for SA_P&R. This is because the degree of freedom of node placement improved as PEC patterns increased, and it became easy to discover the P&R solution for SA_P&R. However, the P&R discovery rate decreases because the search time is insufficient though the degree of freedom of the node placement is high for $S = 8$.

From the results of Table1, Table2, and Table3, the P&R discovery rate is more than 50% for all parameters for P_ P&R. But, the P&R discovery rate of SA_P&R is the same as the P&R discovery rate of P_P&R only in very limited pipeline architecture. When the scale of the pipeline architecture is expanded, a decrease of the discovery rate in the case of using SA_P&R is more remarkable than in the case of using P_P&R. Thus, our P_P&R algorithm is applicable for various types of pipeline architectures, and it is difficult to employ SA_P&R algorithm for those architectures.

4.3 Efficacy Evaluation

In this subsection, we perform efficacy evaluation of the proposed pruning functions. Our P&R algorithm introduced two pruning functions (Search Nodes Selection and Search Direction Decision). So, we compare the discovery rates of P&R solutions between enable and disable the target function. In this evaluation, we use a $test\_set$ that consists of 100 BDAGs, and count the number of resolved BDAGs.

4.3.1 Efficacy Evaluation of Search Nodes Selection

First, we perform efficacy evaluation of the Search Nodes Selection. This function decide the number of pipeline columns. So we evaluate the P&R discovery rate. In this evaluation, the number of PEs is $N = 6$, the number of ALUs is $M = 8$ and the number of PEC patterns is $S = 4$. Fig.10 shows the result of efficacy evaluation of

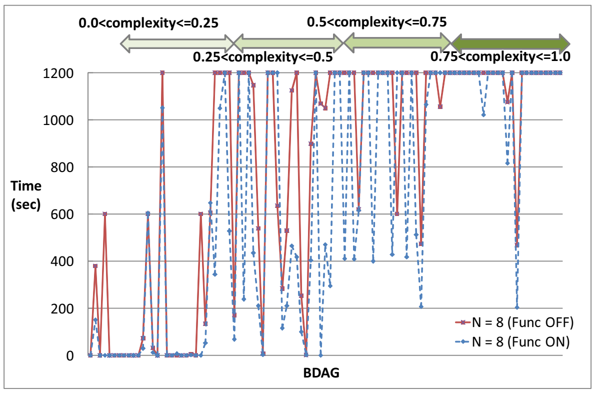


Fig. 10: Efficacy Evaluation : Search Nodes Selection

Table 4: Efficacy Evaluation : Search Direction Decision

Decision type / N=	2	4	6	8	10
Optimal Decision	99%	70%	49%	46%	47%
Func ON	99%	69%	49%	46%	42%
Func OFF(from Top)	99%	56%	35%	30%	33%
Func OFF(from Bottom)	99%	56%	39%	30%	34%

Search Nodes Selection. In the graph, a horizontal axis is a BDAG and a vertical axis is a time scale. BDAGs are sorted in ascending order of complexity. And, the BDAG of $1,200 seconds$ shows that the solution cannot be discovered and the search was broken off. Moreover, N = 6 (Func ON) is a result for validating Search Nodes Selection, and N = 6 (Func OFF) is a result for invalidating the function. This result shows that the search time of BDAGs is equal or less. The search time of 37% BDAGs is improved, and that of 6% gets worse. The case of the search time deteriorates, the order of generating the search pattern is changed by the Search Nodes Selection, and the search time became long. The search time of low-complexity BDAGs is short, regardless of the proposed function. And, a few BDAGs of high complexity improve the search time. Moreover, the proposal function is effective in many of the middle-complexity BDAGs, and reduces the search time. As a result, if we validate Search Nodes Selection, some BDAGs can discover P&R solution earlier.

4.3.2 Efficacy Evaluation of Search Direction Decision

Next, we perform efficacy evaluation of the Search Direction Decision. This function selects effective search direction and obtains the P&R result faster. And, we compare the discovery rates of P&R solutions of four cases. The first case is selection of the best search direction discovered manually (Optimal Decision). In the next case, the search direction is decided by using the proposal function (Func ON). And in the third case, all Binary DAGs are searched from top to bottom (Func OFF from Top). In the last case, the search is from bottom to top (Func OFF from Bottom). Then, the maximum search time is $600 seconds$.

There are few differences between the result of Func OFF (from Top) and that of Func OFF (from Bottom). When Func OFF and Func ON were compared, the discovery rate

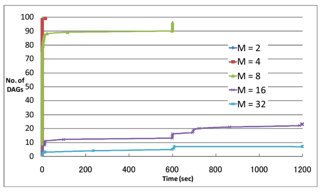


Fig. 11: Parameter : ALU

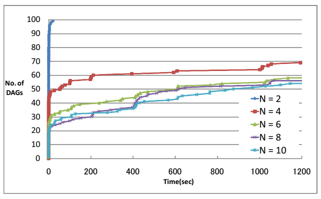


Fig. 12: Parameter : PE

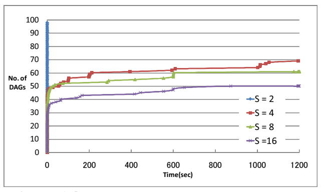


Fig. 13: Parameter : PEC pattern

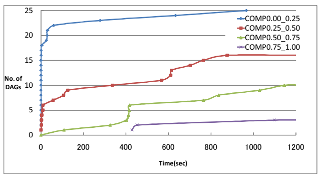


Fig. 14: Complexity of PE=10

improved with all N. Moreover, the result of Func ON approaches that of Optimal Decision, and when N increases, some BDAGs that cannot select a correct search direction are generated. Table4 shows that more than 40% of BDAGs discovered the P&R solution at Func ON. Moreover, the discovery rate of the search result of the proposal function is higher than the Function OFF cases. In addition, it became almost the same result as the Optimal Decision case. As a result, Search Nodes Selection is effective for the faster search. From the observations above, the two pruning functions are effective for increasing P&R discovery rate and search time reduction.

4.4 Versatile Evaluation

In this subsection, we evaluate scalability of our P_P&R algorithm with architectures whose architectural parameters are changed. In this evaluation, we count the number of BDAGs resolved in less than $1,200 seconds$. To evaluate the execution time of various pipeline architectures, we use three parameters, namely, the number of ALUs in each PE (M) and the number of PEs (N), and the number of PEC patterns (S). The result when each parameter is changed is shown as follows. In the graph, a horizontal axis is a time scale and a vertical axis is the number of BDAGs that can discover P&R solution. In short, the value on the graph shows the number of BDAGs that can discover P&R solution in a certain time.

Fig.11 shows the result in the case that the number of ALUs (M) is changed. In this evaluation, the number of PEs is $N = 4$ and the number of PEC patterns is $S = 2$.

Fig.12 shows the result in the case that the number of PEs (N) is changed. In this evaluation, the number of ALUs is $M = 8$ and the number of PEC patterns is $S = 2$. Fig.13 shows the result in the case that the number of PEC patterns (S) is changed. In this evaluation, the number of ALUs is $M = 8$ and the number of PEs is $N = 4$. At first, we comment on Fig.12, Fig.13, and Fig.11.

According to Fig.12 and Fig.13, many BDAGs are resolved in the first few seconds for any parameter. On the

Table 5: reduced PEs with Search Node Selection

Complexity	0-0.25	0.25-0.5	0.5-0.75	0.75-1
Reduced PEs	3.52	0.24	0.04	0

other hand, according to Fig.11, very few BDAGs in $M = 16$ and $M = 32$ are resolved. From those results, it is clear that the number of ALUs greatly affects the search area and leads to far fewer P&R solutions if it is increased. In all results, the smaller the value of the changed parameter, the more BDAGs are resolved.

In the following, we describe the impact of three P_P&R functions on P&R of BDAGs. The three P_P&R functions are Search Direction Decision, Search Node Selection, and the pattern reduction with P&R Search, which are important functions of our P_P&R. First, Search Direction Decision is considered. Some BDAGs are resolved in about $600 seconds$, which is an effect of Search Direction Decision explained in Section.3.2. Search Direction Decision is effective for deciding the search direction because many BDAGs are resolved in the first few seconds. However, the direction decision failed for some BDAGs. Therefore, we switch the direction at $600 seconds$, and the increase of the search time can be suppressed by finding P&R solution after the direction switch.

Next, Search Node Selection is considered. In Fig.12, even if the value of N increases, the decrease in the P&R discovery rate in BDAGs is slight. For considering this result, shown in Fig.14 are the evaluation results of 4 different DAG groups, each of which has different Complexity. This graph shows the P&R discovery rate for $N = 10$, which seems to be the most difficult to place and route. The total of the vertical axis is 25, because each Complexity group has 25 BDAGs. As a result, the P&R discovery rate is high when Complexity is low. In particular, when Complexity is $0.00 < Complexity \leq 0.25$, most BDAGs are resolved in $1,200 seconds$. On the other hand, when Complexity is $0.75 < Complexity \leq 1.00$, most BDAGs could not be resolved. So, the BDAG that has low Complexity maintains the constant discovery rate. This is because the BDAG that has low Complexity is easy to resolve regardless of the value of PE. In addition, the BDAG that has low Complexity can find many BDAGs in the first few seconds even though the number of pipeline stages is large. This result is an effect of Search Node Selection explained in Section3.2. Table5 shows average of PEs that is reduced by Search Node Selection in the set of BDAGs of each Complexity. Complexity $0 - 0.25$ expresses the set of 25 BDAGs that is $0.00 < complexity \leq 0.25$. Everything else is similar. Reduced PEs show the average number of reduced PEs. As a result, the set of BDAGs that has low Complexity can reduce many PEs from all PEs in the search area. In short, the search time can be significantly reduced because the BDAG that has low Complexity has many nodes that should not be searched and the search area is reduced by Search Node Selection.

Finally, we consider the pattern reduction by the P&R Search phase. In Fig.13, the P&R discovery rate decreases as the value of S increases because the search area expands owing to the increase of PEC patterns. But, on the other hand, the search area is reduced by selecting PEC pattern by constraint of P&R in the P&R Search phase. As a result, a rapid decrease of the discovery rate by the search area expansion is suppressed.

From the observations above, it is obvious that our P_ P&R algorithm can obtain solutions exceeding constancy. In addition, most of those solutions are discovered in the first few seconds. By the evaluation of Complexity, P&R solution is easily found for BDAG that has low Complexity even if the scale and the flexibility of the pipeline architecture are improved.

5. Conclusion

For the architecture that has severe P&R constraint, it was shown that it is hard to apply a metaheuristic algorithm such as SA algorithm. On the other hand, the proposed P&R algorithm that prunes the search area by using those constraints is a valid algorithm. In future work, we intend to consider an extended algorithm that places and routes high-Complexity BDAG in the pipeline architecture that has large scale or high flexibility. If the BDAG has high Complexity, we take the approach that decreases the search time through further pruning of the search area by starting to search from the place that has many P&R constraints. Moreover, we can take another approach in which the search time is decreased through simplification of the search problem by dividing pipeline architecture and BDAG.

References

[1] H. Amano, A. Jouraku, and K. Anjo. A dynamically adaptive hardware on dynamically reconfigurable processor. *ACM*, E86-B(12), 2003.

[2] V. Cerny. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Application*, January 1985.

[3] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. Piperench: a co/processor for streaming multimedia acceleration. *IEEE Computer Society*, 1999.

[4] S. Kirkpatric, J. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Sience*, May 1983.

[5] A. Kuroda, M. Koezuka, H. Matsuzaki, T. Yoshikawa, and S. Asano. Mapping method for dynamically reconfigurable architecture. *ASP-DAC*, January 2009.

[6] B. Mei, V. Serge, V. Diederik, D. M. Hugo, and L. Rudy. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures. *IEEE international conference on field-programmable technology (FPT)*, December 2002.

[7] M. Okada, T. Hiramatsu, H. Nakajima, M. Ozone, K. Hirase, and S. Kimura. A reconfigurable processor based on alu array architecture with limitation on the interconnection. *In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, 04, April 2005.

[8] T. Yoshikawa, Y. Yamada, and S. Asano. An implementation of hardware accelerator using dynamically reconfigurable architecture. *IEEE HotChips*, August 2006.

[9] T. Yoshikawa, Y. Yamada, and S. Asano. Implementation and evaluation of the processor for stream multimedia applications using dynamic reconfiguration. *IEEE COOLChips X*, April. 2007.

Accelerating Real-time processing of the ATST Adaptive Optics System using Coarse-grained Parallel Hardware Architectures

V. Venugopal, K. Richards, S. Barden, T. Rimmele, S. Gregory, L. Johnson
National Solar Observatory, P.O. Box 62, Sunspot, New Mexico USA
Email: {vivekv, richards, sbarden, rimmele, bscott, ljohnson}@nso.edu

Abstract—*The real-time processing of the four meter Advanced Technology Solar Telescope (ATST) adaptive optics (AO) system with approximately 1750 sub-apertures and 1900 actuators requires massive parallel processing to complete the task. The parallel processing is harnessed with the addition of hardware accelerators such as Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU). We investigate the hybrid data processing architecture of the Shack-Hartmann correlation and wavefront reconstruction using FPGAs and GPUs. The ATST AO algorithm is implemented, benchmarked on the FPGA-GPU system and compared with the existing legacy Digital Signal Processing (DSP) based hardware system. The cost-effective FPGA platform provides better throughput for the AO processing as compared to the GPU architecture.*

Keywords: Adaptive optics systems; wavefront correction; field programmable gate arrays; parallel processing; graphics processing units; real-time systems

1. Introduction

Atmospheric turbulence distorts the wavefront by generating phase variations in the incoming light and limits the resolution of large solar telescopes such as the four meter solar telescope, Advanced Technology Solar Telescope (ATST) now beginning construction at Maui's Haleakala [1], [2]. The adaptive optics (AO) system senses the wavefront aberrations and applies the corresponding correction to the adjustable deformable mirror to improve the resolution of the telescope. The key components of the ATST AO system as shown in Figure 1 are:

1) tip-tilt mirror (TTM), which moves fast to remove the image shift,
2) deformable mirror (DM) to flatten the incoming wavefront by changing shape,
3) Shack-Hartmann lenslet array, that focuses sub-aperture images of the sun's surface on the sensor of the high-speed camera,
4) high-speed camera, which captures the images of the sun,
5) processing system, that computes the shift in the images using cross-correlation as part of the wavefront sensing and calculates the actuator commands for the DM and TTM.

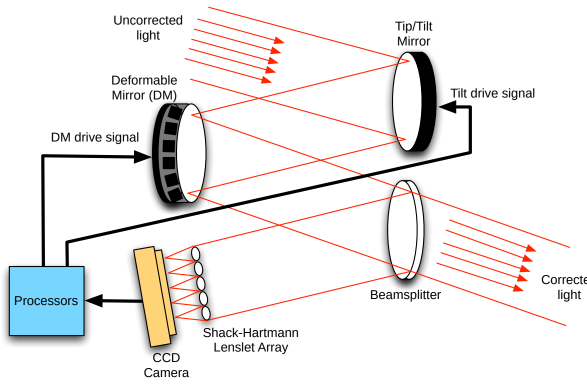


Fig. 1: Adaptive Optics system

This paper focuses on the implementation of the AO system using Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) as the major processing elements. The usage of FPGAs and GPUs as hardware accelerators effectively reduces the computation time of the wavefront correction and reconstruction algorithm in AO systems. Section 2 describes the real-time processing of the ATST AO system; Section 3 explains the camera data unpacking, the dark and flat correction routine implementation on the FPGA; Section 4 describes the implementation of the different correlation procedures and the remaining steps of the AO algorithm on the GPUs; Section 5 is work in progress on the FPGA implementation of the correlation routine; Section 6 describes the GPU routine for the reconstruction matrix; the results and conclusion are presented in Sections 7 and 8.

2. Real-time processing of the AO system

The real-time processing of the ATST AO system is shown in Figure 2. The high speed camera sends the 1750 20×20 pixel raw sub-aperture images to the processing system. The sub-apertures undergo a dark field correction followed by a flat field correction, which is the equivalent to correcting the images for zero level and gain equalization. The resulting flat field corrected image is 2D correlated with a reference image captured approximately every 10 seconds. The 2D cross-correlation step determines the shift in the x and y direction of each sub-aperture, as compared to the reference image.

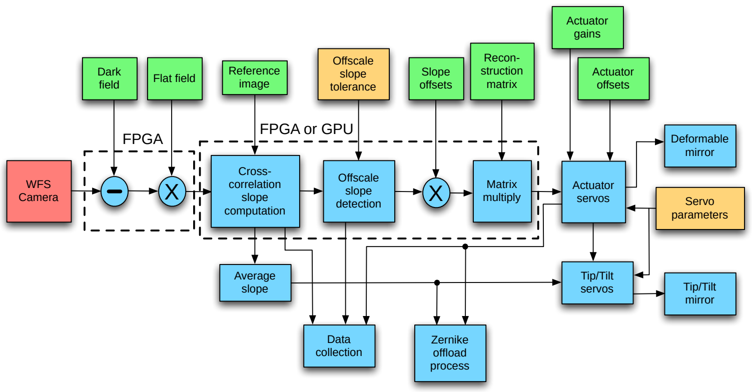


Fig. 2: Real-time processing for the ATST Adaptive Optics system

The wavefront reconstruction step consists of a precomputed 3500×1900 reconstruction matrix, which is multiplied with the x and y shifts. The matrix multiplication products are summed after all the sub-aperture images in that particular frame have been processed and is then transmitted to the DM and TTM actuators after applying a servo loop algorithm.

The above steps encapsulate the AO system which is also currently used at the Dunn Solar Telescope (DST) operated by the National Solar Observatory (NSO) [3], [4]. The real-time processing requirements are determined by the frame rate of the camera and the number of actuators used for the DM and TTM. The real-time processing requirements of 1232 sub-aperture images and 1313 actuators for the ATST AO system in [5] have been revised to 1750 sub-aperture images and 1900 actuators for the DM. The frame rate of the camera is equal to or greater than 2000 Hz, so that it can output 1750 sub-apertures of 20 × 20 pixels each. Figure 2 also shows the AO processes targeted on FPGA and GPU architectures. The complete AO operation for 1750 sub-aperture images has to be completed within 500 μs, which is a hard deadline. The current DSP-based AO system at the DST processes all the 76 sub-aperture images within 240 μs, but the DSP system would be more expensive if it was to be scaled for processing 1750 sub-apertures.

3. FPGA Implementation of the Dark and Flat correction process

The existing AO system at the Dunn telescope [3] uses a custom camera and interface to route the incoming data to the Analog Devices Hammerhead SHARC DSPs for the real-time processing. With the increase in gate density and the different IP cores that map complex functions to FPGAs, it is important to leverage the processing power of the FPGA. Also, the timing constraints tied to the processing budget is satisfied if some of the processing can be done by the FPGA.

Allocating the front-end processing on the FPGA reduces the data starvation and communication overhead for the following stages. The selection of FPGA device is influenced by the following constraints: (1) maximum number of transceivers for receiving the data from the camera, (2) maximum logic density for the design to fit on a single FPGA. To satisfy these criteria, we selected the Xilinx Virtex-6 XC6VLX550T FPGA, which has 549,888 logic cells and 36 high-speed Rocket I/O transceivers. The front-end processing modules are implemented on the FPGA as shown in Figure 3.

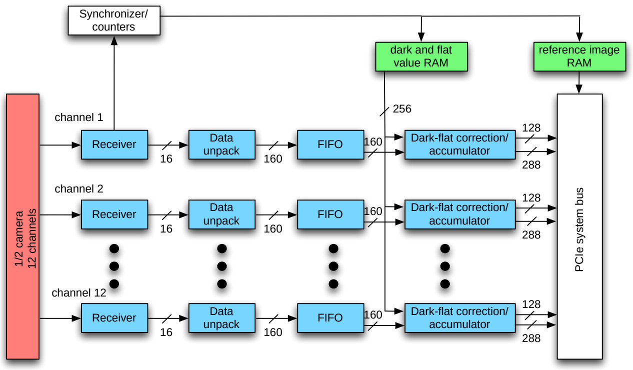


Fig. 3: FPGA implementation of the dark and flat correction routines

Each FPGA receives data from the high-speed camera through 12 fiber-channels at 2.125 Gbps. The Rocket I/O transceivers are configured to receive the camera data and the data unpacking is implemented using a Finite State Machine (FSM) module. The data unpacking module outputs 16 pixels (10 bits/pixel) to the dark and flat correction module. The unpacked pixels undergo dark and flat correction, which is a multiply-subtract routine and has a latency of 3 clock cycles. The reference image is acquired by the camera and sent to the FPGA every 10 seconds. The dark and flat correction module also generates a calibrated image, which is used to update the reference image. Both the reference image and the flat corrected image are made available for the following cross-correlation routine through the PCIe bus.

4. GPU implementation of the cross-correlation routine

The Nvidia Tesla C2050 Fermi-series GPU architecture consists of 14 streaming multi-processors with 32 cores clocked at 1.15 GHz each [6]. All the 32 streaming processors have common access to 64 KB of shared memory within a multi-processor. Each multiprocessor has one set of 32-bit registers per processor, constant memory and texture caches. Each streaming core can execute the same instruction on different data making it similar to a Single Instruction Multiple Data (SIMD) processor. The multi-processors communicate with the CPU through the GPU memory using the PCI Express interface. The GPU is referred to as a co-processor or device and the CPU is referred to as the host. Therefore, the application is partially executed on the host and the device. The host program copies the data to the device memory and the device program launches computational kernels which run on the multiple streaming cores. The GPU is programmed using Nvidia's C-based Compute Unified Device Architecture (CUDA) environment. The CUDA compiler assumes that the host and the device have separate accesses to their memory, also referred to as host memory and device memory. The AO algorithm consists of correlation routines which can be executed in parallel on the GPU and is described in the following sections.

4.1 FFT-based correlation

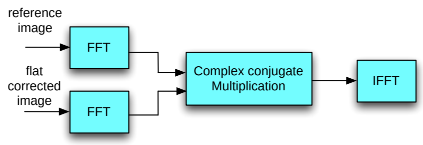


Fig. 4: FFT-based correlation routine

The Fast Fourier Transform (FFT) correlation routine shown in Figure 4 is prototyped on the GPU. The FFT correlation routine consists of taking the FFT of the flat corrected image and the reference image. The FFT values undergo complex conjugate multiplication followed by an inverse FFT procedure. The FFT and the inverse routines are implemented on the GPU using the existing *cufft* library.

4.2 7×7 correlation

Although the FFT-based correlation is preferred, the computational complexity is reduced by performing a 7×7 correlation. The Dunn AO system [3] operation assumes that the difference in the shift between the sub-aperture image and the reference image is rarely greater than plus or minus 3 pixels. Hence, an output cross-correlation array of 7×7 values is enough for computing a partial 2D cross correlation.

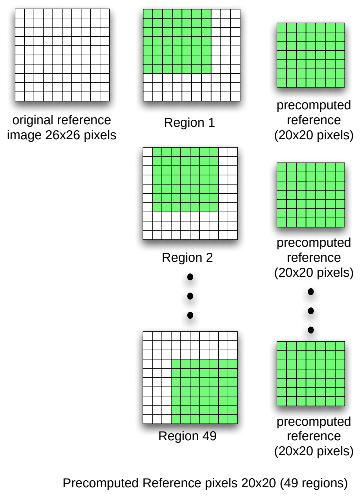

original reference image 26x26 pixels
Region 1
precomputed reference (20x20 pixels)
Region 2
precomputed reference (20x20 pixels)
Region 49
precomputed reference (20x20 pixels)
Precomputed Reference pixels 20x20 (49 regions)

Fig. 5: Precomputation step for the 7×7 correlation

Since the reference image consists of 26×26 pixels, the correlation step would incur an overhead due to uneven accesses by the GPU threads. This overhead is eliminated by precomputing the reference image as 20×20 pixels and the precomputation step yields 49 regions of 400 pixels each as shown in Figure 5. The 7×7 correlation involves multiply accumulate operation of the 400 pixels yielding 1 accumulated pixel per region. The resulting 49 pixels per image are then passed on to the next kernels for further processing.

4.3 find_max and interpolation routines

The *find_max* routine is based on the data reduction operation where the maximum value and it's index is calculated from the 49 cross-correlation values per sub-aperture. The maximum value is then used to calculate the x and y shifts using the interpolation equations shown in Equations 1-2.

$$\begin{aligned} num\_x &= max\_value - \\ &\quad out(shifted\_y\_index, (shifted\_x\_index - 1) \\ den\_x &= 2 * max\_value - \\ &\quad out(shifted\_y\_index, (shifted\_x\_index - 1)) - \\ &\quad out(shifted\_y\_index, (shifted\_x\_index + 1)) \end{aligned}$$

$$x = (shifted\_x\_index - 0.5) + \frac{num\_x}{den\_x} \quad (1)$$

$$\begin{aligned} num\_y &= max\_value - \\ &\quad out((shifted\_y\_index - 1), shifted\_x\_index) \\ den\_y &= 2 * max\_value - \\ &\quad out((shifted\_y\_index - 1), shifted\_x\_index) - \\ &\quad out((shifted\_y\_index + 1), shifted\_x\_index)) \end{aligned}$$

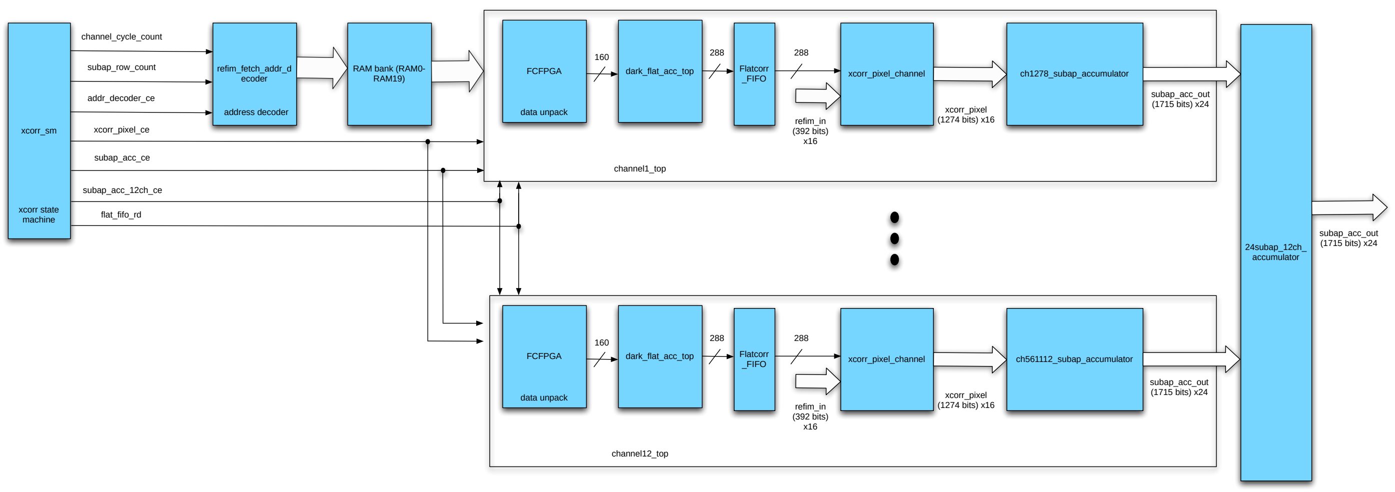


Fig. 6: Hardware design of the cross-correlation module

$$y = (shifted\_y\_index - 0.5) + \frac{num\_y}{den\_y} \qquad (2)$$

5. FPGA implementation of the correlation routine

The FPGA implementation is a preliminary analysis to determine if the complete AO algorithm can be mapped on to a single FPGA. The questions we seek to answer by the FPGA implementation are:

- How many FPGAs do we require to implement the complete AO algorithm?
- What is the latency and throughput that can be achieved by using FPGAs?

The hardware implementation of the correlation routine on the FPGA is shown in Figure 6. The correlation routine is controlled by the 20-state FSM module, *xcorr_sm*. Each FPGA only does half of the image processing and therefore it receives 875 sub-aperture images in 960 rows $\times$ 480 columns of pixels. Each channel requires 5 cycles to completely reconstruct the row of sub-aperture image. The *xcorr_sm* FSM keeps track of the rows processed and the cycles taken by the correlation module *xcorr_pixel_channel*. The row and the cycle information is also used by an address decoder to fetch the correct reference value for the correlation module.

The memory bank RAM0-RAM19 stores the pre-computed reference image values, where each value is a 7×7×8-bits corresponding to the pixel acquired by *xcorr_pixel_channel*. The *xcorr_pixel_channel* module implements a multiplier to multiply the reference pixel with the flat-corrected pixel. Each sub-aperture is 20×20 pixels and a total of 24 sub-apertures are processed in the first 20 rows $\times$ 480 columns. The sub-aperture values are accumulated and each channel only has certain pixels of the 24 sub-apertures. These values belonging to the different unique sub-apertures are accumulated by the *ch_subap_accumulator* module. Each channel performs the same operation and the final 24 sub-aperture values across all the 12 channels are accumulated by the *24subap_12ch_accumulator* module.

6. GPU-based implementation of the reconstruction procedure

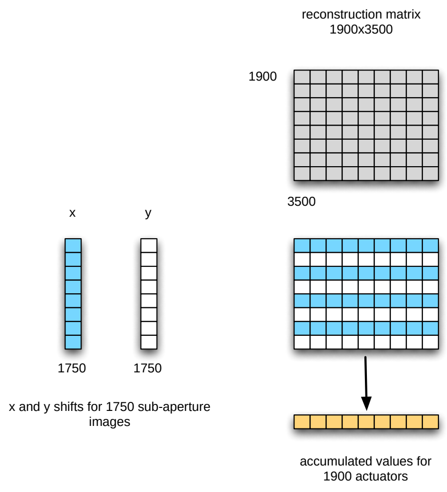

reconstruction matrix 1900x3500
1900
3500
x
y
1750
1750
x and y shifts for 1750 sub-aperture images
accumulated values for 1900 actuators

Fig. 7: Reconstruction matrix multiplication for 1900 actuators $\times$ 1750 sub-aperture images

Once the AO algorithm determines the x and y shifts for the 1750 sub-aperture images, these values are used to compute the 1900 actuator values for driving the mirrors as shown in Figure 7. The reconstruction routine involves the multiplication of the 1750 sub-aperture images with a 3500×1900 reconstruction matrix and the resulting products are then accumulated to give a single array of 1900 actuator values. The reconstruction procedure is executed on the DSP at the Dunn telescope and we implement the reconstruction procedure on the GPU to determine if a reasonable speedup can be achieved.

7. Results

Our host system consisted of 2 quad-core AMD Phenom processors clocked at 2.6 GHz with Ubuntu Linux OS. We used Nvidia's Tesla C1060 and Tesla C2050 GPUs to implement the AO algorithm and the reconstruction routine. The host system supports 3 GPUs and hence we partition the 1750 images into 584 images per GPU to emulate a pipe-lined system with better throughput. We used Nvidia's CUDA 3.2 release for implementing on the GPU. We prototyped the FPGA implementation on the Xilinx Virtex-6 XC6VLX550T FPGA, which has 549,888 logic cells and 36 high-speed Rocket I/O transceivers. For the GPU implementation of the AO algorithm, we emulated the flat and dark correction process on the FPGA by executing those kernels on the host. Also, the FFT of the reference image is computed only when the reference image is updated. Hence the FFT of the reference image was computed on the host to save time.

7.1 FFT correlation results

The FFT correlation based AO algorithm was implemented on both the Tesla GPUs, however the limiting constraint was the time taken for the complete algorithm execution. The results for 1 image and 50 images for all the routines including the *FFT kernels*, *find_max* and the *interpolation* routines are shown in Table 1.

GPU	**Execution time in μs**	
	1 image	**50 images**
Tesla C1060	1188	1619
Tesla C2050	1510	1889

Table 1: Comparison of FFT correlation based AO algorithm

7.2 7x7 correlation results

The 7×7 correlation based AO algorithm was implemented for 3 cases: 1 image, 50 images and 584 images. The results for all the routines including the *7×7 correlation*, *find_max* and the *interpolation* routines are shown in Table 2

It is evident from Tables 1 and 2 that the 7×7 correlation method is faster than the FFT correlation method. The FFT correlation method's latency is more than 500 μs for both test cases, whereas the 7×7 correlation method exhibits a latency of less than 400 μs for all the 3 test cases. The lowest timing numbers indicate best performance in both

GPU	**Execution time in μs**		
	1 image	**50 images**	**584 images**
Tesla C1060	278.36	279.35	281.39
Tesla C2050	312.9	307.49	300.93

Table 2: Comparison of 7×7 correlation based AO algorithm

tables and it is interesting to note that the Tesla C1060 with 240 cores has a better speedup over the Tesla C2050 with 448 cores. The Tesla C2050 has more cores and is the more recent GPU. In spite of using more threads and optimized shared memory utilization for the Tesla C2050, we did not see any improvement in speedup over the Tesla C1060.

7.3 Latency and throughput comparison

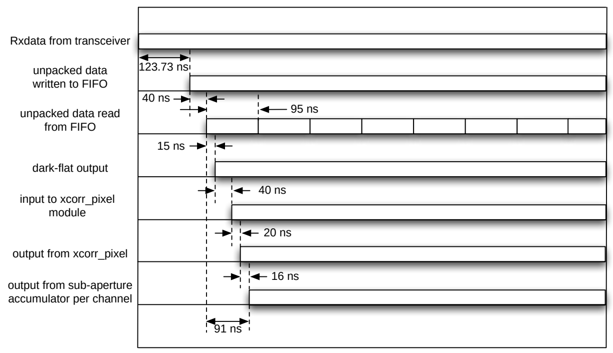

Rxdata from transceiver
unpacked data written to FIFO
123.73 ns
40 ns
95 ns
unpacked data read from FIFO
15 ns
dark-flat output
40 ns
input to xcorr_pixel module
20 ns
output from xcorr_pixel
16 ns
output from sub-aperture accumulator per channel
91 ns

Fig. 8: Timing diagram of correlation routine for FPGA

The latency of each block and the overall throughput can be determined from Figure 8. Each data packet is available from the FIFO after $95ns$. The latency for accumulating all the 24 sub-apertures per channel is $91ns$. It takes $95ns \times 5$ packets × 10 rows = 4.75 μs to have the data read from the FIFO. Since 2 rows are processed at a time, we need to only multiply by 10 rows to calculate the latency. The 4.75μs also represents the time taken for determining the cross-correlation for 24 sub-apertures present in the first 20 rows x 480 columns. Therefore, the total latency for completing the cross-correlation of 960 rows × 480 columns is given by,

$$4.75\mu s \times \frac{960}{20} = 228\mu s \tag{3}$$

The complete datapath for the 12 channels up to the correlation routine is targeted on the Xilinx Virtex-6 XC6VLX550T FPGA and simulated using Xilinx ISE 12.4 ISIM simulator. The mapping, place and route process was used to determine the size of the design. Table 3 shows that the design takes more than 100% of the available resources and does not fit on the selected FPGA. Two approaches can be considered: (1) partition the design over 2 FPGAs, (2) target the design for Xilinx Virtex-7 FPGA. The partitioning of the design introduces synchronization constraints which needs to be investigated. The Virtex-7 XC7V2000T FPGA has 1,954,560 logic slices with 36 GTX transcivers making it a suitable prototyping platform. However, the availability of the Virtex-7 XC7V2000T FPGA within a suitable timeframe is another criteria that influences this decision.

FPGA resources	Utilization	%age of FPGA used
Slice Registers	992448 out of 687360	144%
Slice LUTs	1126081 out of 343680	327%

Table 3: FPGA utilization summary

The throughput of the GPU vs FPGA based implementation is shown in Figure 9. Each Tesla C2050 GPU computes the x and y shifts for 584 images in 300.93 μs and therefore the GPU gives a throughput of 525.93 μs for 1750 images. Each GPU starts computing when the data for 584 images is transferred over the PCIe bus. The GPU implementation suffers due to the latency involved in the host to GPU transfer and the GPU to host transfer. There is additional latency for the reconstruction routine as the data needs to be moved from the GPUs to the host and then back to a single GPU. This latency can be eliminated if the global memory of the GPUs can be accessed or shared between all the GPUs. The FPGA starts computing on the data, as the camera streams the images through the 12 fiber channels. The FPGA performs the dark, flat correction and the correlation. The FPGA implementation exhibits the least latency for data movement. The DSP solution [3] is scaled to compute the 1750 images and requires 96 DSPs to complete the task in 495 μs. The FPGA solution provides the best throughput and is cost-effective as compared to the DSP solution.

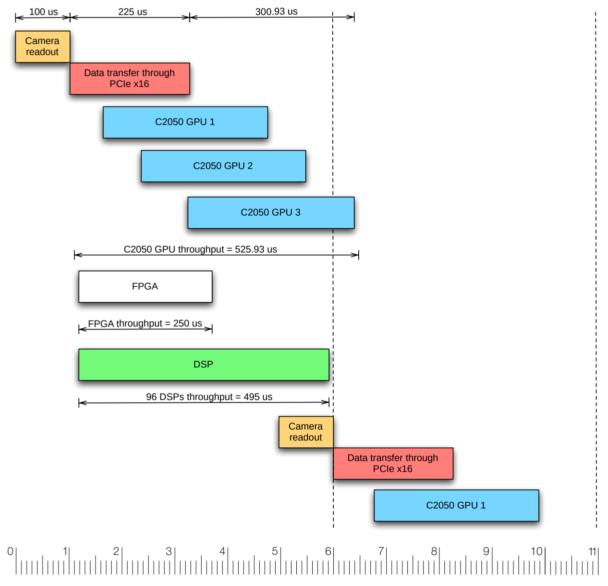


Fig. 9: Latency and throughput comparison for the GPU vs FPGA vs DSP implementations

7.4 GPU reconstruction results

Execution time in ms			
CPU	Tesla C1060 GPU	Tesla C2050 GPU	DSP
46.769	0.964	0.956	0.229

Table 4: Comparison of reconstruction algorithm

Table 4 shows the comparison of the execution time for the 1750 sub-apertures $\times$ 1900 actuators reconstruction algorithm. The Tesla C1060 and C2050 GPUs provide a speedup of 47$\times$ over the CPU implementation. The DSP solution uses 96 DSPs to compute the reconstruction routine and each DSP has a latency of 0.229 ms for 20 sub-apertures. However, the GPU provides a cost-effective solution over the DSPs for the implementation of the reconstruction routine.

8. Conclusion

The FPGA implementation of the AO algorithm provides the best throughput as compared to the GPU and the DSP implementations. The GPU provides excellent speedup over the CPU implementation of the reconstruction algorithm. However the GPU's performance is relevant only when there are no throughput constraints imposed on it. FPGAs provide a cost-effective solution over the DSPs with flexibility for combining the computation with custom I/O to meet the latency and throughput constraints.

Acknowledgements

The National Science Foundation (NSF) through the National Solar Observatory (NSO) funds the ATST Project. The NSO is operated under a cooperative agreement between the Association of Universities for Research in Astronomy, Inc. (AURA) and NSF.

References

[1] S. Keil, J. Jacobus M. Oschmann, T. R. Rimmele, R. Hubbard, M. Warner, R. Price, N. Dalrymple, B. Goodrich, S. Hegwer, F. Hill, and J. Wagner, "Advanced Technology Solar Telescope: conceptual design and status," *Proc. SPIE*, vol. 5489, no. 1, pp. 625–637, 2004.

[2] S. L. Keil, T. R. Rimmele, J. Wagner, and ATST team, "Advanced Technology Solar Telescope: A status report," *Astronomische Nachrichten*, vol. 331, pp. 609–+, 2010.

[3] K. Richards and T. Rimmele, "Real-time processing for the ATST AO system," in *Advanced Maui Optical and Space Surveillance Technologies Conference*, 2008.

[4] K. Richards, T. Rimmele, S. L. Hegwer, R. S. Upton, F. Woeger, J. Marino, S. Gregory, and B. Goodrich, "The adaptive optics and wavefront correction systems for the Advanced Technology Solar Telescope," B. L. Ellerbroek, M. Hart, N. Hubin, and P. L. Wizinowich, Eds., vol. 7736, no. 1. SPIE, 2010, p. 773608. [Online]. Available: http://link.aip.org/link/?PSI/7736/773608/1

[5] T. Rimmele, K. Richards, J. Roche, S. Hegwer, and A. Tritschler, "Progress with solar multi-conjugate adaptive optics at NSO," B. L. Ellerbroek and D. B. Calia, Eds., vol. 6272, no. 1. SPIE, 2006, p. 627206. [Online]. Available: http://link.aip.org/link/?PSI/6272/627206/1

[6] Nvidia Inc. (Last Accessed: March 2010) Nvidia Tesla C2050/C2070 GPU Computing Processor. [Online]. Available: http://www.nvidia.com/object/product$_{t}esla_{C}2050_{C}2070_{u}s.html$

A New Hardware/Software Partitioning Methodology Combining Search Space Smoothing and Discrete Particle Swarm Optimization

Yu Chen[1,3], Pranav Vaidya[1], Jaehwan John Lee[1]
[1]Department of Electrical and Computer Engineering
Indiana University-Purdue University
Indianapolis, U.S.
yc28@iupui.edu

Chandima Hewa Nadungodage[2], Yuni Xia[2]
[2]Department of Computer and Information Science
Indiana University-Purdue University
Indianapolis, U.S.
chewanad@umail.iu.edu

Renfa Li[3], Qiang Wu[3]
[3]School of Computer and Communication
Hunan University
Hunan, P.R. China
lirenfa@vip.sina.com

***Abstract* - Hardware/Software partitioning is one of the most important problems in the embedded system co-design. Based on a model using the task flow graph, this paper presents a new method that combines the search space smoothing and discrete particle swarm optimization to tackle the problem. Experimental results demonstrate that this new method can reach the best quality of solutions at a relatively low time cost.**

***Keywords* - Hardware/Software partitioning, Discrete particle swarm optimization, Search space smoothing**

I. INTRODUCTION

More and more embedded systems are taking benefits from the emerging new architectures containing processors of various components such as Field Programmable Gate Arrays (FPGA) and Application Specific Integrated Circuits (ASIC). One of the widely used approaches for designing these systems is the hardware/software co-design, which consists of a succession of steps ranging from the functional specification of the system parts to their synthesis. The most important step is the hardware/software partitioning that has a strong impact on cost/performance of the final products. Many optimization algorithms, such as Genetic Algorithm (GA) [1] [2], Tabu Search (TS) [3] [4], Simulated Annealing (SA) [5], Ant Colony Optimization (ACO) [6], and Particle Swarm Optimization (PSO) [7], have been proposed to deal with this issue. Eberhart and Kennedy proposed a Discrete version of PSO (DPSO) [8] by redefining the trajectories of a population of particles as changes in the probability that a coordinate takes on a 0 or 1 value. DPSO has been found to be a highly efficient parallel optimization method in scientific and engineering fields, and also used for the hardware/software partitioning [9] [10].

Although the local search algorithms used in the aforementioned optimization approaches are very effective ways to solve combinatorial optimization problems, they often stuck at a locally optimum configuration due to the rugged terrain surface of the search space. In the meantime to deal with the Traveling Salesman Problem (TSP), Gu et al. originally proposed the Search Space Smoothing (SSS) method [11], which turns out to help overcome local optimum trenches and thus obtain better results for local search algorithms at a cost of linearly increased run-time. Since then, many other efforts have been made to apply this method to combinatorial optimization problems such as VLSI design [12]. Wu et al. in 2004 first applied this method to solve the hardware/software partitioning problem [13], and the results revealed the high potential of the method on this issue.

This paper presents our design of a new hardware/software partitioning method using a search space smoothing technique to guide discrete particle swarm optimization in finding the best solution. To prove its effectiveness, we compare this new method with the original DPSO as well as Improved Simulated Annealing algorithm (ISA) in some randomly generated instances. Experimental results indicate that our method can reach the best quality of solutions at a relatively low time cost.

The rest of the paper is organized as follows. Section II describes the model of hardware/software partitioning problem that we incorporate. Section III and Section IV discuss the basic ideas and characteristics of the DPSO and SSS, respectively. Section V gives the workflow and operations of the new method, Search Space Smoothing combined with Discrete Particle Swarm Optimization (SSS+DPSO). Section VI shows the experimental results with comparisons. Finally, Section VII concludes this paper.

II. HARDWARE/SOFTWARE PARTITIONING MODEL

The main function of the system under consideration is usually specified by a high-level programming language like C or Java first, and then the function will be mapped into a task graph that can be processed by optimization algorithms for different purposes. The hardware/software partitioning can be modeled by the task flow graph, as shown in Fig. 1. In the task flow graph, nodes represent tasks to be partitioned, and directed edges between them denote their relationship.

Then, given a process task flow graph G = (*N*, *E*), the set of nodes is partitioned into two subsets:

G = ({*HW*, *SW*}, *E*), where $HW \cup SW = N$, $HW \cap SW = \varnothing$.

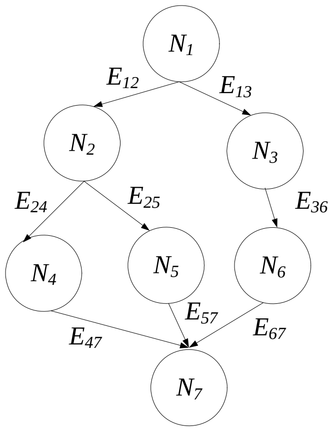


Figure 1. A 7-node task graph.

After the task flow graph has been generated, the partitioning information, such as the hardware cost, software cost, and communication cost, which are captured as weights associated with the nodes and edges, have to be merged into an objective function. Then, the objective function should be optimized under some constrains by the partitioning method. In this paper, the objective function and cost estimation methods are borrowed from [3]. However, the hardware/software partitioning method proposed here is not limited to such an objective function. Other functions are also applicable. The objective function of [3] is the following:

$$C(HW,SW) = Q_1 \times \sum_{(i,j \in cut)} W1^E{}_{ij} + Q_2 \times \frac{\sum_{i \in HW} \frac{\sum_{\exists(ij)} W2^E{}_{ij}}{W1^N{}_i}}{N_H} - Q_3 \times \left[\frac{\sum_{i \in HW} W2^N{}_i}{N_H} - \frac{\sum_{i \in SW} W2^N{}_i}{N_S} \right] \qquad (1)$$

where *HW* and *SW* represent the sets of hardware and software parts, respectively; N_H and N_S are the number of nodes in these two sets; *cut* is the set of edges cut by the partitioning; $W1^N{}_i$ and $W2^N{}_i$ represent the Computation Load (CL) and Relative Computation Load (RCL) [14] of node *i*, respectively; $W1^E{}_{ij}$ and $W2^E{}_{ij}$ represent the total amount of communication data and the degree of synchronization of edge<i, j>, respectively. The first term captures the amount of communication cost between hardware and software parts. The second term stimulates placement into hardware, which reduces amount of interaction with the rest of the system relative to their computation load. It can improve parallelism between tasks executed by hardware. The third term pushes tasks with high node weight into the hardware partition and those with low node weight into the software partition, by increasing the difference in the average weight of nodes between hardware/software parts. Q_1, Q_2, Q_3 are used to trade off among these terms, hence guide the partitioning procedure to designer's intention. In this paper, the goal is to minimize the function value of (1). In addition, the total hardware and software costs have to be limited by the following constraints:

$$\sum_{i \in HW} H\_\mathrm{cost}_i \le Max^H \qquad (2)$$

$$\sum_{i \in SW} S\_\mathrm{cost}_i \le Max^S \qquad (3)$$

where H_cost_i and S_cost_i represent the hardware cost and software cost of node *i*, respectively; Max^H and Max^S represent specified limits of hardware and software constraints, respectively.

III. Discrete Particle Swarm Optimization

In our hardware/software partitioning model, the problem is similar to finding the largest connected sub-graph that meets certain conditions. Because it is an intractable problem in graph theory, many optimization techniques have been proposed to find the best solution.

Eberhart and Kennedy in 1995 proposed the Particle Swarm Optimization (PSO) [15] as an optimization technique for use in a real-number space. A potential solution to a problem is represented as a particle having coordinates x_{id} and rate of change v_{id} in a *D*-dimensional space. Each particle *i* maintains a record of the position of its previous best solution in a vector called $pbest_{id}$. An iteration comprises the evaluation of each particle and stochastic adjustment of v_{id} in the direction of particle *i*'s best previous position along with the best previous positions of the particles in the neighborhood, which can be defined in innumerable ways: for example, a typical implementation evaluates particle *i* in a neighborhood consisting of itself, particle *i* – 1, and particle *i* + 1. The vector $gbest_d$ is assigned with the index value of the particle having the best performance so far in the search space. Thus, in the original version, particles move by the following equations:

$$\begin{aligned} v_{id}^{k+1} &= W v_{id}^{k} \\ &+ c_1 r_1 (pbest_{id} - x_{id}^{k}) \\ &+ c_2 r_2 (gbest_{d} - x_{id}^{k}) \end{aligned} \tag{4}$$

$$x_{id}^{k+1} = x_{id}^{k} + v_{id}^{k+1} \tag{5}$$

where x_{id}^{k} is the current position of individual i at iteration k with velocity v_{id}^{k}. Besides, W is the inertia weight factor, c_1 and c_2 are acceleration constants, and r_1 and r_2 are uniform random numbers between 0 and 1.

The PSO has been found to be robust in solving problems featuring nonlinearity, non-differentiability, multi-peak, and some more complex optimization. However, many optimization problems exist in a space featuring discrete, qualitative distinctions between variables, such as in hardware/software partitioning problems. To solve this kind of problems, Eberhart and Kennedy in 1997 proposed the Discrete Particle Swarm Optimization (DPSO) [8] in which original PSO is redefined such that a particle moves in a state space restricted to either 0 or 1 on each dimension, and each v_{id} represents the probability of bit x_{id} taking value 1. Thus, the $pbest_{id}$ and x_{id} are integers in {0, 1} and v_{id} is constrained to the interval [0.0, 1.0]. A logistic transformation is used to accomplish this last modification. DPSO can be obtained by replacing (5) with (6).

$$\begin{aligned} &if\,(rand() < S(v_{id}^{k+1})) \\ &then\; x_{id}^{k+1} = 1; \\ &else\; x_{id}^{k+1} = 0 \end{aligned} \tag{6}$$

where $S(v) = 1/(1+e^{-v})$, which is a sigmoid limiting transformation function, and *rand()* is a quasi-random number selected from a uniform distribution in [0.0, 1.0].

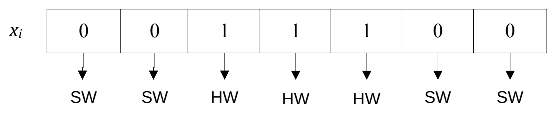


Figure 2. An example of a particle used in hardware/software partitioning (HW and SW stand for hardware and software, respectively).

DPSO can be easily implemented to deal with the hardware/software partitioning problem for the partitioning model shown in Section II. The number of dimensions D equals to the number of tasks to be partitioned. As shown in Fig. 2, each particle is a string of 0/1-bits, which represents a solution of hardware/software partitioning. For example, if $x_{i3} = 1$, the third task will be executed by hardware. Fig. 3 shows a hardware/software partitioning result of the tasks shown in Fig. 1.

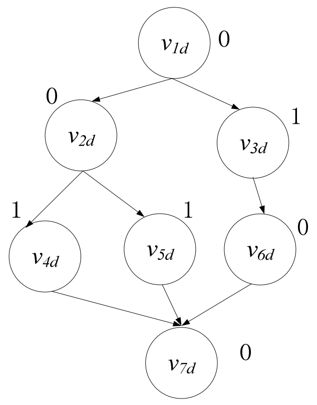


Figure 3. A partitioning result in a task graph.

The procedure of the DPSO algorithm can be described as follows:

Step 1: generate the initial particle swarm randomly; k is used as iteration time,

Step 2: calculate the objective function for each particle according to (1); update $pbest_{id}$ and $gbest_{d}$,

Step 3: update the particle swarm according to (4) and (6),

Step 4: go back to Step 2 until the algorithm finds the final solution (when any termination condition is met, which is usually maximum generation limit, run-time limit, or a predefined threshold for convergence of the algorithm).

IV. Search Space Smoothing

Local search, such as DPSO, is an effective technique to cope with the overwhelming computational intractability of NP-hard combinatorial optimization problems. Given a minimization problem with objective function f and feasible region F, in a typical local search algorithm, for each solution point $x_i \in R$, there is an associated predefined neighborhood $N(x_i) \subset R$. Given a current solution point $x_i \in R$, set $N(x_i)$ is searched for a point x_{i+1} evaluated for $f(x_{i+1})$ while satisfying $f(x_{i+1}) < f(x_i)$. If such a point exists, it becomes a new current solution point, and the process is repeated. Otherwise x_i is retained as a local optimum with respect to $N(x_i)$. Then a set of solution points is generated, and each of them is locally improved within its neighborhood. To apply local search to a particular problem, one needs to specify only the neighborhood structure and the randomized procedure for obtaining a feasible starting solution point. However, the major weakness of local search is that it has a tendency to get stuck at a locally optimum point, unable to find the global minimum.

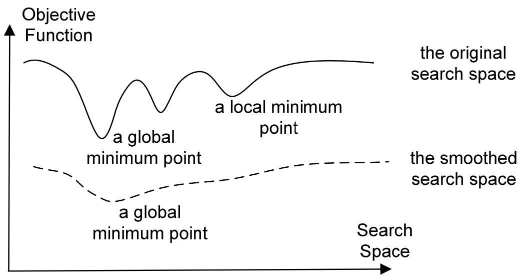


Figure 4. Illustration of smoothing a search space.

In order to improve the performance of a local search algorithm, Gu et al. [11] has developed the Search Space Smoothing technique in 1994 which temporarily limits the number of local minimum points in the search space and then gradually releases limitation as it approaches to the best solution. The basic idea of the method can be explained as follows. Assume there is a search space with many local minimum points (see Fig. 4), where a solution point could be easily trapped. A smoothed search space, in which some local minimum points are temporarily filled (i.e., flattened) and thus they no longer cause trapping, will be used to approximate the original search space. In the smoothed space, the number of local minimum points is reduced, and optimistically the smoothing process only changes the metric characteristics of the search space and leaves its topological structure unchanged.

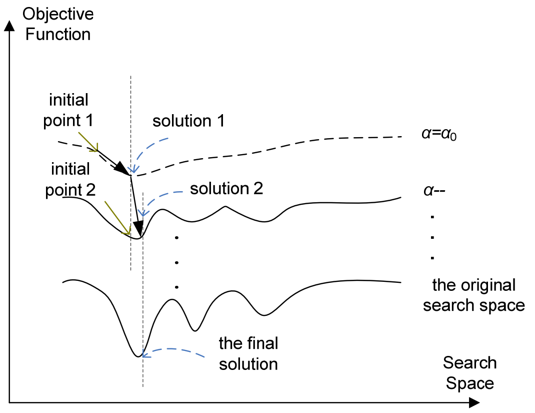


Figure 5. A gradually approximated smoothing proces.

If the global minimum point found in the smoothed search space is used as an initial starting point in the original search space, as illustrated in Fig. 5, not only would the probability of finding the global minimum point in the original search space be increased considerably but also search time would be dramatically reduced. To increase the chance of finding the global minimum point in the smoothed space, a strong smoothing operation that can produce a flatter search space may be attractive. Unfortunately, however, it may lead to losing some heuristic guidance information. On the contrary, if a weak smoothing operation is applied, the topological structure of the smoothed search space is similar to the original one, and thus, it may result in less reduction in the number of local minimum points of the original search space. This dilemma can be resolved by using a gradually approximated smoothing scheme [8], which executes a series of search space smoothing operations as follows.

A smoothing factor, α, is introduced to characterize the degree of a smoothing operation. If $\alpha = 1$, no smoothing operation is applied, and thus the search space is the same as the original search space. If $\alpha > 1$, a smoothing operation is applied, and thus the smoothed search space is flatter than that of the original search space. If $\alpha >> 1$, the smoothing operation has a stronger effect, resulting in a nearly flat search space.

The following procedure is a typical gradually approximated smoothing scheme. Initially the search space is most smoothed, and then each later search space is made to be a less smoothing of the earlier search space. Therefore, the solutions of a more smoothed, flatter search space are used to guide the search of those in the more rugged search space.

Step 1: initialize a smoothing factor, $\alpha >> 1$, and randomly generate an initial solution, X_{in}, then run a local search algorithm to get its new solution, X_{out},

Step 2: let $\alpha = \alpha - 1$ and $X_{in} = X_{out}$, then run the local search algorithm in the new search space to get its solution,

Step 3: repeat Step 2 until $\alpha = 0$.

V. Search Space Smoothing Combined with Discrete Particle Swarm Optimization for Hardware/Software Partitioning

A. *SSS+DPSO Algorithm for Hardware/Software Partitioning*

The key idea of SSS+DPSO is that DPSO can make use of a series of smoothed problem instances gradually generated by the Search Space Smoothing (SSS) to optimize the hardware/software partitioning. It starts from an approximate trivial problem instance, i.e., the instance with a fairly flat search space, and then finds the solution to this simplified problem using DPSO. The solution of the problem instance is then taken as the initial point to the next problem instance that has a slightly more complex (less smoothed) search space, and then this new problem is again solved by DPSO. The above procedure is repeated until the final problem instance generated with the original search space is solved. The idea described here is processed in a workflow as illustrated in Fig. 6.

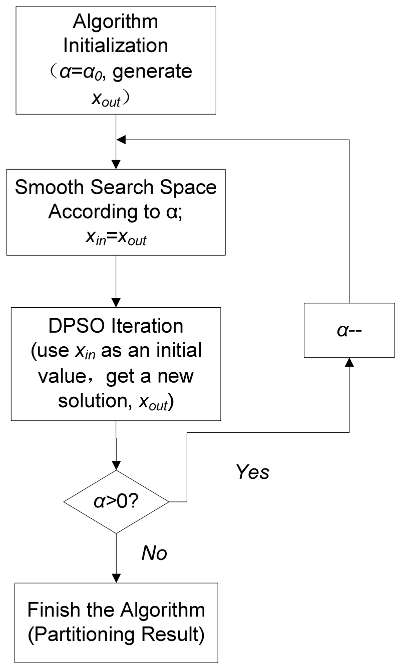

Algorithm Initialization (α=α_0, generate x_{out})
Smooth Search Space According to α; x_{in}=x_{out}
DPSO Iteration (use x_{in} as an initial value, get a new solution, x_{out})
α--
α>0?
Yes
No
Finish the Algorithm (Partitioning Result)

Figure 6. The workflow of the SSS+DPSO algorithm.

According to the workflow, we combine the SSS and DPSO as follows:

Step 1, select an initial smoothing factor, α >> 1, and randomly generate an initial solution, X_{out}, and let X_{in} = X_{out},

Step 2, begin the iterative search space smoothing according to the smoothing factor, run the DPSO algorithm (according to the procedure described in Section III) in the new search space (use X_{in} as an initial solution), and obtain a new solution, X_{out},

Step 3, let X_{in} = X_{out}, α = α – 1 (the smoothed search space will gradually approximate to the original search space),

Step 4, go back to Step 2 if α > 0, otherwise finish the algorithm and output X_{out} as the final solution.

B. Smoothing Operation of SSS+DPSO

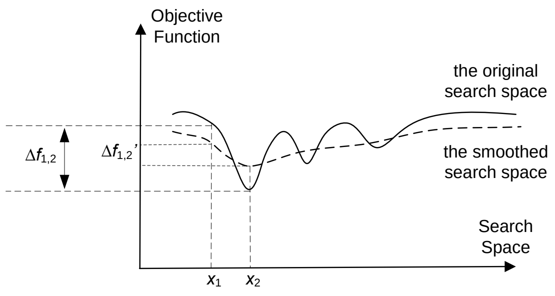


Figure 7. Smoothing Operation

We here first discuss how to control the effect of smoothing of the search space in our partitioning problem. As shown in Fig. 7, the solid line represents the original search space of the problem, whereas the dotted line represents the smoothed search space. x_1 and x_2 are two solutions that are randomly chosen. $f(x_1)$ and $f(x_2)$ represent the evaluated values of the objective function for x_1 and x_2 in the original search space, respectively. $f(x_1)'$ and $f(x_2)'$ represent the evaluated values of the objective function for x_1 and x_2 in the smoothed search space, respectively. $\Delta f_{1,2} = (f(x_1) - f(x_2))$ and $\Delta f_{1,2}' = (f(x_1)' - f(x_2)')$ are the difference between the values of the objective function, respectively. Obviously, for the same set of solutions, x_1 and x_2, the difference between the values of the objective function in the original search space is larger than that of the smoothed search space (as shown in Fig. 7, $\Delta f_{1,2}$ > $\Delta f_{1,2}'$). Thus, reducing or enlarging the difference between the values of objective function will produce an effect of smoothing. Note that in an extreme case of smoothing, if all the nodes and edges in our hardware/software partitioning model have the same weights, the search place is absolutely flat (as a straight line). Any partitioning result will lead to the same value of the objective function, since there is no local minimum point in the search space any more.

According to the analysis, the search space of the hardware/software partitioning problem can be smoothed by reducing the difference between the evaluated values of objective function of the solutions in the search space. As described in (1), the value of objective function is determined by the weights of nodes and edges in our hardware/software partitioning model. Thus, we can use the equations in (7) for the weights of nodes to lead the effect of search space smoothing, which are normalized between 0 and 1 beforehand. Since the gradually approximated smoothing is used in SSS+DPSO, the weights of nodes are determined by the smoothing factor for a hardware/software partitioning problem instance.

$$w_i(\alpha) = \begin{cases} \overline{w} + (w_i - \overline{w})^\alpha & \text{if } w_i \ge \overline{w} \\ \overline{w} - (\overline{w} - w_i)^\alpha & \text{if } w_i < \overline{w} \end{cases}$$
$$\overline{w} = \frac{1}{N} \sum_{i \in N} w_i \qquad (7)$$

where w_i represents the weight of node *i*; $\overline{w}$ is the average weight among all the nodes; *N* represents the number of nodes in the task flow graph; α has been defined in Section IV as the smoothing factor. When $\alpha \to \infty$, $w_i(\alpha) \to \overline{w}$, thus producing a graph with equal weights. When α = 1, $w_i(\alpha)$ returns to its original value, so the graph returns to its original shape. The weights of edges are adjusted in the same method as the nodes, and thus the description is omitted for succinctness.

VI. Experimental Results and Performannce Comparisons

We generate six sets of graphs by the TGFF [16] tool for each node number of 10, 20, 50, 100, 200, 400, and there are 50 graphs in each set (thus 300 graphs in total). In many earlier papers [3][5][14], the advantages of the PSO and SA in dealing with the HW/SW partitioning problem over other local search algorithms, such as GA, TS (Tabu Search) and KL(Kernighan-Lin), have been confirmed already. Thus, we chose to combine the best algorithms in previous work with the Search Space Smoothing (SSS) technique and then analyzed both the solution quality and computation time. We compared our implementation with Discrete Swarm Optimization (DSPO) and the Improved Simulated Annealing (ISA) algorithm given by [13], which also takes advantage of the search space smoothing technique. All the algorithms were executed on a computer with 3.0GHz Intel dual-core processor and 1GB memory. Parameters of each algorithm are as follows:

SSS+DPSO: for 100 particles, the inertia weight factor W=2, the acceleration constants c_1=c_2=0.2, the smoothing factor α_0=10, and the maximum number of iteration is 800.

DPSO: for 100 particles, the inertia weight factor W=2, the acceleration constants c_1=c_2=0.2, the algorithm ends when the changes of objective function are less than 0.08.

ISA: the initial temperature is 10^{16}, the end temperature is $1/10^6$, the cooling ratio is 0.9 and the smoothing factor α_0=10.

A. Solution Quality

Each algorithm runs 150 times on one instance, and the solutions that lead to smaller values of the objective function are considered as better solutions. We found that the quality of the best solution (the best one in all solutions) of each algorithm is in little difference, and thus, we compare the number of times that each algorithm correctly finds the best solution instead. Table 1 (see at the tail of the paper) shows the results where T*n* represents the graph containing *n* nodes. Fig. 8 is the illustration of the data shown in Table 1. In the illustration, the horizontal axis represents the number of nodes of each instance while the vertical axis represents the number of times the algorithms find the best solution. Each curve in the illustration represents the group of data of the corresponding algorithm as labeled.

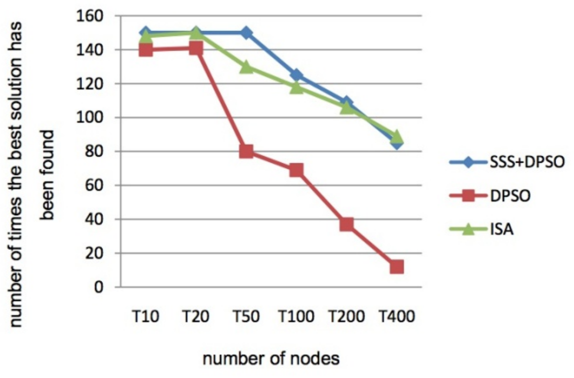


Figure 8. The number of times that each algorithm finds the best solution.

The figure shows that all three algorithms perform very well when the number of nodes is below 20 but when the number exceeds 20, the number of times that DPSO finds the best solution is largely decreased. In cases of SSS+DPSO and ISA, they perform in the same level except when the number of nodes is in the range of 50 to 200, SSS+DPSO performs better. The fact that both SSS+DPSO and ISA use search space smoothing to guide their local search algorithms proves that search space smoothing can effectively improve the solution of DPSO. In the next sub-section, the run-times of the algorithms for obtaining such solutions are compared.

B. Run-Time

Table 2 (see at the tail of the paper) gives the average run-time (in sec.) of each algorithm. Fig. 9 is the illustration of the data shown in Table 2. In the illustration, the horizontal axis represents the number of nodes of each instance while the vertical axis represents the run-time of the algorithms. Each curve in the illustration represents the group of data obtained by executing the corresponding algorithm as labeled.

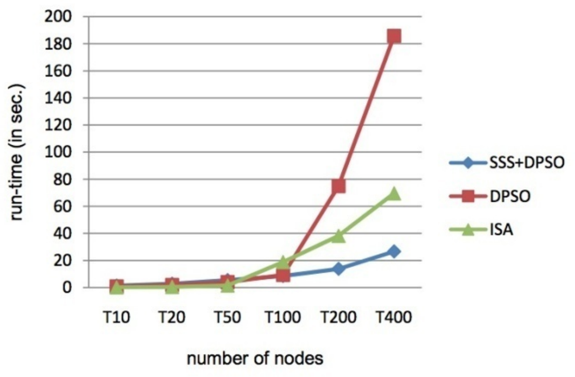


Figure 9. The average run-time of each algorithm.

The figure shows that all three algorithms perform very fast when the number of nodes is below 100, but when the number exceeds 100, the curves of DPSO and ISA show a trend to climb much faster than the curve of SSS+DPSO. Especially when the number reaches 400, the advantage of SSS+DPSO becomes more obvious. It is a notable characteristic that SSS+DPSO is almost three to ten times faster than the others in dealing with large size partitioning problems, which are very common in the real world.

VII. Conclusion

In this paper, we have discussed a discrete version of particle swarm optimization that operates on discrete binary variables and a search space smoothing technique that relies on the effectiveness of using intermediate solutions of smoothed search space to guide the search of increasingly complex problem instances. Taking advantages of these methods, a new method combining search space smoothing and discrete particle swarm optimization has been proposed to deal with the hardware/software partitioning.

Experimental results indicate that this new method can stably increase the quality of solutions as the probability of finding the best solution is more than 56% (85 in 150 times) even in a 400-node graph. In the meanwhile, it is worth mentioning that the method has the lowest time cost in graphs with large number of nodes (more than 100 nodes) as compared to DPSO and ISA. Our further exploration will focus on implementing this method on a practical application [17] for run-time hardware/software partitioning.

ACKNOWLEDGMENT

This research was funded in part by the U.S. National Science Foundation Grant 0834682 and in part by the National Natural Science Foundation of China Grant 60973030.

REFERENCES

[1] D. Saha, A. Basu and R. S. Mitra, "Hardware Software Partitioning Using Genetic Algorithm," Proc. of the 10th International Conference on VLSI Design: VLSI in Multimedia Applications, pp. 155, 1997.

[2] S. Zheng, Y. Zhang and T. He, "The Application of Genetic Algorithm in Embedded System Hardware-Software Partitioning," Proc. of the 2009 International Conference on Electronic Computer Technology, pp. 219-222, 2009.

[3] L. Li, and M. Shi, "Software-Hardware Partitioning Strategy Using Hybrid Genetic and Tabu Search," Proc. of 2008International Conference on Computer Science and Software Engineering, pp. 83-86, 2008.

[4] P. Elis, Z. Peng, K. Kuchcinski, and A Doboli, "Hardware/software partitioning with iterative omprovment heuristics," Proc. of 9th International Symposium on System Synthesis, pp. 71-76, 1996.

[5] P. Elis, Z. Peng, K. Kuchcinski, and A Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search," Design Automation for Embedded Systems, vol. 2, No. 1, pp. 5-32, Jan. 1997.

[6] M. Koudil, K. Benatchba, S. Gharout, and N. Hamani, "Solving Partitioning Problem in Codesign with Ant Colonies," Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, vol. 3562/2005, pp. 324-337, 2005.

[7] A. Bhattacharya, A. Konar, S. Das, C. Grosan, and A. Abraham, "Hardware Software Partitioning Problem in Embedded System Design Using Particle Swarm Optimization Algorithm," Porc. of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems, pp. 171-176, 2008.

[8] J. Kennedy and R. C. Eberhart, "A Discrete Binary Version of the Particle Swarm Algorithm," IEEE International Conference on Systems, Man, and Cybernetics, vol. 5, pp. 4104-4108, Oct. 1997.

[9] A. Farmahini-Farahani, M. Kamal, and S. M. Fakhraie, "HW/SW Partitioning Using Discrete Particle Swarm," Proc. of the 17th ACM Great Lakes Symposium on VLSI, pp. 359-364, 2007.

[10] T. Eimuri and S. Salehi, "Using DPSO and B&B Algorithm for Hardware/Software Partitioning in Co-desing," Proc. of International Conference on Computer and Development, pp. 416-420, 2010.

[11] J. Gu and X. Huang, "Efficient local search with search space smoothing: a case study of the traveling salesman problem (TSP)," IEEE Transactions on Systems, Man and Cybernetics, vol. 24, pp. 728-735, May. 1994.

[12] S. Dong, X. Hong, S. Zhou, and J. Gu, "Efficient VLSI Module Placement with Solution Space Smoothing," Proc. of International Conference on Communication Circuits And Systems (ICCCAS02), vol.2, pp. 1396-1400, 2002.

[13] Q. Wu, J. Bian, H, Xue, Y. Fan, W. Wu, X. Hong, and J. Gu, "Applying search space smoothing technique to hardware:software partitioning: Experiments and Analysis," Proc. of 5th International Conference on ASIC, vol.1, pp. 85-88, Oct. 2003.

[14] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "Performance Guided System Level Hardware/Software Partitioning with Iterative Improvement Heuristics," Res. Rep. LiTH-IDA-R-95-26, Dep. of Comp. and Inf. Science, Linkoping University, 1995.

[15] J. Kennedy and R. C. Eberhart, "Particle Swarm Optimization," Proc. IEEE International Conference on Neural Networks, vol. 4, pp. 1942-1948, 1995.

[16] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task Graphs for Free," Proc. of the 6th International Workshop on Hardware/Software Codesign, pp. 97-101, 1998.

[17] P. S. Vaidya, J. J. Lee, F. Bowen, Y. Du, C. H. Nadungodage, and Y. Xia, "Symbiote: A Reconfigurable Logic Assisted Data Stream Management System (RLADSMS)," Proc. of the 2010 International Conference on Management of Data, pp. 1147-1150, 2010.

TABLE I. THE NUMBER OF TIMES THAT EACH ALGORITHM FINDS THE BEST SOLUTION

	T10	T20	T50	T100	T200	T400
SSS+DPSO	150	150	150	125	109	85
DPSO	140	141	80	69	37	12
ISA	148	150	130	118	106	89

TABLE II. THE AVERAGE RUN TIME OF EACH ALGORITHM (IN SEC.)

	T10	T20	T50	T100	T200	T400
SSS+DPSO	1.43	2.79	5.43	8.53	13.77	26.63
DPSO	0.82	1.95	4.01	9.28	74.83	185.71
ISA	0.13	0.49	1.50	18.84	38.17	69.52

Dynamic reconfiguration on a dynamically reconfigurable vision-chip architecture

Amarjargal Gundjalam and Minoru Watanabe
Electrical and Electronic Engineering
Shizuoka University
3-5-1 Johoku, Hamamatsu, Shizuoka 432-8561, Japan
Email: tmwatan@ipc.shizuoka.ac.jp

Abstract— **Recently, a high-speed image recognition function that is superior to the image recognition speed of the human eye is demanded for autonomous vehicles and robots. A dynamically reconfigurable vision-chip architecture with a holographic memory and large-bandwidth optical connections has been proposed to meet that need. Such a dynamically reconfigurable vision-chip architecture can realize high-speed image recognition and high-speed template matching operations. However, in the architecture, while an image is detected at every 1 ms, a gate array on an ORGA-VLSI is frequently reconfigured to recognize it. Since the reconfiguration photodiodes and image detection photodiodes are placed close together, an emergent problem is that the background diffraction light produced for dynamic reconfiguration degrades the image quality. This paper therefore presents a demonstration based on a two-configuration-context dynamically reconfigurable vision-chip architecture to clarify that the deterioration in image quality is slight.**

I. Introduction

Recently, a high-speed image recognition function with superior image recognition speed to that of a human eye is demanded for autonomous vehicles and robots [1]. In robots that are currently available, processor-based embedded systems are typically implemented [2][3]. Such embedded systems consist of one or two image sensors, a processor, and memory. However, the performance of current embedded systems is insufficient to execute real-time image recognition.

To date, to improve the performance of image processing operations, analog-type vision chips were fabricated using standard CMOS processes [4][5]. The analog-type vision chips can execute spatial filter operations of smoothing filters and Laplacian of Gaussian filters along with detection of images, just as the human retina can. Nevertheless, analog-type vision chips are unable to execute image recognition operation because of the lack of memory. On the other hand, digital vision chips with a single instruction multiple data (SIMD) type processor have been developed [6][7]. They can detect images and can execute many visual operations. However, since the processing power of the simple SIMD processors is not high compared with that of current general-purpose processors [8], and because the amount of memory inside the chips is not large, the operations are also limited to simple operations, just like spatial filter operations. As a result, the digital vision chips can not support the image recognition function.

To recognize many images on a robot, many template images must be stored in memory and must be sent quickly from the memory to the processor. For example, assuming that the system receives an external image with 1 million pixels at every 1 ms and assuming that the system must execute template matching operations of 1 million template images with the same million pixels within 1 ms, the transfer speed from the memory to the processor and the processor's template matching operation reaches 1 Petapixel/s. Therefore, in image recognition systems, the most burdensome operation is the template-matching operation. Even if the latest VLSI technology is applied to the template matching operation, the realizable processing power and the realizable transfer rate are insufficient.

Therefore, to realize high-speed template matching operation, a dynamically reconfigurable vision-chip architecture with a holographic memory and a large-bandwidth optical connection has been proposed [11][12]. The architecture is based on optically reconfigurable gate arrays (ORGAs) [13]–[15]. In this architecture, some of the many photodiodes for optical reconfigurations can be used for detecting external images. In addition, in the vision-chip architecture, a high-density gate array can execute heavy template matching operations in a perfectly parallel manner. Therefore, the dynamically reconfigurable vision-chip architecture can support the performance of 100,000 template-matching operations per millisecond. That architecture offers extremely high performance.

However, in the architecture, while an image is detected at every 1 ms, a gate array on an ORGA-VLSI is frequently reconfigured at nanosecond-order to recognize it. Since the reconfiguration photodiodes and image detection photodiodes are placed close to each other, an emergent issue is that the background diffraction light produced by dynamic reconfigurations degrades the image quality. Therefore, this paper presents a demonstration based on a two-configuration context dynamically reconfigurable vision-chip architecture in order to clarify that only very slight deterioration in image quality occurs.

II. Dynamically reconfigurable vision chip architecture

Figure 1 presents an overview of a dynamically reconfigurable vision-chip architecture [11]. The dynamically reconfigurable vision-chip architecture comprises a laser array, a holographic memory, a beam splitter, a lens array, an imaging lens, and a fine-grain optically reconfigurable gate array VLSI (ORGA-VLSI) [13][14][15]. The dynamically reconfigurable vision-chip architecture enables an image input for the ORGA-VLSI in addition to high-speed optical reconfiguration.

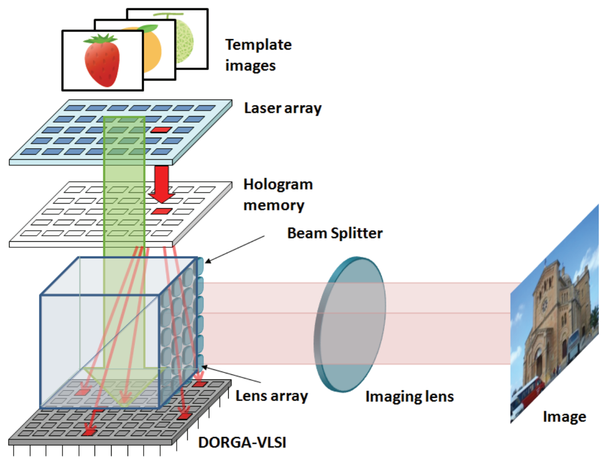


Fig. 1. Associative operations on a dynamically reconfigurable vision chip architecture.

A. Image input

An ORGA-VLSI has many photodiodes for optical reconfiguration. Therefore, some photodiodes can be used also for sensing an image. In this architecture, an image is given through an imaging lens. Then, each pixel of the image is focused onto a photodiode on the ORGA-VLSI by a lens array through a beam splitter. The lens array also plays a role in increasing the open area ratio of sensors. Always, photodiodes in an image sensor are designed to be as large as possible to increase the sensing sensitivity. However, the photodiodes in ORGA-VLSIs are designed to be as small as possible to increase the gate density. Therefore, the photodiode sensitivity of ORGA-VLSIs as an image sensor is not high. However, the photodiode's weak sensitivity can be improved to that of larger photodiodes on image sensors using the lens array. Therefore, in this architecture, the open area ratio problem is beyond consideration for that reason. This architecture aims at sensing an external image at every 1 ms.

B. Image processing operations

The image-processing operations that are performed on a dynamically reconfigurable vision-chip architecture are executed based on an ORGA-VLSI [13][14][15]. The ORGA-VLSI takes a fine-grained gate array including look-up tables as well as field programmable gate arrays (FPGAs) [9][10]. In an ORGA, many dynamic reconfiguration contexts are provided from a holographic memory. They are addressed by a laser array. Various visual operations, such as template-matching operations and filtering operations can be executed using high-speed dynamic reconfiguration. Circuits dedicated for each operation are implemented dynamically for the gate array at a clock cycle. This architecture can support nanosecond-order dynamic reconfiguration.

C. Associative operation

In this architecture, when a template matching operation is executed, a circuit to recognize the template image is implemented onto a gate array. A simple example of four-bit binary template matching operations is presented in Fig. 2. Each pixel of the images has a binary value. The upper side figures show five sample template images. In this case, template-matching operations for the sample images are executed using recognition circuits implemented onto look-up tables of a gate array, as shown on the lower side of the figures. In the case of the No. 1, the circuit becomes a 4-input AND circuit. In the case of the No. 2, the circuit consists of a 4-input AND circuit and an inverter. In advance, such circuits are converted to corresponding holographic memory patterns and are stored on a holographic memory. While the operation is executed, such template matching operation circuits, No. 1, No. 2, No. 3, .., No. 5 are dynamically read out from the holographic memory and are programmed onto a gate array in turn. Each time a configuration is completed, a single template matching operation is executed on the gate array. As a result, among five configurations and template matching executions, one operation can find its own image.

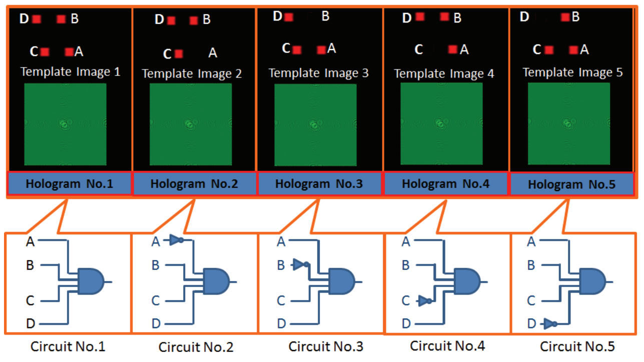


Fig. 2. Examples of binary template images, corresponding holographic memory patterns, and operation circuits.

D. One concern

In the architecture, image detection is assumed to be executed every 1 ms. Consequently, the detection period for one image becomes 0–1 ms. Therefore, image detection is executed full-time. In contrast, in this architecture, optical reconfigurations are executed dynamically at nanosecond-order. Here, a concern is that, at that time, the background diffraction light of dynamic reconfigurations can degrade the quality of a detected image. Reconfiguration photodiodes and image detection photodiodes are placed close together, which makes image quality deterioration unavoidable because of effects of optical reconfiguration. Therefore, this paper presents a demonstration based on a two-configuration context dynamically reconfigurable vision-chip architecture in order to clarify that only very slight deterioration in image quality occurs.

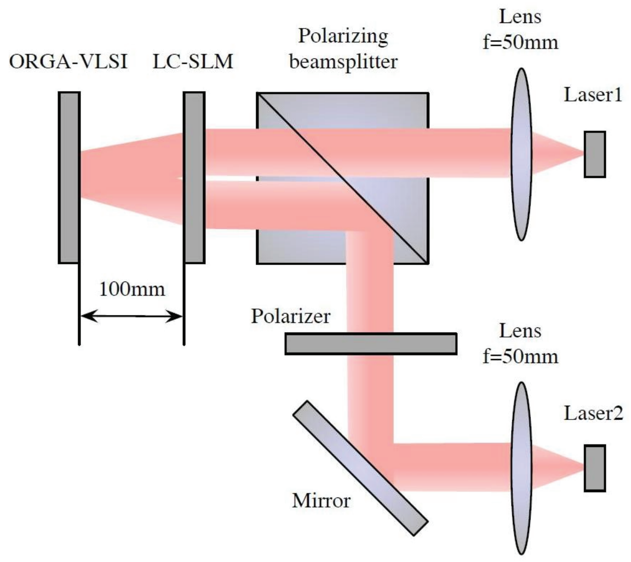


Fig. 3. Block diagram of the experimental system.

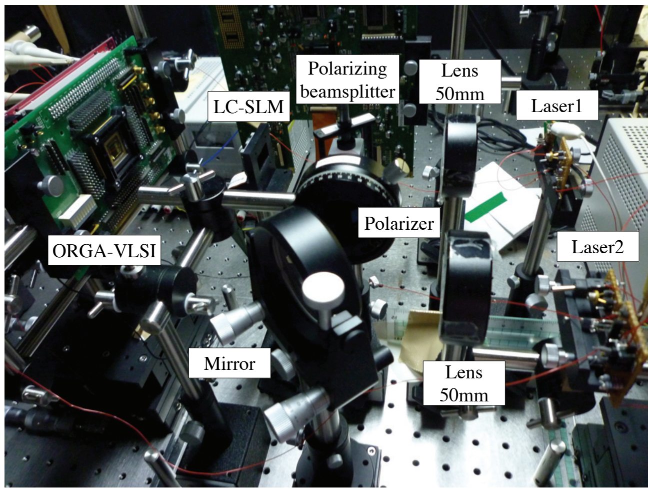


Fig. 4. Photograph of the experimental system.

III. Experimental System

A. Context Generation

Here, the calculation method of a holographic memory is described. A hologram for an ORGA is assumed as a thin holographic medium. A laser aperture plane, a holographic plane, and an ORGA-VLSI plane are parallelized. The laser beam is collimated. The holographic medium comprises rectangular pixels on the holographic plane. The pixels are assumed as analog values. On the other hand, the input object comprises rectangular pixels on the object plane. Its pixels can be modulated to be either on or off. The intensity distribution of a holographic medium is calculable using the following equation.

$$\begin{aligned} H(x_1,y_1) &\propto \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} O(x_2,y_2)\sin(kr)dx_2dy_2, \\ r &= \sqrt{Z_L^2+(x_1-x_2)^2+(y_1-y_2)^2} \end{aligned} \quad (1)$$

In that equation, $O(x_2,y_2)$ is a binary value of a reconfiguration context, k is the wave number, and Z_L signifies the distance between the holographic plane and the object plane. The value $H(x_1,y_1)$ is normalized as 0–1 for minimum intensity H_{min} and maximum intensity H_{max} as the following.

$$H'(x_1,y_1) = \frac{H(x_1,y_1)-H_{min}}{H_{max}-H_{min}}. \quad (2)$$

Finally, the normalized image H' is used for implementing the holographic memory. Other areas on the holographic plane are opaque to the illumination.

B. Experimental system setup

In this experiment, an external image input was emulated using one configuration context. While the emulated image is detected continuously, another configuration context is frequently programmed onto a gate array. At that time, the image quality deterioration was measured. A block diagram and photograph of the experimental system are presented in Figs. 3 and 4. The experimental system was constructed using a liquid crystal spatial light modulator (LC–SLM) as a holographic memory, two semiconductor lasers (DL-8141-035; Sanyo Electric Co., Ltd.) as light sources, and an ORGA-VLSI. The respective power and wavelength of the semiconductor lasers are about 150 mW and 808 nm. Each laser beam of the semiconductor laser was collimated by a lens of 50 mm focal length. The collimated laser beam diameter is about 4.25 mm. The two lasers were controlled by an FPGA board. The LC–SLM is a projection TV panel (L3D07U-81G00; Seiko Epson Corp.). It is a 90° twisted nematic device with a thin film transistor. The panel consists of 1,920 × 1,080 pixels, each having a size of 8.5 × 8.5 μm^2. The LC–SLM is connected to an evaluation board (L3B07-E60A; Seiko Epson Corp.). The board's video input is connected to the external display terminal of a personal computer. Programming for the LC–SLM is executed by displaying a holographic memory pattern with 256 gradation levels on the personal computer display. The ORGA-VLSI was placed 100 mm away from the LC–SLM. In this experiment, a 0.35 μm triple-metal CMOS process-fabricated ORGA-VLSI chip was used [13][14][15]. The photodiodes were constructed between the N-well layer and the P-substrate. The photodiode size and distance between photodiodes were designed respectively as 25.5 μm × 25.5 μm and as 90 μm to facilitate the optical alignment. The gate array structure is fundamentally identical to that of a typical FPGA. The ORGA-VLSI chip includes 4 logic blocks, 5 switching matrices, and 12 I/O bits. In all, 340 photodiodes are used to program the gate array.

IV. Experimental results

Using the previously explained experimental system setup, the degradation in image quality has been estimated. For the estimation, a holographic memory pattern including an emulation image and a NOR circuit was calculated using

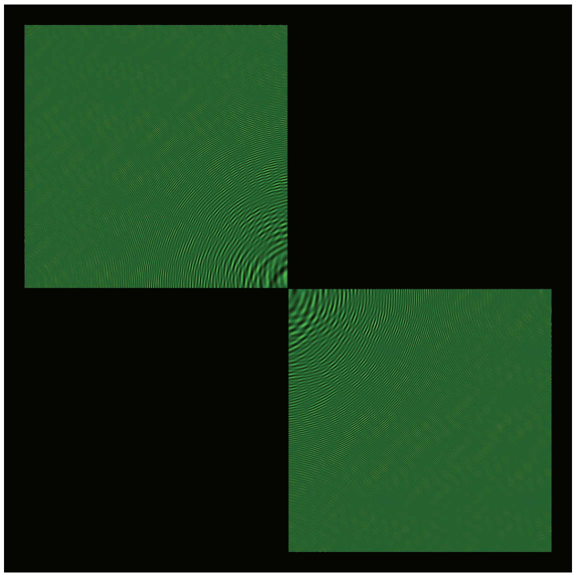


Fig. 5. Holographic memory pattern including an external image and a NOR circuit.

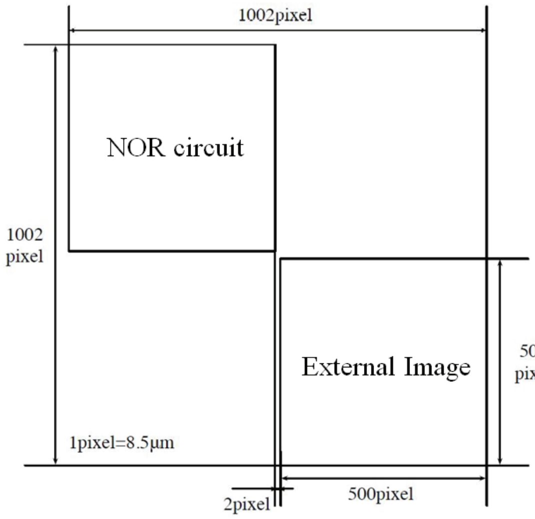


Fig. 6. Mapping of the holographic memory pattern including an external image and a NOR circuit.

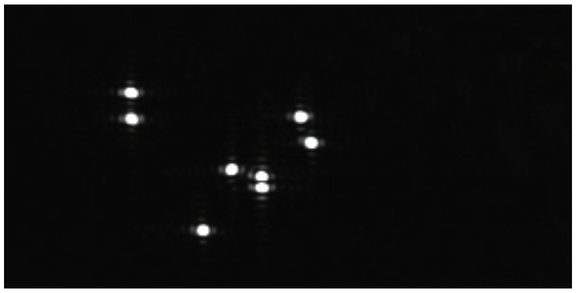


Fig. 7. Emulated external image.

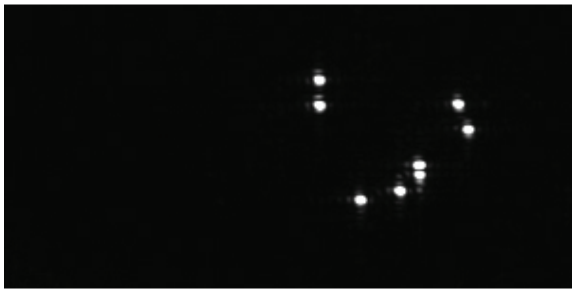


Fig. 8. Configuration context of a NOR circuit.

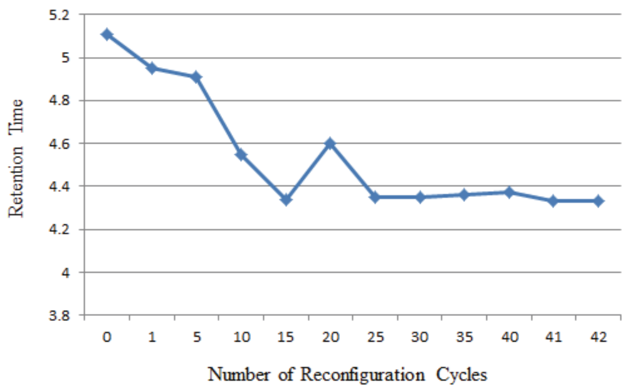


Fig. 9. Relation between the number of reconfiguration cycles and retention time.

Eqs. 1 and 2. The calculated holographic memory pattern and mapping information are presented respectively in Figs. 5 and 6. The holographic memory pattern was implemented onto the LC–SLM. The holographic memory pattern generated a high-contrast image and a high-contrast configuration context of a NOR circuit. The CCD-captured external image and CCD-captured configuration context of a NOR circuit are shown in Figs. 7 and 8. Using the emulated image and the NOR circuit, the retention time was measured. In the ORGA-VLSI, the junction capacitances of target photodiodes are fully charged before sensing an image and a configuration context. After this, an image or/and a configuration context is applied from a holographic memory onto an ORGA-VLSI. In the sensing phase, the amount of discharged electric charge is proportional to the light intensity. Finally, a binary state signifying whether the electric charge in the junction capacitance is retained or not is sensed on the ORGA-VLSI. The retention time is that period for which the junction capacitance of photodiode can retain its electric charge. The equation is shown as follows:

$$T_{retention} = \frac{k_0}{P_1 + N \times P_2}, \qquad (3)$$

where k_0, P_1, P_2, and N respectively represent a coefficient, background light power of the experimental system, the background light power of configuration context, and the number of reconfiguration cycles. In this measurement, the laser-turn-on period was designed as 1.0 ms. When no reconfiguration is executed, the retention time of image detection was measured as 5.11 s. This condition is equal to that of currently available CMOS image sensors without any optical reconfiguration. However, as the number of reconfiguration cycles of a NOR circuit is increased, the retention time is decreased. At the minimum case, the retention time reached 4.33 s.

This result demonstrates that while an external image is detected by a photodiode, the gray scale of the image bit was distorted. Currently, when 42 reconfiguration cycles were executed on the dynamically reconfigurable vision-chip architecture, there was maximum 15 % variation. Image deterioration is dependent upon the contrast ratio of a holographic memory. The contrast ratio of the liquid crystal holographic memory is not good. Always, the ratio is about 20:1. Therefore, in the future, by introducing a volume-type holographic memory, the contrast ratio can be improved to greater than 1000:1. At that time, a greater than 2,000 configuration cycle can be achieved. In such a case, 2,000 template-matching operations can be executed.

V. Conclusion

For use in autonomous vehicles and robots, high-speed image recognition functions that are superior to the image recognition speed of a human eye are demanded. A dynamically reconfigurable vision-chip architecture that can realize high-speed image recognition or a high-speed template matching operation has been developed to satisfy that demand. However, in the architecture, a salient concern is that while an image is detected, a gate array on an ORGA-VLSI must be reconfigured frequently to recognize it. During use, the issue arose that the background diffraction light of dynamic reconfigurations degrades the image quality. However, this paper's experiments have demonstrated that the effect is less than 15 % when 42 reconfiguration cycles are executed on the dynamically reconfigurable vision chip architecture. In the future, the number of reconfiguration cycles can be improved to 2000 by introduction of a volume-type holographic memory. In such a case, 2,000 template-matching operations can be executed.

VI. Acknowledgments

This research is supported by the Ministry of Internal Affairs and Communications of Japan under the Strategic Information and Communications R&D Promotion Programme (SCOPE). The VLSI chip in this study was fabricated in the chip fabrication program of VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Rohm Co. Ltd. and Toppan Printing Co. Ltd.

References

[1] A. Hajiaboli, M. Popovie, "FDTD Analysis of Light Propagation in the Human Photoreceptor Cells," IEEE TRANSACTIONS ON MAGNETICS, vol. 44, no. 6, pp. 1430 - 1433, 2008.

[2] Jong-Il Yoon, Kyoung-Kwan Ahn, "Face Tracking System using Embedded Computer for Robot System," International Joint Conference SICE-ICASE, pp. 5593-5597, 2006.

[3] P. A. Ruetz, R. W. Brodersen, "A realtime image processing chip set," IEEE International Solid-State Circuits Conference, pp. 148-149, 1986.

[4] S. Kameda, T. Yagi "An analog silicon retina with multi-chip configuration," International Joint Conference on Neural Networks, pp. 387- 392, 2003.

[5] J. Ohta, A. Uehara, T. Tokuda, M. Nunoshita, "Pulse-Modulated Vision Chips with Versatile-Interconnected Pixels," IPDPS Workshops 2000: 1063-1071

[6] T. Komuro, S. Kagami, and M. Ishikawa, "A Dynamically Reconfigurable SIMD Processor for a Vision Chip," IEEE JOURNAL OF SOLID-STATE CIRCUITS, vol. 39, no. 1, pp. 265-268, 2004.

[7] J. Dubois, D. Ginhac, M. Paindavoine, B. Heyrman, "A 10 000 fps CMOS Sensor With Massively Parallel Image Processing," IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 43, NO. 3, pp. 706-717, 2008.

[8] Intel Corporation, "Processors," http://www.intel.com/ products/

[9] Altera Corporation, "Altera Devices," http://www. altera.com.

[10] Xilinx Inc., "Xilinx Product Data Sheets," http://www. xilinx.com.

[11] M. Yasuda, M. Watanabe, "Dynamically reconfigurable vision chip architecture," International conference on Field-Programmable Logic and its Applications, pp. 508 - 512, 2010.

[12] H. Nakada, M. Watanabe, "Template matching operations on a dynamically reconfigurable vision-chip architecture," International Symposium on Communications and Information Technologies, pp. 1091-1096, 2010.

[13] S. Kubota, M. Watanabe, "Programmable Optically Reconfigurable Gate Array Architecture and its writer," Applied Optics, Vol. 48, Iss. 2, pp. 302-308, 2009.

[14] N. Yamaguchi, M. Watanabe, "Liquid crystal holographic configurations for ORGAs," Applied Optics, Vol. 47, No. 28, pp. 4692-4700, Oct., 2008.

[15] D. Seto, M. Watanabe, "A dynamic optically reconfigurable gate array - perfect emulation," IEEE Journal of Quantum Electronics, Vol. 44, Issue 5, pp. 493-500, May, 2008.

[16] Wei Miao, Qingyu Lin, Wancheng Zhang, and Nan-Jian Wu, "A Programmable SIMD Vision Chip for Real-Time Vision Applications," IEEE JOURNAL OF SOLID-STATE CIRCUITS, pp. 1470-1479, vol. 43, no. 6, 2008.

[17] J. Mumbru, G. Zhou, X. An, W. Liu, G. Panotopoulos, F. Mok, and D. Psaltis, "Optical memory for computing and information processing," SPIE on Algorithms, Devices, and Systems for Optical Information Processing III, Vol. 3804, pp. 14–24, 1999.

[18] J. Mumbru, G. Panotopoulos, D. Psaltis, X. An, F. Mok, S. Ay, S. Barna, E. Fossum, "Optically Programmable Gate Array," SPIE of Optics in Computing 2000, Vol. 4089, pp. 763–771, 2000.

[19] J. Mumbru, G. Zhou, S. Ay, X. An, G. Panotopoulos, F. Mok, and D. Psaltis, "Optically Reconfigurable Processors," SPIE Critical Review 1999 Euro-American Workshop on Optoelectronic Information Processing, Vol. 74, pp. 265-288, 1999.

[20] M. Watanabe, F. Kobayashi, "An Optically Differential Reconfigurable Gate Array using a 0.18 um CMOS process," IEEE International SOC Conference, pp. 281–284, 2004.

[21] M. Watanabe, F. Kobayashi, "A high-density optically reconfigurable gate array using dynamic method," International conference on Field-Programmable Logic and its Applications, pp. 261–269, 2004.

[22] M. Watanabe, F. Kobayashi, "A 51,272-gate-count Dynamic Optically Reconfigurable Gate Array in a standard 0.35um CMOS Technology," International Conference on Solid State Devices and Materials, pp. 336-337, 2005.

[23] D. Seto, M. Watanabe, "Reconfiguration performance analysis of a dynamic optically reconfigurable gate array architecture," IEEE International Conference on Field Programmable Technology, pp. 265-268, 2007.

A Scalable FPGA Vehicle Monitoring and Classification Architecture

Francis Bowen, Jaehwan John Lee, and Eliza Yingzi Du
ECE Department, Purdue School of Engineering and Technology
Indiana University-Purdue University Indianapolis, Indiana, USA

***Abstract*— In this paper, we propose an FPGA-based vehicle monitoring and classification architecture. The proposed architecture is scalable and allows multi-lane concurrent processing. Furthermore, the system presented utilizes much less FPGA area while achieving real-time and high recognition accuracy. By utilizing an adaptive background subtraction method, we have produced accurate segmentation for varying conditions where other methods may fall short. Our experimental results show that an accuracy of 93% is feasible for most applications while each lane processing unit occupies only 13% of a Virtex-4 FPGA's slices. Furthermore, a simple post processing architecture is proposed for further improving the accuracy of the segmentation unit. Because of its small footprint, the system is suitable for portable applications such as a distributed traffic monitoring system.**

***Index Terms*—Field Programmable Gate Array, Image Segmentation, Vehicle Tracking and Classification**

I. Introduction and Background

There exist several popular image processing techniques for monitoring traffic. The first step is to segment the object from the background. The background subtraction technique is most popular, which involves a background subtraction from every frame followed by comparing the result with some threshold to determine if the pixel represents a background or foreground object [1]-[3]. The accuracy of the background estimation would be very important for the entire system performance. In [3], it is proposed to average a finite number of frames to produce a background estimate which is later combined with the previous background according to the following equation. The α-factor is defined as the background combination rate and can be adjusted to improve accuracy while x and y represent the pixel location, and z signifies one of three color dimensions in the RGB color space.

The combination of the current background with the most recently calculated background is required to offset any problems due to changes in illumination. One such scenario exists with moving clouds that can change the illumination of the detected vehicles.

Another common approach used in [4],[5] utilizes the difference between two or more consecutive frames to detect and classify moving vehicles. This method becomes erroneous when traffic has stopped moving altogether, such as with congested traffic or the delay caused by an accident.

Statistical models provide another interesting solution to background estimation. Stauffer suggests modeling every pixel with a mixture of K Gaussian distributions [6]. For a finite number of pixels over time, the pixel process can be expressed as

$$X = \{x_i = I_i(x, y)\}_{i=t}^{t+n}.$$

The corresponding posterior probability that a pixel belongs to the i-th Gaussian mixture (G_i) is given by [7] as

$$P\left(X \in G_i \middle| X = x_j, \theta\right) = \frac{w_i \psi(x_j|\theta_i)}{f(x_j|\theta)}$$

where w_i represents the relative weight of the G_i component and $\psi(x_j|\theta_i)$ is the PDF for G_i, which is defined as

$$\psi\left(x_j|\theta_i\right) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-\frac{1}{2}\left(\frac{x-\mu_i}{\sigma_i}\right)^2}.$$

The θ_i vector is comprised of the PDF parameters of G_i, which are the standard deviation (σ_i) and mean (μ_i). Lastly, $f(x_j|\theta)$ is defined as the mixture of all K Gaussians, i.e.,

$$f\left(x_j|\theta\right) = \sum_{i=1}^{K} w_i \psi(x_j|\theta_i).$$

As new frames are processed, each new pixel of the frame is compared with the set of K Gaussian distributions for that particular pixel location and ultimately determined to be either a foreground or background pixel. The pixel is determined to be a background component if it is within 2.5σ of the mean of G_i and $G_i \epsilon B$, where B is the set of computed background distributions.

In [8], Tan states that Gaussian Mixture Models (GMM) with a fixed number of components have shown to be inaccurate for large scale segmentation. To mitigate this drawback, Tan proposes using an adaptive GMM algorithm where the number of Gaussians can vary from 1 to L, with L representing the upper limit defined by assigned weights of the GMM. If a new pixel does not belong to a known model, a new set of Gaussian components is calculated for that particular pixel location. The most obvious constraint of this algorithm is the unknown amount of memory required to track the varying number of models.

In either situation of fixed or adaptive GMMs, prior knowledge of initial background components is necessary. The Expectation Maximization (EM) algorithm is a common method to learn these parameters [6]-[8], but is too complex to execute in real-time. Furthermore, once the initial parameters are calculated, the initial distributions need to be analyzed to determine which models represent a background distribution. This is also completed offline and adds to the complexity of the GMM solution. Due to the initialization process and overall complexity of GMMs, such a system would not be appropriate for traffic monitoring when the processing nodes are local to the sensors.

A modified GMM FPGA architecture is proposed by Appiah in [12], where a standard GMM approach is combined with a temporal low pass filter for background generation. This architecture requires four banks of off-chip memory which may be a limiting factor unless a custom hardware platform is developed. Furthermore, even with the added complexity of their algorithm, the reported accuracy for correctly segmented objects is less than 85%.

In [14], Jiang describes another GMM FPGA architecture that utilizes a compression technique for processing and storing the Gaussian parameters. Although Jiang reduces the memory bandwidth constraint, the overall complexity of the algorithm and resulting architecture occupy a large amount of the FPGA's resources.

The focus of the FPGA architecture outlined in [13], by Gorgon, lies solely on efficient background generation and motion detection. Gorgon proposes a simple sum of absolute differences (SAD) algorithm to determine if the difference between consecutive frames represents a vehicle in motion. In conjunction with SAD, a weighted average technique is identified for background calculation. The SAD motion detection requires $\lceil \log_2 M * N * 255 \rceil$ $bits/pixel$, where M*N is the resolution of each frame. This memory requirement is in addition to the space needed for background calculation and greatly limits the design.

Similarly, in [15], Juvonen describes a background estimation algorithm that involves creating a histogram for each pixel in a frame. Aside from the hefty memory requisite, the computed background does not change efficiently when the actual background changes such as with a moving camera or new background objects.

There have been several FPGA-based classifiers proposed in literature. Papadonikolakis defines a scalable Support Vector Machine (SVM) architecture for FPGAs. Although accurate, SVM's generally suffer from large dimensionality constraints, require supervised training and involve computationally expensive arithmetic [16]. In [17], Shi proposes a classifier based on GMM, however, the approach relies on time-consuming exponential calculations and large memory requirements.

Considering the complexity of GMMs, the inaccuracies of the difference-of-frames method, and the favorable results reported in [3], we developed a simple, yet effective, background estimation and subtraction architecture for vehicle counting and classification that uses less memory than other similar FPGA architectures. Additionally, none of the aforementioned FPGA implementations address noise identification and removal. The simple linear classifier described in [3] was implemented to minimize logic and maximize speed. The algorithm which is the basis for the hardware implementation will be briefly introduced in the following section.

II. Algorithm Overview

Our algorithm presented in this section contains two distinct phases. The process of image segmentation is presented first and followed by a discussion on the type of classifier implemented.

A. Segmentation

The initial step prior to segmentation is the color space conversion from the luminance and chrominance (4:2:2 YUV) to the RGB color space. By producing three correlated color dimensions per pixel, greater flexibility is given to the segmentation algorithm. Conversion is completed by applying the following transformation matrix:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.344 & -0.714 \\ 1 & 1.772 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U-128 \\ V-128 \end{bmatrix}.$$

Segmentation, the process of identifying a pixel as either a background or a foreground component, is achieved by a per pixel subtraction of a background estimate from the current frame's pixel. The result is a binary image where ones indicate foreground pixels and zeros represent background pixels. If we define the binary image for the i-th color component as BI_i, the value stored in the binary image buffer (*BI*) for the pixel located at (x,y) is the logical-OR of all $BI_i(x,y)$ buffers. This relationship is summarized as follows, where $I_i(x,y)$ is the pixel and $B_i(x,y)$ is background information for that particular location:

$$BI_i(x,y) = \begin{cases} 1, & |I_i(x,y) - B_i(x,y)| \geq T \\ 0, & otherwise \end{cases}$$

where $i = \{red, green, blue\}$ and

$$BI(x,y) = BI_{red}(x,y) \,|\, BI_{green}(x,y) \,|\, BI_{blue}(x,y).$$

For m consecutive frames, a particular background is used by the segmentation module; however, this background must be updated regularly to include luminosity changes in the field of view. If the background is updated too frequently, errors will be introduced into the background estimation when traffic is slowly moving or completely stopped. However, if the estimation is not updated within a reasonable timeframe, sudden changes in illumination will cause segmentation errors.

Our initial parameter for m is set to 32, which means the background is updated approximately once every second. This parameter does not only affect the segmentation operation, but also dictate the size of the accumulators needed to store the latest background. The size for each memory location must be at least $\lceil \log_2(765 * n) \rceil$ bits. For a resolution of 640x480 (standard VGA), and a background calculation frequency of 32 frames, the total space required for the accumulator will be 562.5 KB. For this specific application, setting the background update frequency to 32 frames provides a good tradeoff between accuracy and resource utilization.

The binary image resulted from the segmentation module requires post processing to remove noise and merge regions that were not correctly identified by segmentation. Figure 1 depicts a video frame, the estimated background and corresponding segmented image while Figure 2 provides an example with both disjoint regions and noise.

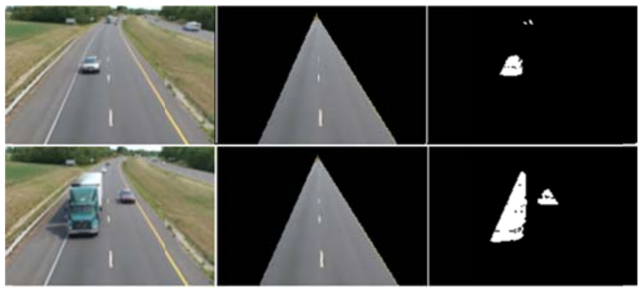


Figure 1. Video frame, estimated background and resulting vehicle segmentation.

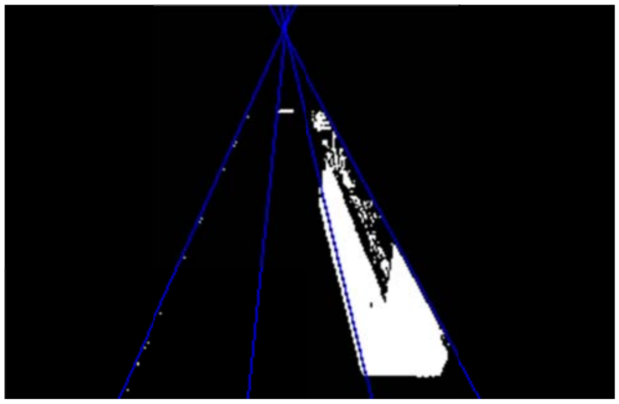


Figure 2. Example segmentation result illustrating noise from the imprecisions of segmentation as well as disjoint foreground regions that should belong to the same object.

Pixels are streamed in raster order and subsequently produce a stream of binary values which drive the state machine shown in Figure 3 that we use. On a per line basis, noise is identified by monitoring the pattern of detected background and foreground pixels. At the beginning of a scan line, the state machine is initialized in the idle state. If the incoming pixel is a foreground component, pixel is tagged as noise. Once however six consecutive pixels are found grouped on the same scan line, the group is assumed to be foreground pixels and does not reflect any noise.

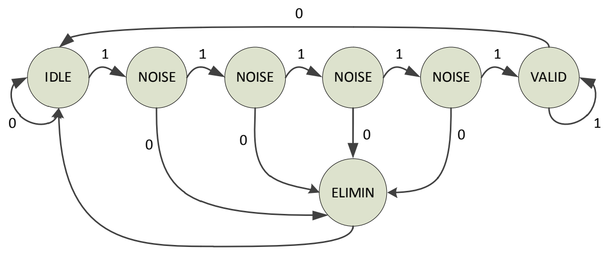


Figure 3. Binary image noise detection state machine.

Disjoint regions are another source of error from segmentation. Often an entire object is not segmented but rather sections are identified and must be merged together. To accomplish this task, a simple proximity algorithm is adopted in which, for example, by inspecting the previous two scan lines, two regions are merged if they are within two pixels of a neighboring region.

The only requirement to efficiently execute the region merging is the need to have the three most recent binary image lines buffered in local registers. This will require three 640-bit registers which are within close vicinity of the segmentation module.

B. Classification

After segmentation, vehicles are classified based on their size. A complete description of each of the four classes can be found in [3] and Table 1 summarizes the types of vehicles per class.

Table 1: Four classes of vehicles

Class	Type of Vehicle
1	All passenger cars (motorcycles, sedans, trucks, SUVs)
2	Large trucks (moving vans, industrial vehicles)
3	Regular semi-truck
4	Double semi-truck

The proposed classifier requires knowledge of both endpoints of a vehicle, and therefore, the trajectory of either the front or back of the vehicle must be tracked. Figure 4 illustrates this process which requires flags to be set from the noise detection and region merging modules. Once a scan line has been found to be occupied with foreground pixels, the noise detection unit signals the object tracking unit which uses the information to count the vehicle or calculate the size of the vehicle in terms of pixels.

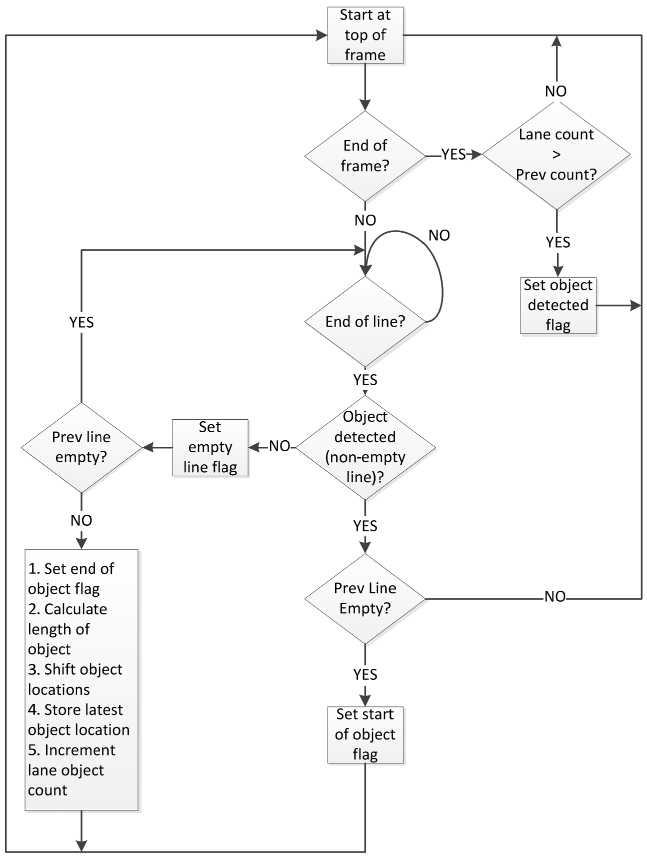


Figure 4. Vehicle tracking and counting algorithm.

The pixel length of detected objects is passed to a linear classifier that sets flags based on predetermined classification values that are determined experimentally. A classifier's success is dependent on the quality of the segmentation output; therefore, the focus of this research has been the segmentation unit and supporting modules.

III. Hardware Architecture

The aim for the hardware architecture was to minimize the logic and to exploit parallelism in the original software algorithm with the goal of processing video in real-time (30 fps). Scalability is realized by implementing independent modules for each lane of a given highway. Figure 5 provides an overview of a single lane processing unit.

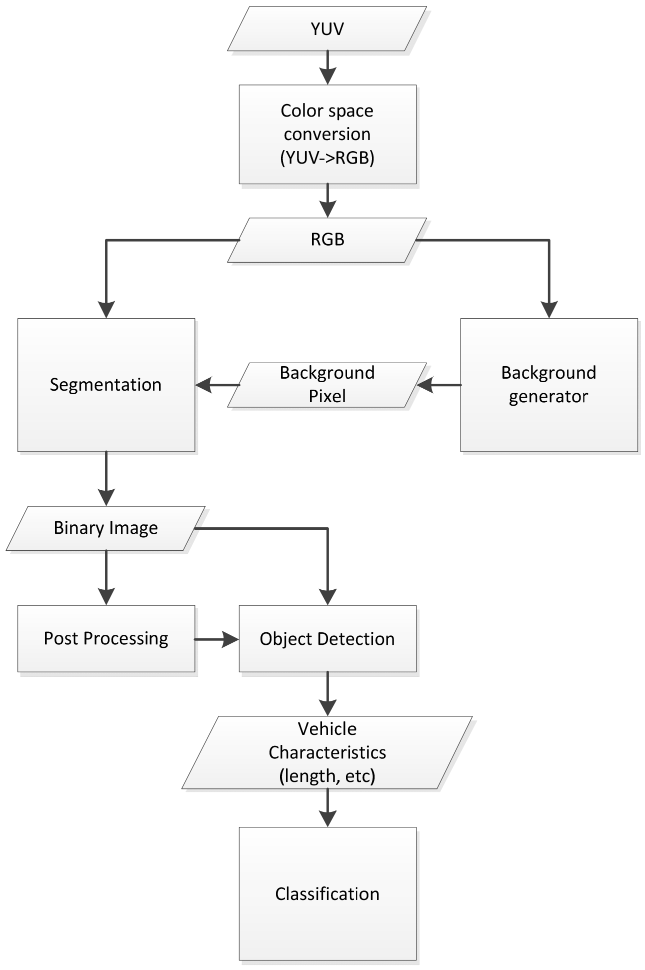


Figure 5. Lane processing unit.

The Xilinx YCrCb-to-RGB Color-Space Converter v1.0 [26] was employed for the conversion from luminance and chrominance to RGB values. This module consists of a seven stage pipeline for applying the transformation matrix. After an initial delay of seven pixel clock cycles, a new set of RGB values is ready on every rising edge of the pixel clock.

A. Clock Distribution

For a color VGA stream with a resolution of 640x480, a pixel clock of 27MHz is required $(640x480\ samples * 3\ bytes/sample * 30\ frames/sec \approx 27\ MB/sec)$. The pixel clock drives the video decoder and the color space conversion unit; however, a faster clock is required for the external memory, segmentation unit, and classifier. Xilinx provides Digital Clock Managers (DCM) for buffering, multiplying, or dividing clock sources [13]. This resource was exploited to generate each two sets of 54 MHz, 108 MHz, and 216 MHz internal clocks from the original pixel clock.

One 108 MHz clock is used to drive the segmentation module and background generator while the other controls the sequential logic of the post processor, object tracker, and finally the classifier. The choice to generate multiple clocks was motivated by the desire to avoid clock skew.

The external memory presented a challenge in the form of read and write latencies; consequently two 216 MHz clocks were generated for each memory bank utilized. This accelerated clock rate allowed for three reads and one write operations between incoming pixels, which are required by the background generator.

B. Memory Hierarchy

To adequately handle the large amount of data required for pixel information, accumulators for each pixel, binary image, registers required by KNN, lane boundary information, and a separate background frame buffer for the current background used in segmentation, all available memory resources were exploited. Figure 6 outlines the type of resource and its assignment within the architecture.

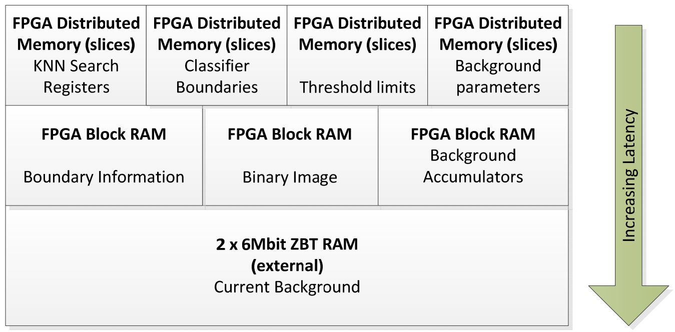

FPGA Distributed Memory (slices) KNN Search Registers
FPGA Distributed Memory (slices) Classifier Boundaries
FPGA Distributed Memory (slices) Threshold limits
FPGA Distributed Memory (slices) Background parameters
FPGA Block RAM Boundary Information
FPGA Block RAM Binary Image
FPGA Block RAM Background Accumulators
2 x 6Mbit ZBT RAM (external) Current Background
Increasing Latency

Figure 6. Memory hierarchy

The accumulators required for the background estimation accesses memory for every processed pixel, and this imposes the strictest constraint upon the architecture. For the given resolution and a background update frequency rate of 32 frames per cycle, each pixel location requires 45 bits (three 15-bit accumulators for each color space components) of memory, which amounts to a total of 562.5KB. The obvious choice is to store and fetch the accumulators from the external ZBT memory; however, the width of the data bus is 16 bits and would require three read and another three write operations to update all accumulators for a particular pixel location. This is not feasible, even with a 216 MHz memory clock; therefore, the accumulators are stored on-chip within the Block RAM of the FPGA.

The current background pixels used in the segmentation module can be stored in an interleaved fashion to minimize memory accesses. With interleaved storage, four pixels can be read with three read operations.

C. Background Generator

Figure 7 describes the background estimation datapath. It performs summing of the incoming RGB pixel with its corresponding BRAM accumulator value. Since a lane is defined by two distinct boundaries, if the current sample lies outside of the defined boundaries, a zero is propagated through the datapath. A counter is used to determine if the current background should be updated. As previously mentioned, an update frequency of 32 was chosen according to the discussion in Section III. However, another criteria was considered. If the update frequency is chosen such that it is a power of two, the averaging operation is reduced to a logical shift of a register. By updating the background every 32 frames and choosing the combination factor α as 0.25, the new background is computed using three shift registers and two adders. For similar reasons, α is chosen such that the combination of the old background and the accumulator is accomplished with simple shift registers and adders.

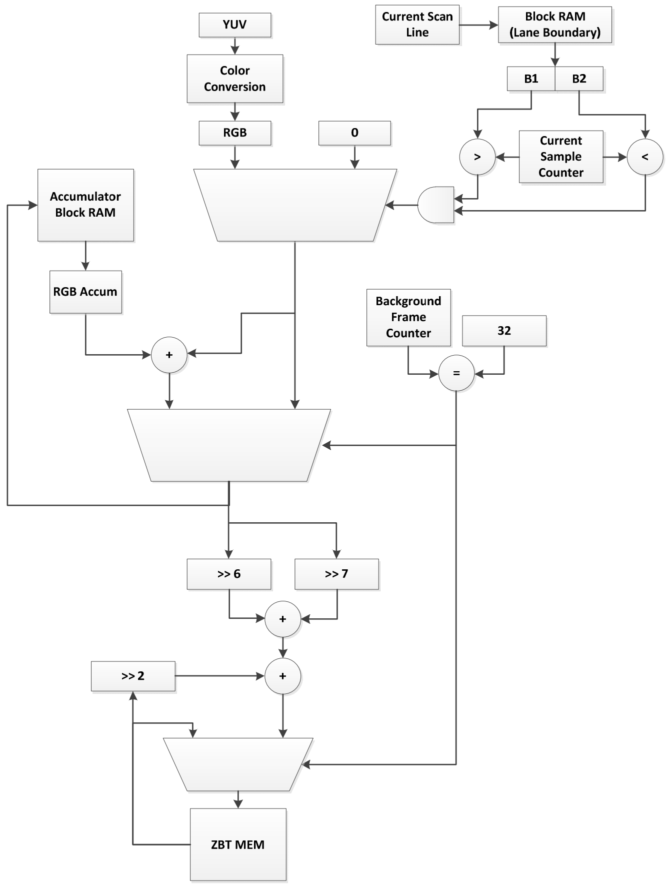


Figure 7. Background estimation datapath.

D. Segmentation

Figure 8 provides a partial illustration of the segmentation datapath. The logic shown is replicated three times and combined with a logical-OR to calculate the segmentation result that is stored in a local register for analysis by the noise detection module. At the end of a scan line, the temporary register is transferred to a new register that represents the previous scan line. Preceding the transfer, the temporary register holding the line prior to the previous line is stored in the local BRAM buffer, and the previous line is transferred into the third register. The *Thresh* register contains the defined threshold limit for a particular color dimension.

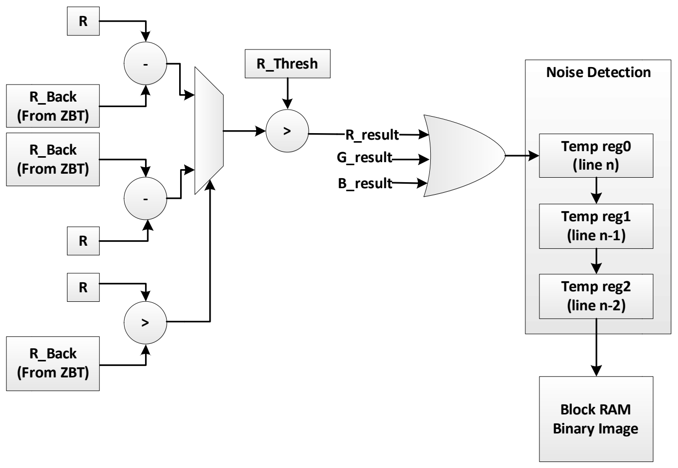


Figure 8. Partial segmentation datapath. R is the current pixel's red component; R_Back is data from the background generator; and R_Thresh contains the threshold limit.

E. Classifier

The aforementioned linear classifier has been implemented using minimal combinational logic. The necessary parameters such as the class and lane boundaries are communicated during system initialization.

The structure consists of four stages (for four classes) where a flag is set if a vehicle is determined to belong to that particular class. This design is similar to a ripple carry adder in that the output of one stage cannot be considered until some delay after the output of the previous stage is stable. Although the propagation delay of each stage is relatively small, the overall delay from the first to last stage may be an issue, especially if this type of design is used in a pipeline.

F. Scalability

The segmentation and classifier units are used concurrently for processing a single lane which is defined by a set of user-provided boundary locations. For processing of additional lanes, a segmentation and classifier pair must be allocated, however, additional memory is not required beyond what is used by segmentation and classification. Regardless of the number of lanes to process, the background number of background accumulators and amount of internal BRAM remains constant. As new frames are streamed in raster order, the processing units for each lane will either accept the incoming pixel, or ignore the data if its location lies outside of that particular unit's lane boundaries.

IV. Results and Discussion

The entire system was implemented and tested using the Sundance SMT339 [14], which is comprised of a Xilinx Virtex-4 FPGA, dual ZBT memory banks, a Philips SAA7109AE/108AE encoder/decoder IC, and a DSP that is capable of communication with a host PC through the host's PCI interface for validation purposes. For data collection, a simple user interface was created using Sundance-provided APIs to read the binary image produced by the segmentation module.

Since the data is provided in raster order and the architecture is designed to process the pixels as they become available, the overall system is shown to be capable of real-time processing. There does not exist any scenario where the processing will delay the system result.

The entire design utilizes approximately 40% of the available slices on a Virtex4 FX60 FPGA, 93% of the on-chip BRAM, and less than 10% of the off-chip ZBT SRAM. Regardless of the off-chip ZBT SRAM usage, memory operations are continually being issued by the background generator; consequently, the other 90% of the memory locations will remain unavailable to other designs. Approximately 13% of the FPGA's slices are required per lane for segmentation and classification.

It is worth noting that the memory requirements do not scale relative to the number of lanes. Regardless of the number of lane modules, the background accumulator in BRAM and the current background stored in external memory will still occupy the same amount of space which is determined by the resolution of the incoming frames. To ease the memory requirements, it is suggested that the incoming frames are down-sampled. For each iteration of down-sampling, the external and internal memory requirements can be reduced by a factor of four times. Down-sampling too severely will cause aliasing; however, it is understood that down-sampling done once does not drastically affect the overall results. If the video source is analog, one can crudely down-sample the video frame by only processing even or odd interlaced frames while ignoring every other pixel from the decoder. This approach does not require any additional buffers and will not have any other effect on the rest of the proposed architecture.

Figure 9 shows the results of segmentation with noise removal and region merging algorithm. The lane boundaries are superimposed to provide perspective. As shown, the simple method for segmentation, coupled with post processing to remove noise and detect regions to merge, is effective for identifying moving vehicles.

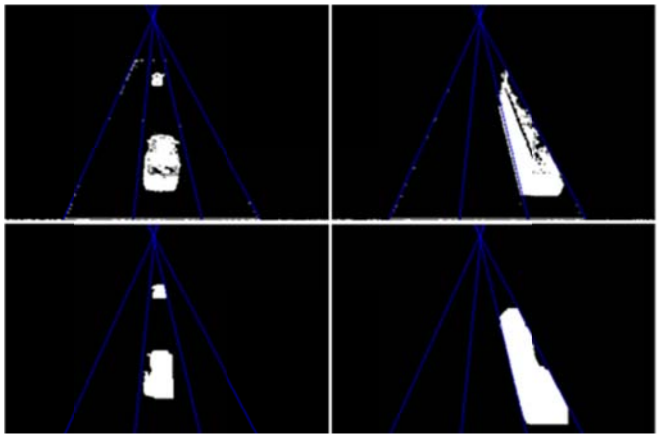


Figure 9a-b. Noise removal and region merging for a passenger car and regular semi-truck, respectively.

The method presented does not always correctly segment a moving vehicle, and the most common error were two regions that belong to the same object but are counted as individual vehicles. Such false positive errors have been detected to occur in approximately 3% of a total of 247 vehicles tested. To overcome this issue, it is suggested that the threshold values for each color component be updated adaptively from the results of the noise detection circuit. A simple approach to this problem may use the total amount of noise detected in a frame to increase or decrease the threshold limit. Another algorithm to minimize the detected error may involve varying the frequency in which the background is updated. Such a solution may require updating the background more aggressively if no vehicles are detected and less frequently when traffic is determined to be heavy. The total number of undetected vehicles is less than 1% of the tested data; thus, further efforts should concentrate on the double counting issue.

Lastly, the success rate of vehicle counts was found to be 97% for class 1, 86% for class 2, 93% for class 3, and 50% for class 4. The high error rate of class 2 and 4 is due to two factors. First, both classes are considered rare, and thus, the population size of all the samples is quite low, and small amounts of error quickly increase the overall error for that class. Second factor is especially true for class 4 vehicles; if the vehicle is large, there is more of a chance to double count the vehicle as previously mentioned. This occurred in one scenario where only two class 4 vehicles were observed; one was not segmented into a single object, was double counted, and subsequently we detected a 50% error rate for that particular class.

V. Conclusion

In this paper, we have proposed a simple yet effective scalable image processing architecture for the task of vehicle counting and classification. The architecture has been outlined and underlying algorithm explained with results which demonstrate the effectiveness of the system. Being energy efficient and small, such a system is best suited for local processing in a distributed camera network.

Continued work on this project could focus on developing an adaptive threshold limit algorithm and an adaptive background generator such that errors due to poor segmentation can be minimized. The proposed architecture can be altered for general object tracking and classification applications.

VI. Acknowledgment

This research is partly supported by NSF grant 0834682. Both software tools and FPGAs (Virtex-4 FX60) were graciously donated by Xilinx.

References

[1] C. Chung-Cheng, K. Min-Yu, and L. Li-Wey, "A Robust Object Segmentation System Using a Probability-Based Background Extraction Algorithm," Circuits and Systems for Video Technology, IEEE Transactions on , vol.20, no.4, pp.518-528, April 2010.

[2] D. Culibrk, O. Marques, D. Socek, H. Kalva, and B. Furht, "Neural Network Approach to Background Modeling for Video Object Segmentation," Neural Networks, IEEE Transactions on , vol.18, no.6, pp.1614-1627, Nov. 2007.

[3] F. Bowen, Y. Du, S. Li, Y. Jiang, T. Nantung, S. Noureldin, M. Knieser, and M. Rizkalla, "Dynamic Content Based Vehicle Tracking and Traffic Monitoring System," SPIE Electronic Imaging, Vol. 6497, 64970I-1~11, 2007.

[4] C. Jiang-Zhong and D. Qing-Yun, "A novel online fingerprint segmentation method based on frame-difference," Image Analysis and Signal Processing, 2009. IASP 2009. International Conference on, vol., no., pp.57-60, 11-12 April 2009.

[5] W. Shigang, W. Xuejun, and C. Hexin, "Video object segmentation based on frame differences and its implementation on DSP," Visual Information Engineering, 2008. VIE 2008. 5th International Conference on , vol., no., pp.618-621, July 29 2008-Aug. 1 2008.

[6] C. Stauffer and W.E.L. Grimson, "Adaptive background mixture models for real-time tracking," IEEE Conference on Computer Vision & Pattern Recognition. Colorado, USA. June 1999. IEEE. Pages 246 – 252.

[7] T. Bouwmans, F. El Baf, and B. Vachon, "Background Modeling using Mixture of Gaussians for Foreground Detection - A Survey", Recent Patents on Computer Science, Volume 1, No 3, pages 219-237, November 2008.

[8] R. Tan, H. Huo, J. Qian, and T. Fang, "Traffic Video Segmentation Using Adaptive-K Gaussian Mixture Model." IWICPAS 2006: 125-134.

[9] Xilinx, "YCrCb to RGB Color-Space Converter v1.0," DS659, March 24, 2008.

[10] Xilinx, "Virtex-4 FPGA User Guide," UG070 (v2.6), December 1, 2008.

[11] Sundance Multiprocessor Technology Ltd., "SMT339 User Manual V1.3," August 11, 2000.

[12] K. Appiah and A. Hunter, "A single-chip FPGA implementation of real-time adaptive background model," Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on , vol., no., pp.95-102, 11-14 Dec. 2005.

[13] M. Gorgon, P. Pawlik, M. Jabtonski, and J. Przybyto, "FPGA-based Road Traffic Videodetector," Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on, vol., no., pp.412-419, 29-31 Aug. 2007.

[14] H. Jiang, H. Ardo, and V. Owall, "Hardware accelerator design for video segmentation with multi-modal background modelling," Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on , vol., no., pp. 1142- 1145 Vol. 2, 23-26 May 2005.

[15] M. Juvonen, J. Coutinho, and W. Luk, "Hardware Architectures for Adaptive Background Modelling," Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on , vol., no., pp.149-154, 28-26 Feb. 2007.

[16] M. Papadonikolakis and C. Bouganis, "A novel FPGA-based SVM classifier," Field-Programmable Technology (FPT), 2010 International Conference on , vol., no., pp.283-286, 8-10 Dec. 2010.

[17] M. Shi, A. Bermak, S. Chandrasekaran, and A. Amira, "An Efficient FPGA Implementation of Gaussian Mixture Models-Based Classifier Using Distributed Arithmetic," Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on , vol., no., pp.1276-1279, 10-13 Dec. 2006.

SESSION

SHORT PAPERS

Chair(s)

TBA

The Tools Have Arrived: Two-Command Compilation and Execution of Scalable, Multi-FPGA Applications

Brian Holland
SRC Computers, LLC
4240 N. Nevada Ave
Colorado Springs, CO 80907
www.srccomputers.com
bholland@srccomputers.com

***Abstract*—Continual increases in the computational resources of Field-Programmable Gate Arrays (FPGAs) provide greater capabilities for algorithm migration to hardware to capitalize on higher performance at lower power. However, wider adoption of FPGA technologies is often assumed to be limited by development costs associated with lengthy toolflows dissimilar to traditional development of software applications. Despite this perception, an efficient toolflow for FPGA development is currently available and has demonstrated success in cost-effective migration of applications to hardware. The toolflow allows straightforward compilation and execution of applications through GNU make and a standard UNIX runtime environment. This paper describes general requirements and challenges of productive toolflows for FPGA systems. It also presents the SRC[1] MAP processor, Carte development environment, and associated runtime environment for efficient application execution. Lastly, the paper demonstrates the efficiency of the SRC toolflow with a string-matching application.**

I. Introduction

FPGAs are continually evolving beyond the traditional role of replacing application-specific integrated circuits (ASICs). For example, systems from SRC Computers [6] have expanded the usage of FPGAs from glue-logic replacement to peer processors based on the customer's needs for faster and more customized data processing. FPGAs provide increasing capabilities for scientific computation including more fixed resources (e.g., multiply accumulators) and soft macros (e.g., IEEE floating-point operations). They also have grown both in I/O capacity and ability to rapidly integrate new interconnect standards.

FPGAs typically use more discrete tools than other parallel-computing technologies, which has been traditionally considered a key challenge for greater deployment of FPGA-based computing systems. In contrast, multi-core processing uses applications code based on threads (e.g., pthreads), shared memory models (e.g., SHMEM), or message passaging libraries (e.g., MPI). Sequential codes are modified using first or third-party libraries plus an associated runtime environment for relatively straightforward application migration. Similarly, graphics and other vector-processing technologies involve adaptation of sequential codes with technology-specific instructions (e.g., CUDA or OpenCL). However, FPGA systems traditionally require conversion of legacy microprocessor applications to parallelized (e.g., pipelined) circuits using a hardware description language (HDL), integration with a third-party system-level or board-level interfaces, synthesis, and finally place and route using FPGA vendor tools. This manual application development process implies three distinct steps and at least two toolsets from different sources plus further complexity depending on the runtime environment for the FPGA system.

Despite these widespread perceptions, efficient application development and migration for FPGA-based systems exists through effective integration of tools and encapsulation of tedious elements outside the scope of a typical application developer. High-level compilers can greatly assist in conversion of traditional software code into hardware-oriented circuits. These compilers and other third-party tools can also assist in the connection of the application kernel(s) to the requisite system interface. Assuming correct functionality of the kernels and system interfacing, the synthesis and place-and-route (PAR) toolflow can be scripted and therefore largely invisible to the user except for their nontrivial duration. SRC Computers provides the only standard language toolflow, the Carte development environment, that tightly integrates all three levels of tools and includes an efficient runtime management system.

[1]SRC, MAP, Carte, and CDA are trademarks or registered trademarks of SRC Computers, LLC. All rights reserved. All other trademarks are the property of their respective owners.

The desire for efficient application development and system usage is a key research emphasis for FPGA computing. Some efforts have focused on identifying strategic challenges [5] and classifying traditional limitations [4] of these systems. Panels such as one at the international conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)[3] described productivity as a major requirement for application migration. However, researchers and panel participants often lack exposure to the capabilities of the available toolflows. Insufficient attention is given to summarizing application developer needs, identifying toolflows such as the Carte toolflow that efficiently and effectively meet these needs, and quantitatively exploring their capabilities.

The rest of this paper is structured as follows. Section II further details key features and challenges for efficient application kernel migration, system integration, FPGA synthesis and PAR encapsulation, and runtime management. Section III describes the SRC integrated and unique approach to compilation, system integration, encapsulation of synthesis and PAR, and runtime management. Section IV presents a walkthrough of efficient application migration using SRC hardware and the Carte toolflow with a string-matching application. Conclusions are given in Section V.

II. Application Toolflows: Features and Challenges

The complexity of migrating application kernels to hardware circuits, integrating these kernels with FPGA-based systems, encapsulating the low-level synthesis, place and route details, and using the associated runtime environment are key challenges for efficient usage of such systems. This section describes the requisite features of these four aspects of application development and how increasing their integration can overcome the perceived limitations of FPGA-based computing.

A. Application Kernel Migration

Developing or migrating applications to FPGAs is often related to the classical challenges of algorithm parallelization. Although applications may benefit exclusively from lower power consumption on FPGA systems as compared to microprocessor-based platforms, the performance of many applications increases from the spatial and temporal parallelism inherent in the FPGA's architecture. Approaches to FPGA parallelization can benefit from microprocessor strategies such as loop unrolling and pipelining, but can also suffer from similar challenges of lengthy iterations of implementation, evaluation, and revision. Particularly for FPGA systems, complex structures need to be implemented and subsequently modified simply and quickly. However, these parallel structures must be granular enough to allow for widespread reuse, capitalizing on a library-type approach.

Reusable structures maintain their simple and quick characteristics if they minimize deviation from the original application source code, description, or programming model. For example, textual application specification with C syntax provides a paradigm familiar to programmers. While nontrivial to construct, a specialized compiler can convert this textual code into efficient FPGA designs if the underlying specification includes intuitive representations of parallelism and timing. The compiler would construct full application designs from pre-optimized components. However, even with efficient application specification, revision, and compilation, FPGA designs cannot be developed in isolation of system-level details.

Fig. 1. Cluster System

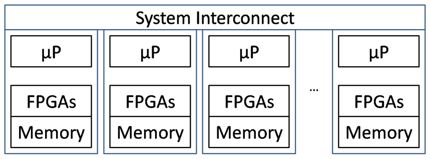
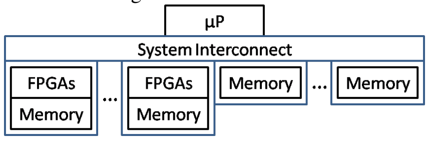


Fig. 2. Direct Connect



B. System Integration

Efficient application migration requires scoping the parallelization to the constraints of the system. The number and capacity of the FPGAs represents a key constraint, often requiring a different and potentially more complex parallelization strategy than a microprocessor-based system. Tradeoffs include application decomposition among multiple FPGAs and within the resources of a single FPGA (e.g., lower area, lower performance kernels or fewer higher area, higher performance kernels). A lack of integration between kernel-level and system-level tools can require manual selection of parallelization strategies, implementation, and ultimately costly reimplementation if initial choices prove infeasible.

Additionally, management of the system interconnect bandwidth can be a neglected aspect of application migration with disconnected toolsets. Beyond the classical problem of ensuring sufficient overall bandwidth for sustained computation, dividing the available bandwidth to maximize performance and prevent deadlock can become the burden of the application developer. The memory hierarchy further complicates the effective usage of the communication infrastructure. Without system infrastructure, applications must manually connect data transfers between possible combinations of microprocessor memory, FPGA on-chip memory, and the one or more levels of off-chip memory (e.g., SRAM or SDRAM) associated with an FPGA board or module.

C. Runtime Libraries

The runtime environment for an FPGA system also affects the efficiency of the aforementioned FPGA, interconnect, and memory resources. Ideally, an application should only occupy the FPGAs and memories necessary for the computation and this allocation should not prevent the usage of the remaining system resources by other applications. An application cannot determine a priori the resources currently assigned to other applications. Therefore, the application must either wait for its statically assumed resources to become available or be sufficiently adaptive to function with any available resources assigned to it by the runtime environment. The capabilities of the runtime system are related to the organization of the resources in the system architecture.

Systems defined by FPGA cards connected to the microprocessor peripheral bus have traditionally lacked a single unified manager of the global resources. Figure 1 illustrates a common architecture for an FPGA system that organizes a number of FPGAs and memories around a microprocessor and manages this resource node as an atomic unit. Node allocations are controlled by traditional cluster management software while low level drivers manage individual resources and interconnects. The microprocessor-oriented system is relatively easy to manage but wasteful due to coarse-grain resource allocation and has lower performance due to a communication bottleneck through the microprocessor. Additional bottlenecks are also present when coordinating multiple resources through a common peripheral interconnect such as PCI Express. An alternative organization, Figure 2, allows individually addressable FPGA and shared memory resources albeit with increased system and runtime complexity.

D. Encapsulation of Build Environments

Even with a sufficiently integrated methodology for application kernel migration and integration to an FPGA system, overreliance on developer input during synthesis and PAR can reduce the applicability of FPGA systems for broader use in scientific computing. Assuming the preceding toolflow generated a correct and optimized application code, direct usage of these tools is unnecessary and potentially confusing to traditional application developers. Interaction with synthesis and PAR can provide an experienced developer with fine-grain control over low-level application details, but explicit interaction with these FPGA-specific tools should be optional when streamlined application development is desired.

Additionally, disconnects in toolflows for application development are not limited strictly to the generation of final hardware implementations. Regardless of manual or tool-assisted migration, application functionality is first verified (typically as a software emulation), then implemented or translated to HDL, evaluated in a simulator, and finally built into a full hardware design. Without the ability to progress from software emulation to hardware simulation and final implementation using a single application representation, FPGA systems can become prohibitively expensive in terms of real world code development and maintenance.

III. SRC Computers' Carte Toolflow

The SRC Carte toolflow and unified SRC-7 system architecture [7] are specifically designed to address the aforementioned limitations of disconnected methodologies for application migration to SRC FPGA-based MAP processors. The Carte application development toolflow includes a Code Development Assistant (CDA[1]) and an architecture-aware compiler for rapid implementation using new or legacy C and Fortran. The compiler and subsequent synthesis and PAR tools are tightly integrated with software emulation and hardware simulation for an encapsulated, two-command environment (i.e., “make” and “run”) for application design. The Carte runtime environment manages both the access control to system resources and interconnection bandwidth to help maximize the average communication bandwidth of multiple applications simultaneously.

A. Application Kernel Migration

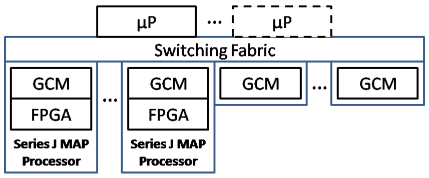
The Carte toolflow provides several mechanisms for constructing applications with extremely minimal exposure to the underlying hardware specifics. This toolflow was designed to minimize developer effort by focusing on application migration through developer-familiar standard ANSI C and Fortran languages for the MAP processor. (Microprocessor code can remain in other languages.) The CDA tool can automatically replace minimally annotated software segments with highly optimized MAP processor functions. Alternatively, legacy codes can incorporate special functions and semantic mechanisms, such as pipelining, loop fusion, and loop unrolling, to exploit inherent parallelism in the application kernel while maintaining C or Fortran syntax. These optimizations include simple yet powerful functions for effective data movement to and from system resources. The Carte methodology defines MAP processor resources as the primary initiators of data movement through efficient direct memory access (DMA) or streaming between global memory and local resources.

B. System Integration

The macro-based library approach of the Carte toolflow allows effective usage of key architecture features of the system while hiding low-level implementation details. Memory resources exist in a flat global-address space, regardless of locality, with the system interconnect handling appropriate communication routing. MAP processors operate independently, synchronizing only as necessary based on barrier messages handled by the system interconnect. Consequently, application codes can achieve comparable performance whether using different resources in a system or different system configurations.

An SRC-7 system (Figure 3) corresponds to the second system architecture described in Section II-C. The system interconnect is comprised of a tiered switching fabric that provides full connectivity between FPGA and memory resources (i.e., a nonblocking crossbar switch). The current MAP processor, the Series J, contains an Altera EP4SE530 FPGA for user applications. The memory resources, referred to as global common memories (GCMs), physically reside in pairs of banks on either a MAP processor or as an independent unit. However, each GCM bank is independently addressable and accessible by all MAP processors and microprocessors in the system. The Series J MAP processor physically combines the logically separate FPGA and

Fig. 3. SRC System Architecture



memory resources to allow a mix of resources within a small footprint. The size and configuration of the overall system can be customized depending on application requirements.

C. Runtime Libraries

Application codes are organized into microprocessor functions (primarily for application coordination and other ancillary operations such as file I/O) and MAP functions for FPGA-based scientific computing. The runtime environment allows the main microprocessor function to request memory and system resources, and initiate the MAP processor-based computation with those resources. The environment is multi-user and prevents illegal actions such as memory accesses outside a valid address range by both the microprocessor and the MAP processors associated with an application. The switch-based interconnect further promotes multi-user behavior by packetizing and interleaving data transfers to minimize the average communication delay through the system. Under normal operation, system loading from multiple users cannot prevent the eventual successful completion of an application.

D. Encapsulation of Build Environments

A key feature of the Carte toolflow is the seamless transition of application development through functional emulation in software, simulation of hardware, and full implementation. This flow allows for extremely rapid (i.e., seconds to compile and run) functional testing during code development with standard debugging methods such as gdb and printf(). Once a prospective code is complete, simulation is available for cycle-accurate performance evaluation (i.e., typically minutes to compile and run). Normally, simulation is unnecessary for application development but is available for more detailed algorithm analysis without lengthy synthesis and PAR, or occupying real system resources. Only codes with suitable functionality and performance need hardware compilation and physical system testing.

Progressing through these tools only requires "make debug", "make simulate", or "make hw" commands from the user followed by "exec myfile.[dbg/sim/hw]" for running the unified executable. A unified executable is created that contains all necessary emulation binaries, hardware simulation components, or FPGA configuration files for debug emulation, hardware simulation, or MAP processor execution, respectively. The underlying tools are completely encapsulated by the Carte toolflow with any specific tool-related options defined in a singular makefile. This allows not only simplistic application management through the toolflow but also high portability as the underlying tools and/or resources evolve.

IV. Application Development

This section presents an example walkthrough of application migration and optimization for the SRC-7 system using the Carte toolflow. The target algorithm is a string-matching kernel used in areas such as packet inspection, data mining, and computational biology [2]. String matching involves locating instances of one or more key strings (also referred to herein as keywords) within a larger sequence of text. Section IV-A outlines the specific implementation of the string-matching algorithm. Section IV-B illustrates how the algorithm can benefit from streaming data for sustained pipelined execution. Section IV-C discusses an additional algorithm optimization which can further increase performance by attempting to match multiple strings in parallel. Quantitative results are presented in Section IV-D.

A. String-Matching Algorithm

This implementation of the string-matching algorithm provides an exhaustive search of the larger text for the keyword. Although pre-computation of a substring index [1] for the keyword can increase the performance of microprocessor implementations, the advantages

Fig. 4. String Matching in Software

```
 1:  uint64_t data, result, keyword, mask;
 2:  char *mem = (char *)malloc(numchars);
 3:  char *res = (char *)malloc(numchars)
 4:  uint64_t data, result, keyword, mask;
 5:  keyword = 0x48454C4C4f         //ascii HELLO
 6:  mask    = 0x000000FFFFFFFFFF; //5 char window
 7:
 8:  for(i=0;i<numchars;i++)
 9:  {
10:      data   = (data<<8) | mem[i];
11:      res[i] = (data & mask) == keyword;
12:  }
```

Fig. 5. String Matching in Carte

```
 1:  Stream_64 S64_in, S64_out;
 2:  Stream_8  S8_in,  S8_out;
 3:
 4:  #pragma src parallel sections
 5:  {
 6:    #pragma src section
 7:    {
 8:      //GLOBAL MEMORY TO FPGA INPUT STREAM
 9:      streamed_dma_gcm(&S64_in, ...);
10:    }
11:
12:    //INPUT STEAM FORMATTING
13:    //QWORDS (64 bit) TO CHARS (8 bit)
14:    ...
15:
16:    #pragma src section
17:    {
18:      for (i=0;i<numchars;i++)
19:      {
20:        get_stream(&S8_in, &mem);
21:        data = (data<<8) | mem;
22:        res  = (data&mask) == keyword
23:        put_stream(&S8_out, res, 1);
24:      }
25:    }
26:
27:    //OUTPUT STREAM FORMATTING
28:    //CHARS (8 bit) TO QWORDS (64 bit)
29:    ...
30:
31:    #pragma src section
32:    {
33:      //FPGA OUTPUT STREAM TO GLOBAL MEMORY
34:      streamed_dma_gcm(&S64_out, ...);
35:    }
36:  }
```

of indexing can be degraded on FPGA systems depending on memory latency. The software implementation presented in Figure 4 provides straightforward execution on microprocessors and FPGAs. The keyword for this example is "HELLO" (Line 5) and a mask (Line 6) isolates a five-character (i.e., the length of "HELLO" excluding the null terminator) window for comparison. For each character in the larger sequence of text, that character is shifted into the window of data (Line 10) and subsequently compared against the "HELLO" keyword (Line 11). The mask ensures that only the relevant characters are compared as the 64-bit datatypes can store upto eight characters. The core algorithm on Lines 10 and 11 involves only simple binary operations, which are highly amenable to FPGAs. The major algorithmic change involves migrating the input and resulting arrays (i.e., mem[] and res[], respectively) to the memory structure and dataflow of the SRC system.

B. Application Migration

As described in Section III-A, streaming provides a mechanism for the resources on a MAP processor to access the system's global memory. Streaming provides a low overhead mechanism for sequential data input and output consistent with the requirements of the string-matching algorithm. The Carte implementation, Figure 5, streams the larger sequence of text to a pipelined implementation of the data windowing and comparison algorithm. The specification of the core algorithm, Lines 21 and 22, remains nearly unchanged from the original software code. The Carte compiler can pipeline this algorithm, automatically stalling or proceeding based on the availability of data on the input stream and capacity of the output stream. The additional Carte code connects the algorithm streams to the global common memory with additional formatting between the native width (and endianness) of the memory, 64 bits, and the character datatype, 8 bits. The "src section" pragmas define the concurrent behavior of the streaming I/O and the algorithm execution. The CDA tool can assist in constructing the formatted character streams based on simple annotations of the relevant arrays in the original software code, thereby making common hardware optimizations even more accessible to application developers.

C. Additional Optimization

Fig. 6. Optimization - Multiple String Matches

```
#pragma src parallel sections
{
  //GLOBAL MEMORY TO INPUT STREAM + FORMATTING
  ...

  //STREAM FANOUT: S8_in to S8_in1 and S8_in2
  ...

  //KEYWORD1: S8_in1
  #pragma src section
  {
      ...
      get_stream(&S8_in1, &mem);
      ...
  }
  //KEYWORD2: S8_in1 ...
  //KEYWORD3: S8_in1 ...
  //KEYWORD4: S8_in1 ...

  //KEYWORD5: S8_in2
  #pragma src section
  {
       ...
      get_stream(&S8_in2, &mem);
       ...
  }
  //KEYWORD6: S8_in2 ...
  //KEYWORD6: S8_in2 ...
  //KEYWORD6: S8_in2 ...

  //MERGE RESULTS
  ...

  //FORMATTING + OUTPUT STREAM TO GLOBAL MEMORY
  ...
}
```

The initial hardware migration focused primarily on temporal parallelism where the input stream loads the next character while the string-matching kernel operates on the current character. The algorithm can also benefit from spatial parallelism with multiple distinct keywords compared against the larger sequence of text. Streams allows for efficient duplication of the datalow to support a number of parallel kernels with minimal overhead. Figure 6 describes simultaneous string matching with eight keywords. To help minimize overhead delays such as fanout, the original input stream, S8_in, is replicated into streams S8_in1 and S8_in2. Because the actual output of each kernel is a singular bit representing a match or no match for each position in the larger sequence of text, the individual bit from each of the eight kernels can be merged into one character stream for more compact output. Again, the core algorithm is mostly unchanged and with proper annotation, the CDA tool can also assist in stream duplication and merging.

Further optimization to this implementation could involve the source and destination of the data streams. The current streaming I/O targets global common memory either on or external to the MAP processor, with currently available bank sizes ranging from 1GB to 16GB depending on location and configuration. This implementation assumes the larger sequence of text in the global common memory is first populated from some other source in the system. If the text originates from a file, the MAP processor implementation could stream directly from microprocessor memory instead with no change to the streaming dataflow of the algorithm. Additionally, the text could originate from a network interface connected to the MAP processor via the general purpose input/output (GPIO) interface, which can support an infinite stream of text and results without intervention or input from the microprocessor.

D. Quantitative Results

Table I summarizes the computation and executions times for software emulation and the physical hardware implementation of the string-matching algorithm on a 1MB sequence of text streaming through global common memory. At 2.6 seconds, the compilation time for software emulation is analogous to compilation of conventional microprocessor codes and therefore rapid enough for frequent evaluation of revisions to an algorithm design. An execution time of 100 seconds is not trivial but still sufficiently short to allow for iterative functional testing. In contrast to emulation, the actual implementation requires over 4000 seconds for compilation, but, as expected, requires considerably less execution time.

TABLE I
TOOLFLOW PERFORMANCE - 1MB TEXT

	Compilation Time (s)	Execution Time (s)
Software Emulation	2.6E+0	1.0E+2
Physical Hardware	4.2E+3	3.2E–1

TABLE II
STREAMING AND PIPELINING EFFICIENCY

Size (B)	Clock Cycles	Efficiency (%)
1K	1396	73.3
10K	10617	96.4
100K	102772	99.6
1M	1048953	99.9+

Table II describes the efficiency of the streaming and pipelining in the string-matching algorithm. This implementation processes one character of the larger sequence of text per clock cycle, excluding system latency. Longer streams of characters better amortize this latency leading to higher efficiency. Consequently, 1KB of text (i.e., 1024 characters) is likely an insufficient volume of text for maximizing performance as only 73.3% of clock cycles perform actual computation. In contrast, sequences of text of 100KB and above have over 99% efficiency and therefore minimize execution time. This predictability of performance for sufficiently large data streams can allow for more accurate analysis and evaluation of algorithm designs prior to implementation.

V. CONCLUSIONS

Perceived limitations of FPGA systems have been associated with lengthy and disconnected toolflows for application development. Efficient migration to FPGA systems historically requires straightforward algorithm specification, compilation, and execution. The SRC Carte toolflow and unified SRC-7 system architecture provide C and Fortran syntax for kernel specification, abstract yet efficient integration with system resources, robust runtime support, and complete encapsulation of low-level FPGA tools. Resulting applications require only two simple, Unix-style commands for compilation and execution. The string-matching case study demonstrated that a high-performance applications can maintain specifications analogous to legacy software while capitalizing on a streaming dataflow for FPGA systems. The compilation and execution times allow for iterative development cycles of functional refinement through software emulation followed by hardware simulation and generation of the final design implementation. Sufficiently large streams of 100KB or more demonstrated that over 99% of their clock cycles perform actual computation.

REFERENCES

[1] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.
[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithmsm, Second Edition*. MIT Press and McGraw-Hill, 2001.
[3] H. Lam, G. Stitt, et. al. Reconfigurable supercomputing: Performance, productivity, and sustainability (panel session). In *Proc. Engineering of Reconfigurable Systems and Algorithms (ERSA)*, July 13 2010.
[4] I. Gonzalez and E. El-Araby and P. Saha and T. El-Ghazawi and H. Simmler and S. Merchant and B. Holland and C. Reardon and A. George and H. Lam and G. Stitt and N. Alam and M. Smith. Classification of application development for fpga-based systems. In *Proc. National Aerospace Electronics Conference (NAECON)*, July 16–18 2008.
[5] S. Merchant and B. Holland and C. Reardon and A. George and H. Lam and G. Stitt and M. Smith and N. Alam and I. Gonzalez and E. El-Araby and P. Saha and T. El-Ghazawi and H. Simmler. Strategic challenges for application development productivity in reconfigurable computing. In *Proc. National Aerospace Electronics Conference (NAECON)*, July 16–18 2008.
[6] SRC Computers, LLC. www.srccomputers.com.
[7] SRC Computers, LLC. Introduction to the SRC-7 MAPstation system, 2009.

Statistical Data Generation System for Scientific Applications

A. Abba, F. Caponio, P. Baruzzi, A. Geraci, G. Ripamonti
Politecnico di Milano, Dipartimento di Elettronica, via C. Golgi 40, Milan MI 20133, Italy

***Abstract* - An innovative algorithm aimed to sequence of numbers generation according to a programmable statistical distribution is presented. Main features of the technique are high efficiency, low requirements in terms of implementation resources and high generation rate. Possible applications range from scientific simulations to system testing setups. In particular as reference case study, we adopted simulation of events generated by radioactive decay processes that are at the forefront in many application and research areas in medicine as in physics. The algorithm has been implemented in a low cost multi-FPGA system. A generation rate one order of magnitude higher with respect to modern PC-based solution has been achieved.**

Keywords – Pseudorandom number generation, Deterministic random bit generation, Simulation, Testing.

I. Introduction

Many methods and techniques for pseudorandom number generation (PRNG), also known as deterministic random bit generation (DRBG), are well known and consolidated [1-2]. These are algorithms for generating sequences of numbers that approximate the properties of random numbers. Although sequences that are close to truly random can be generated, pseudorandom numbers are fundamental in practice for simulations (e.g., of physical systems with the Monte Carlo method [3]), and are central in the practice of cryptography and procedural generation.

However, an increasing number of applications shows the necessity to simulate sequences of numbers that approximate at best properties of specific statistical distributions. This is the case, for example, of simulation of events generated by non-random physical processes and initialization of system testing setups. Since this is now at the forefront in many research areas in medicine as in physics, we adopted this application as reference case study to describe the proposed technique.

The radioactive decay is the process by which an atomic nucleus of an atom loses energy by emitting particles. The distribution of energy values of emitted particles depends on the source and is referred as its energy spectrum. Of course, radioactive decay is a stochastic (i.e. random) process on the level of single atoms, in that according to quantum theory it is impossible to predict when a given atom will decay. However, given a large number of identical atoms the decay rate for the collection is predictable, and for the most cases the emission instants follow a Poisson distribution. Consequently, in order to emulate a radioactive source, the primary task is to generate the emitted particle energy values, according to the source energy spectrum, and the occurrence times following Poisson distribution.

The generation of random data which follows a statistical distribution should be treated so that generated values are as representative as possible of the physical phenomenon that produces them. For example, the generation of energy of events emitted by a ^{60}Co isotope should correspond to about 80% probability of 1.33 MeV with respect to 1.17 MeV emission. This means that the emulated spectrum should grow approaching progressively from the beginning the final shape of the spectrum.

Of course, the requirement of properly emulated randomness must be combined with the need to find a method to generate data at high efficiency.

It is well know that a trivial way of generating random numbers that follow a given distribution consists in addressing the corresponding histogram by means of a white distribution. Each time the random number addresses a bin of the histogram, the corresponding count is decreased by a unit and the bin value is outputted. If the addressed bin count is null, no output is produced. The method is inherently not efficient since the probability of finding a non-zero bin count decreases with increasing generated numbers. Other algorithms require huge amounts of memory and are therefore not suited for embedded or low-cost systems.

We propose a novel technique for generating random numbers according to an arbitrary probability distribution with high efficiency, low requirements in terms of implementation resources and high generation rate.

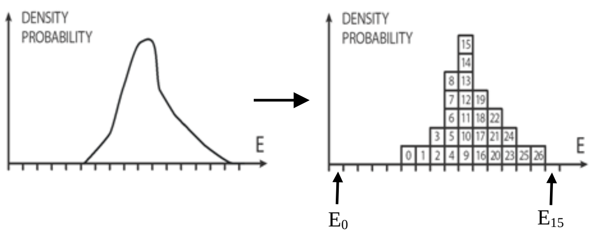


Fig.1 The plot shows a hypothetical emission energy spectrum of a radioactive decay process and its correspondent discrete representation by means of a histogram.

The algorithm has been implemented in a multi-FPGA setup that generates up to 75 Mevents per second, with a word size of 16 bits and quantized at 65,536 levels. The system can be interfaced by means of PCI-Express bus and used as co-processor providing test vectors to embedded hardware simulators or PC-based simulation software environments.

II. Methodology

As reference distribution, we consider the generic histogram depicted in Fig.1 that, for instance, represents the discretized energy emission spectrum $H(E)$ of a radioactive decay process. Aim of the technique is to generate numbers whose distribution grows approaching progressively from the beginning the shape in Fig.1. At basis of the process is a random number generator. In fact, any statistic variable x that is described by a density probability distribution $P(x)$, can be modeled by the cascade of a generator of uniformly distributed random numbers and the transform function $P(x)$ [4]. In this way, the quality of the generated statistic values depends only on the uniform number generator, which can be used for every emulated source that is characterized only by $P(x)$. Therefore, the problem is to transform a white spectrum into whatever kind of distribution. In order to simply explain how the algorithm works, let us consider that the reference histogram is composed by 16 bins, from E_0 up to E_{15}, with a maximum dynamic range equal to 16. The bin width is the spectrum resolution, while the dynamic range is the maximum height of each histogram column. The higher is the number of bins and the dynamic range, the better is the represented spectrum. However, increasing the accuracy of the spectrum is simply a matter of number of bits and this is not a problem using modern digital devices. Each column E_x of the histogram can be thought as composed by a number of small squares and represents the density probability that the event has energy between E_{x-1} and E_{x+1}; if bin x is twice higher than bin y, this means that there is twice the probability for an event to have energy E_x rather than energy E_y. The product of the column value by the bin width returns the probability. The ratio of the probabilities that an event has energy in a certain interval rather than in another one is simply the ratio between the corresponding areas below the density probability curve.

As in Fig.1, squares constituting columns are sequentially numbered. Consider the simplified case in which the total number of squares under the curve is a power of 2, e.g. $2^5=32$. Using 5 bit in the random number generator, all the 32 numbers can be obtained with the same probability, i.e. the random numbers map completely the area under the spectrum curve. Every time a random number is generated, the algorithm searches the number in the spectrum area and delivers the corresponding bin number x thus indicating the corresponding energy value E_x. If we consider again a generic x bin two times higher than a y bin, since random numbers map all the squares with equal probability, there is twice the probability that the random number picks up a square in x rather than in y column, which means that generated pulses with E_x energy are twice those with E_y energy.

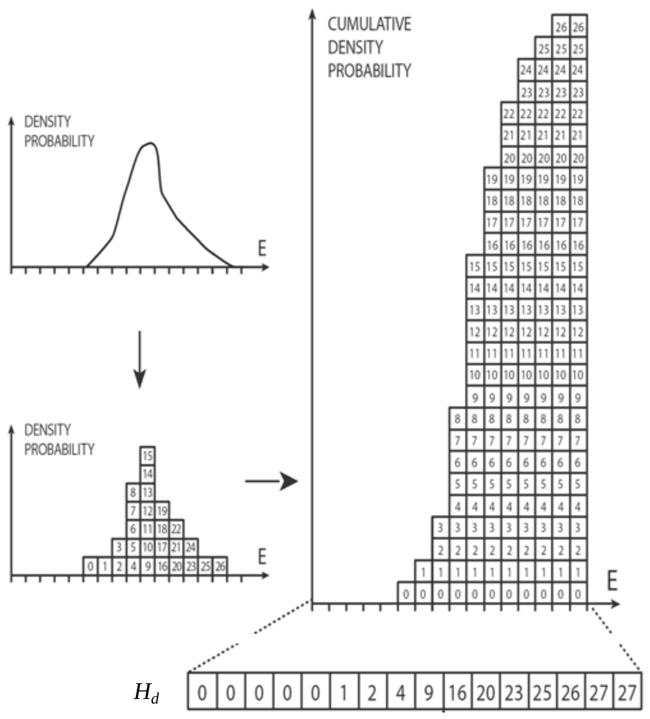


Fig.2 Conversion of the histogram of the reference probability density into the corresponding cumulative histogram.

Operatively, the histogram is converted into the equivalent cumulative histogram (Fig.2), i.e. the cumulative energy spectrum $H_c(E_x)$ is computed as integral function of the energy spectrum $H(E)$. An array is loaded with the cumulative spectrum and only one memory cell per bin is required. Using the cumulative spectrum, it is still possible to identify the bin that contains the generated random number by means of an extension of the described algorithm. For instance, (see Figs. 2 and 3), if the random number is 18, it is easy to see that it belongs to the memory cell E_{10}, which contains a number that is higher than 18 (20), while the preceding cell contains a number that is lower (16); this means that bin E_{10} contains the squares that go from 16 to 19, exactly the range in which 18 belongs. So the output energy value correspondent to the random number 18 is 10.

The fundamental advantage of this approach is the lack of required memory. In fact, alternative solutions could be faster but heavier for memory resources. For instance, using a memory composed by a number of cells equal to the number of bins multiplied by the dynamic range, that is the maximum number of squares available to draw the spectrum, each cell would contain the corresponding bin number, while the random number would be used to address the memory. In this way, no search algorithm is necessary since the data bus directly returns the bin number when the random number addresses the memory and the procedure is significantly faster. Nevertheless, this solution has the serious drawback of the memory occupancy. In fact, considering a histogram represented by 65,536 bins (i.e. 16 bit) with dynamic range equal to 65,536 (i.e. 16 bit), 8 GB of memory are necessary.

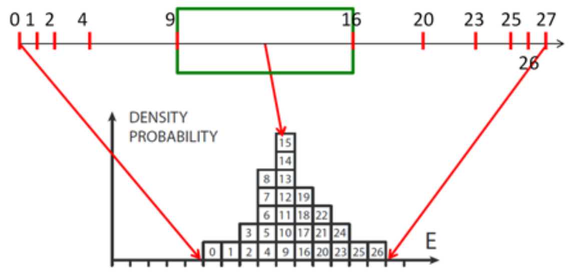


Fig.3 The marked rectangle represents the range of values of the bin E_{10}, which is the largest and consequently most likely to contain the randomly generated number. This non-uniform probability is the key of conversion from white to shaped distribution.

That has to be compared with the proposed approach, which in same operative conditions requires only 256 kB of memory.
In order to increase memory saving, we compress data storing in the array only bins with counts greater than a fixed threshold. In this case, it is important to remap each bin number of the cumulative spectrum with the correct energy value. A look-up table (LUT) is therefore necessary of size equal to the number of non-zero bins of the spectrum. Only if less than half of bins are non-zero elements, the compression is useful and performed.

III. Implementation

The system implementation is partitioned between the generation of the vector H_d, which is calculated storing only the top value of each bin of the histogram H_c (Fig. 3), and the generation of random number. The first task is performed only once and consequently has no impact on generation efficiency. On the contrary, the generation of random numbers is a critical issue in terms of operation frequency and therefore it has to be performed by means of dedicated hardware resources. Considerations on operating frequency, power dissipation and memory access bandwidth led us to choose a FPGA instead of temporal computing devices such as DSP or GPU based solutions. As above stated, conversion from white to any distribution is performed by searching the position p of the generated random number x into the vector H_d and returning p as output. Therefore, two are the main tasks of the FPGA device: the generation of random numbers x and the search of their position p into the vector H_d through a modified version of the binary search algorithm.

Efficient and accurate algorithms for random number generation are fundamental in many fields of application [5], from process simulation to cryptography. In the present application, the linear feedback shift register (LFSR) algorithm has been implemented that is a machine independent algorithm characterized by arbitrary long repetition periods, excellent statistic properties, high generation speed and limited resource expense [5]. It only needs an m-bit shift register and 1 to 3 XOR gates, and thus the resulting circuit is very small and its operation is extremely simple and fast. Furthermore, since the period grows exponentially with the size of the register, we can easily generate a large non-repetitive sequence (e.g. with a 64 bit generator running at 1 GHz, the period is more than 500 years).

Task of spectrum modulation is to look in which interval between two numbers of the vector H_d a random number is included. A very high performance algorithm to perform that in a sorted vector is the binary search [5]. Comparing the target to the middle item in the list, if the target does not match but is greater, the comparison is repeated in the upper half of the list. Otherwise, the lower half of the list is considered. The method halves the number of items to check each time, reaching convergence in logarithmic time with respect to the number of iterations.

IV. Hardware implementation

Before developing a customized hardware implementation, the generation algorithm has been validated on Xilinx FPGA devices Virtex-5 FX110T and Spartan 6 LX-25. Let’s consider the storage of distributions with dynamic range of 24 bit and resolution of 16 bit, i.e. 65536 bins. Each distribution needs a memory allocation of 1.57 Mbit to be stored. With 8 Mbit internal dual-port memory, the Virtex-5 device allows the storage of 5 distributions, namely the simultaneous operation of 5 independent generators. Considering that memories are dual-port and operating frequency can be 300 MHz, the minimum rate is 83 Mnumbers generated per second that corresponds to the maximum value of 17 clock cycles for every binary search.

Drawbacks of this solution are the cost of the device (above 1000 USD) and also the limit of 5 generators that can run simultaneously. The use of cheaper but smaller devices organized as an array has been investigated. The implemented algorithm within the FPGA consists only of a state machine and two comparators for each generator. The Xilinx FPGA Spartan-6 LX-25 is a low-cost device that has enough resources to implement the algorithm of random number generation and research but not enough memory (1 Mbit) to store even only one distribution. Consequently external asynchronous SRAM resources were attached to the FPGA device with the limit of pins available for connection. In practice, the adopted BGA FG484 package of the FPGA device leaves less than 260 pins available and each module of 2 Mbit selected low-cost RAM needs 24 data, 16 address and 2 control lines, which means 6 SRAM at most connected. At the operating frequency of 100 MHz, the single FPGA device should access memory 34 times at worst, which means a minimum rate of 17 Mnumbers generated per second.

The cost of a single computing cell, i.e. FPGA device – SRAM modules – configuration memory, is below 100 USD that is more than one order of magnitude less than Virtex-5 solution. Of course, the generation rate is sensibly slower, but 6 generators can run simultaneously. Doing a cost/benefit analysis, we decided to adopt this second approach.

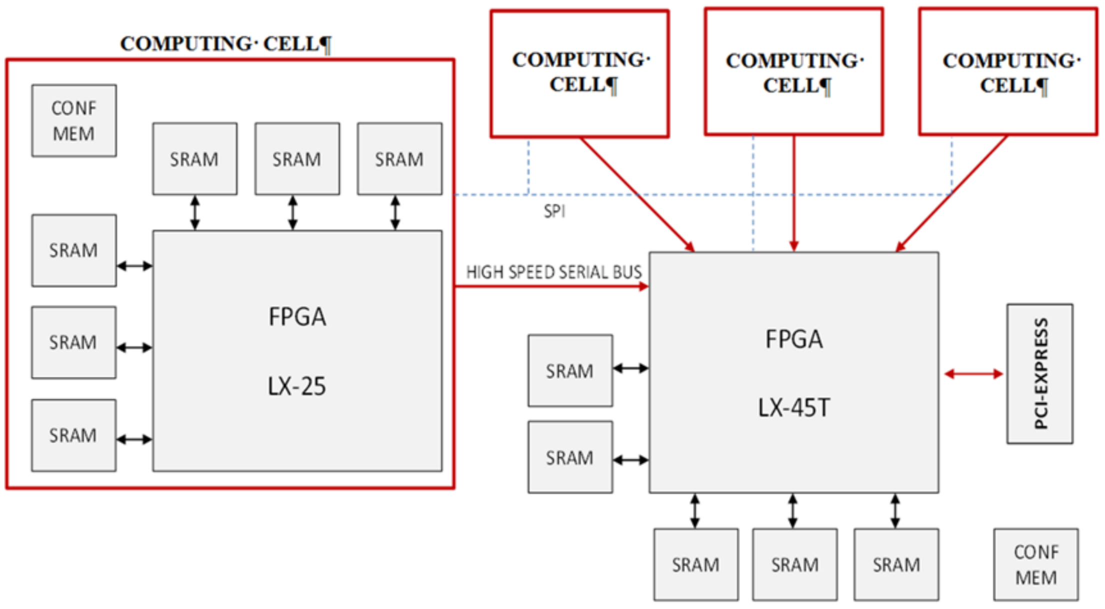


Fig.4 Block diagram of the proposed multi-FPGA architecture. The computing cell plays the role of emulators and the LX-45T device is also devoted to manage the communication to the host PC and to system clocking. The link between communication device and computing cell is performed by high-speed serial bus. The SPI bus is used to initialize the reference distribution and access to local register file in each computing cell.

The number of linked computing cells is limited by the adopted PCI-Express communication bus. In a Xilinx Spartan 6-45T FPGA device with built-in PCI-Express blocks, we implemented a master DMA controller, whose transfer rate to host PC is just below 200 MB/s per lane. Since we decided to use just 1 lane in order to make the system compliant with common 1x PCI-Express slots, the maximum transfer rate of 16 bit random numbers is 100 Mnumbers/s that corresponds to a maximum number of 5 connected computing cells.
Overall, at a cost equal to ¼ with respect to the use of a single Virtex-5 FPGA device, the available system is much more versatile as it can simulate from 30 independent distributions at a rate of 3 Mnumbers generated per second to 1 distribution at a rate of 88 Mnumbers generated per second.

Figure 4 shows a block diagram of the realized prototype. There are 4 Xilinx FPGA Spartan-6 LX-25 devices, which implement the function of generation. Besides PCI-Express communication task to host PC, the Xilinx FPGA Spartan-6 LX-45 device also implements 5 generators. The connection architecture among computing cell LX25 and LX45T devices is a star configuration through serial bus lines operating at 300 MHz. Initialization of computing cells is performed by means of a SPI bus. Each computing cell contains 6 asynchronous SRAM modules, so as not to suffer from pipeline delay and realize a true random access. The SRAM access time is 10 ns.

In order to verify the convenience to use the system in place of available PC-based solutions, the developed algorithm was implemented in C language and run on a Core i7 945 PC over-clocked at 4.5 GHz. Using 1 core, the generation rate is about 5 Mnumbers/s. Parallelizing the algorithm, however, the rate does not increase significantly and settles down to 7.5 Mnumbers generated per second, probably because of the bottleneck due to memory access. Consequently, the proposed system shows a speedup of 12 times at a cost only half the CPU alone and with power dissipation of 15 W compared to 100 W of the PC based solution. In addition, there are no substantial benefits in the use of GPGPU since the local memory of each multiprocessor (both texture and constant) is too small to hold the cumulative vector H_d.

V. CONCLUSIONS

An algorithm for getting statistic properties from a histogram of events has been conceived and implemented. The algorithm has efficiency equal to 100% and has been validated through simulation also with reference to the specific application of emulation of radiation detection setups.

The system has been prototyped and is being fully tested on a processing digital platform based on a FPGA device.

The proposed solution based on FPGA has been shown to achieve a level of quality/price ratio even better than PC-based counterparts at the state of the art.

REFERENCES

[1] M. Luby, "Pseudo-randomness and Cryptographic Applications", Princeton Univ. Press, 1996.
[2] L. Devroye, "Non-Uniform Random Variate Generation", Springer Verlag, New York, 1986.
[3] K. Binder, D.W. Heermann, "Monte Carlo Simulation in Statistical Physics", Springer, 2010.
[4] A. Papoulis, S.U.Pillai, "Probability, random variables and stochastic processes", McGraw-Hill, 4th Ed., 2002.
[5] D.E.Knuth, "Art of Computer Programming, Volume 2: Semi-numerical Algorithms", Addison-Wesley, 3rd Ed., 1997.

An FPGA Based on Synchronous/Asynchronous Hybrid Architecture with Area-Efficient FIFO Interfaces

Masanori Hariyama, Yoshiya Komatsu, Shota Ishihara, Ryoto Tsuchiya, and Michitaka Kameyama
Graduate School of Information Sciences, Tohoku University
Aoba 6-6-05, Aramaki, Aoba, Sendai, Miyagi, 980-8579, Japan

Abstract— *This paper presents an FPGA architecture that combines synchronous and asynchronous architectures. Datapath components such as logic blocks and switch blocks are designed so as to run in asynchronous and synchronous modes. Moreover, a logic block is presented that implements area-efficient First-in-first-out(FIOF) interfaces, which are usually used for communication between synchronous and asynchronous logic cores. The FPGA based on the area-hybrid architecture is fabricated in a 65nm process.*

Keywords: FPGA, Reconfigurable VLSI, Self-timed architecture, Delay-insensitive architecture.

1. Introduction

Field-programmable gate arrays (FPGAs) are widely used to implement special-purpose processors. FPGAs are cost-effective for small-lot production because functions and interconnections of logic resources can be directly programmed by end users. Despite their design cost advantage, FPGAs impose large power consumption overhead compared to custom silicon alternatives [1]. The overhead increases packaging costs and limits integrations of FPGAs into portable devices. In FPGAs, the power consumption of clock distribution is a serious problem because it has an enormously large number of registers than custom VLSIs. To cut the clock distribution power, some asynchronous FPGAs has been proposed [2], [3], [4], [5], [6]. From references [4]-[6], the asynchronous architecture is more power-efficient than the synchronous one under the low-workload condition. Under the high-workload condition, the asynchronous architecture is less advantageous in power than the synchronous one because of its overhead of complex control circuitry. In actual situations, evan a single application consists of many tasks with various workloads. The best way to minimize the power consumption is to use both of asynchronous and synchronous architectures for a single application. For this purpose, we have reported the FPGA architecture based on the hybrid of a synchronous and asynchronous architectures[7]. However, it suffers from the lack of the area-efficient communication interface between synchronous and asynchronous cores.

This paper proposes an FPGA architecture based on the hybrid of asynchronous and synchronous architectures that can implement area-efficient communication interface. Datapath components such as logic blocks and switch blocks are used for both of asynchronous and synchronous modes; Each of the blocks is programmed to be an asynchronous block or a synchronous block in advance. When designing the hybrid datapath components, the major issue is to sharing the datapath resources efficiently. For this purpose, we propose dual-bit operation where a one-bit logic block and interconnection resource in asynchronous mode is exploited as two-bit used for a two-bit logic block and interconnection resources in synchronous mode. As a result, the datapath resources are fully exploited in both modes. In order to use common Look-up tables(LUTs) of the logic block in both modes, 4-phase dual-rail encoding is adopted. Another design issue is to implement area-efficient communication interface. In general, the processing speeds of the synchronous and asynchronous cores are different from each other. Therefore, the First-in-first-out(FIFO) interface is usually used to absorb the difference of the processing speeds. However, FIFO interface imposes a large hardware overhead. To solve this problem, this paper presents the logic block structure that can implements area-efficient FIFOs. The FPGA based on the hybrid architecture is implemented in a 65nm process.

2. Architecture

2.1 Asynchronous protocols

Asynchronous encoding schemes are mainly classified into

- Single-rail encoding (ex. bundled-data encoding)
- Dual-rail encoding
  (ex. 4-phase dual-rail encoding, LEDR encoding)

The bundled-data encoding is most common one in the single-rail encoding. The bundled-data encoding is the most frequently-used way in ASICs since its hardware overhead is relatively small. The major disadvantage is that it requires the constraint of the delay length. If the data path is fixed in advance, it is relatively easy to meet the constraint by optimizing layouts of wires. On the other hand, in reconfigurable VLSIs such as FPGAs, it is not easy to always meet the constraint since the data path is programmable.

The dual-rail encoding encodes a bit onto two wires. In dual-rail encoding, value is made implicit in the request and

Table 1: Code table of 4-phase dual-rail encoding

	Code word (T, F)
Data 0	(0,1)
Data 1	(1,0)
Spacer	(0,0)

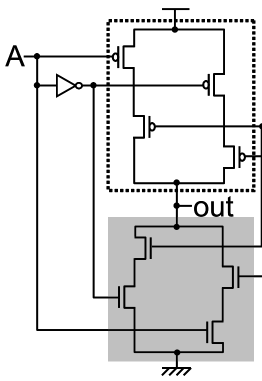


Figure 1: XOR gate of synchronous architecture

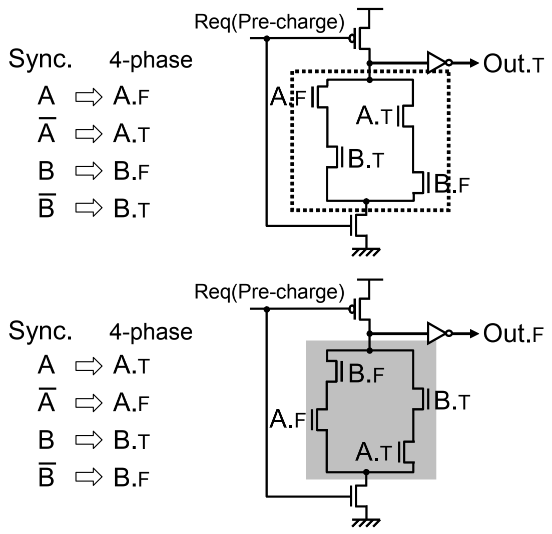


Figure 2: XOR gate of 4-phase dual-rail architecture

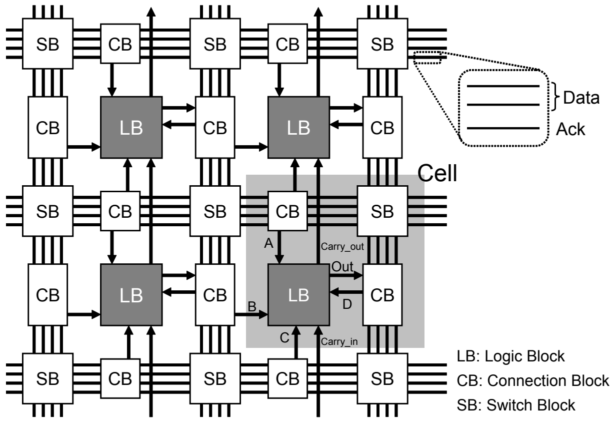


Figure 3: Overall architecture.

no delay insertion is therefore required[8]. Hence, the dual-rail encoding is the ideal one for reconfigurable VLSIs. 4-phase dual-rail encoding is the most common one in dual-rail encodings. Table 1 shows the code table of 4-phase dual-rail encoding. The data value 0 is encoded as $(0, 1)$ and 1 is encoded as $(1, 0)$. Moreover, the spacer is encoded as $(0, 0)$. Figure **??** shows the example where data values 0, 0 and 1 are transferred. The main feature is that the sender sends spacer after a data value. The receiver knows the arrival of a data value by detecting the change of either bit: 0 to 1. The insertion of spacers makes the encoding law simple. This results in a simple hardware of the function unit. Figures 1 and 2 show the XOR gates of synchronous (synchronous XOR) and 4-phase dual-rail (4-phase dual-rail XOR) architecture respectively. The circuit for generate $out.t$ of the 4-phase dual-rail XOR gate is similar to the pMOS network of the synchronous XOR gate. On the other hand, the circuit for generate $out.f$ of the 4-phase dual-rail XOR gate is similar to the nMOS network of the synchronous XOR gate. This similarity causes the function unit of 4-phase dual-rail architecture to implement easily. The number of the transistors of the synchronous XOR gate is 12, while the 4-phase dual-rail XOR gate is 16. Accordingly, the hardware overhead of 4-phase dual-rail architecture is smaller than that of other dual-rail architecture.

2.2 Overall architecture

Figure 3 shows the overall architecture of the proposed FPGA. The FPGA consists of a mesh-connected cellular array likes conventional FPGAs. As mentioned in the previous section, 4-phase dual-rail encoding is employed as the asynchronous protocol because of its similarity to synchronous circuits. The logic blocks, connection blocks and switch blocks are used for asynchronous architecture and synchronous architecture. The clock-tree network is designed based on H-tree topology; For simplicity, the clock tree is not illustrated in this figure. The clock signal is distributed to all the registers in the logic blocks. Since the chip presented in this paper is a prototype, the simplest 2-input LUT is used in the logic block.

2.3 Logic block structure

As shown in Fig. 4, the LUT is constructed by two same smaller LUTs. In asynchronous mode shown in Fig. 4(a), the upper and lower LUTs are used for out.t and out.f, respectively. In synchronous mode shown in Fig. 4(b), the upper and lower LUTs are used for different bits respectively,

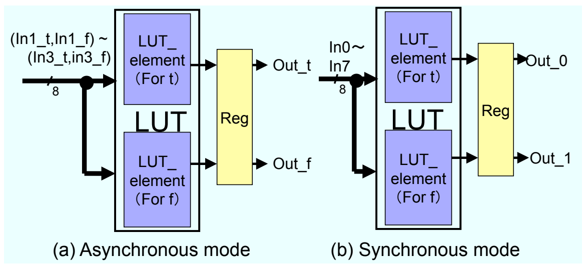


Figure 4: Resource sharing for the logic block of the hybrid architecture.

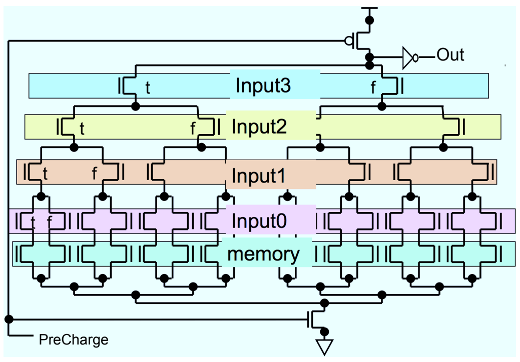


Figure 5: Block diagram of the LUT of the hybrid architecture.

and the all LUTs are fully exploited.

Figure 5 shows the circuit of the LUT. As explained in the previous section, the logic circuit of 4-phase dual-rail encoding is quite similar to dynamic circuit of the synchronous circuit. Based on this observation, the LUT for hybrid architecture is designed using the dynamic circuit. Hence, completely common LUTs can be used for both of asynchronous and synchronous modes.

Another design issue on the hybrid architecture is to implement area-efficient communication interface between synchronous and asynchronous cores. In general, the processing speeds of a synchronous core and an asynchronous core are different. FIFO interface is commonly used to absorb the difference as shown in Fig. 6. If the processing speed of the sender core is higher than that of receiver core, the FIFO stores the data from the sender core until the unbalance of the processing speed is resolved. Moreover, the protocol converter is required to convert data from the synchronous format to the asynchronous one or vice versa. Figures 7 and 8 shows the functions of the protocol converter and the FIFO interface. The number of cells in the FIFO depends on the design and is not determined in advance. Therefore, it is desirable to implement the FIFO cell using a logic block.

Figure 9 shows the structure of the logic block with the FIFO function; Figure 10 shows the FIFO-cell mode of the logic block. Regards can see that the area overhead for the

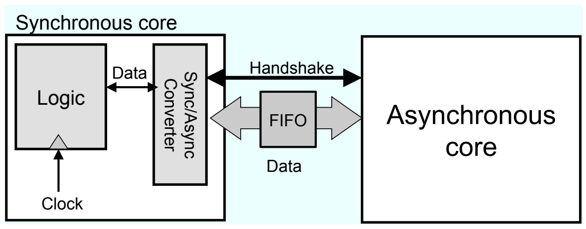


Figure 6: Interface between a synchronous and asynchronous cores.

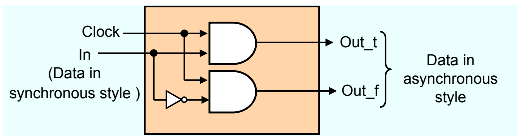


Figure 7: Function of the converter from synchronous style to asynchronous style.

FIFO interface is relatively small in the logic block.

3. Evaluation

The FPGA based on the hybrid architecture is implemented in a 65nm CMOS process. The supply voltage is 1.2V. The processing performance of the asynchronous mode corresponds to that of the 720MHz of the synchronous FPGA. Table 2 summarizes the comparison result of the cells of synchronous architecture and the hybrid architecture in synchronous mode. Thanks to the resource sharing, the energy and area overheads are just 29% and 22%. Table 3 summarizes the comparison result of the cells of asynchronous architecture[4], and the proposed hybrid architecture in asynchronous mode. The transistor-count overhead is as small as 18%.

4. Conclusion

This paper proposes an FPGA based on the hybrid of asynchronous and synchronous architecture, where the FIFO interface between synchronous and asynchronous cores is implemented using logic blocks with the small overhead.

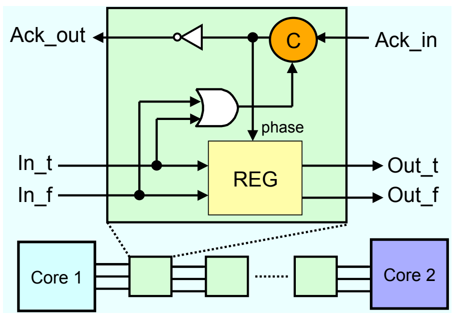


Figure 8: Function of the FIFO interface.

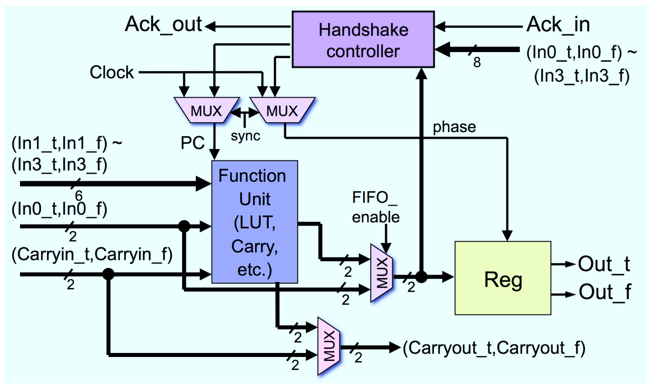


Figure 9: Structure of the logic block with the FIFO function.

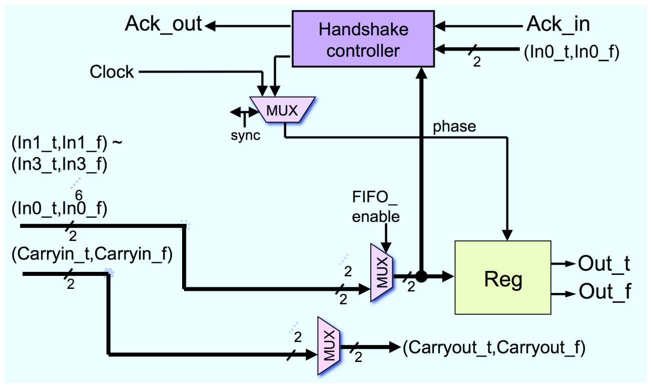


Figure 10: FIFO cell implemented using the logic block.

The FIFO interface will be also efficient for power reduction. If the number of the stored data becomes large, it means that the processing speed of the sender is much higher than that of the receiver. Then, the power consumption can be reduced by lowering the supply voltage of the sender. Moreover, the proposed architecture with FIFO has higher flexibility than the conventional GALS(Globally Asynchronous and Locally Synchronous) architecture. As a future work, we are evaluating the hybrid architecture on some practical benchmarks. Developing the CAD environment is also important topic.

Acknowledgment

This work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with STARC, Fujitsu Limited, Matsushita Electric Industrial Company Limited, NEC Electronics Corporation, Renesas Technology Corporation, Toshiba Corporation, Cadence Design Systems Inc. and Synopsys Inc.

Table 2: Comparison of cells of synchronous and the hybrid architecture.

	Sync.	Our hybrid (Sync. mode)
Energy per data set [fJ]	355	459 (129%)
Delay[ps]	263	367 (151%)
Transistor count	1703	2069 (122%)

Table 3: Comparison of cells of asynchronous and the hybrid architecture.

	Async [4]	Our Hybrid (LB mode)	Our Hybrid (FIFO mode)
Energy per data set[fJ]	755	925 (122%)	664
Delay[ps]	482	713 (148%)	422
Transistor count	1757	2069 (118%)	

References

[1] H. Z. V. George and J. Rabaey, "The design of a low energy FPGA," in *Proceedings of 1999 International Symposium on Low Power Electronics and Design*, Californai, USA, Aug 1999, pp. 188–193.
[2] J. Teifel and R. Manohar, "An asynchronous dataflow FPGA architecture," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1376–1392, 2004.
[3] R. Manohar, "Reconfigurable Asynchronous Logic," in *Proceedings of IEEE Custom Integrated Circuits Conference*, Sept. 2006, pp. 13–20.
[4] M. Hariyama, S. Ishihara, and M. Kameyama, "Evaluation of a Field-Programmable VLSI Based on an Asynchronous Bit- Serial Architecture," *IEICE Trans. Electron*, vol. E91-C, no. 9, pp. 1419–1426, 2008.
[5] M. Hariyama, S. Ishihara, , and M. Kameyama, "A Low-Power Field-Programmable VLSI Based on a Fine-Grained Power-Gating Scheme," in *Proceedings of IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Knoxville(USA), Aug 2008, pp. 430–433.
[6] S. Ishihara, Y. Komatsu, and M. K. Masanori Hariyama, "An Asynchronous Field-Programmable VLSI Using LEDR/4-Phase-Dual-Rail Protocol Converters," in *Proceedings of The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas(USA), Jul 2009, pp. 145–150.
[7] M. Hariyama, R. Tsuchiya, S. Ishihara, and M. Kameyama, "A Field-Programmable VLSI Based on Synchronous/Asynchronous Hybrid Architecture," in *Proceedings of The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas(USA), Jul 2010, pp. 217–274.
[8] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

SESSION

POSTERS

Chair(s)

TBA

Synthesis of Simulink based Models for Parallel Architectures including FPGAs and Multi-core Processors for an Infrared Scene Generator System

Rhonda Gaede, Chris Smeal, Jeff Kulick

The Department of Electrical and Computer Engineering
The University of Alabama in Huntsville, Huntsville, AL, USA

Abstract - *Field Programmable Gate Arrays (FPGAs) can substantially improve the performance of scientific and engineering applications through their massively parallel architecture. The newest FPGA components have thousands of arithmetic units and hundreds of individually and simultaneously addressable banks of memory. However, to utilize this parallelism designers have to create detailed designs in hardware description languages such as VHDL [3]. This has been a significant impediment to the more widespread use of FPGAs in engineering systems. Many engineers do their engineering analysis and system design in modeling languages such as Simulink. This paper explores the issues of implementing parallel computing systems utilizing the full extent of FPGA hardware starting with serial single thread computational models developed in Simulink and the problems in this process.*

Keywords: *FPGAs, Simulink, parallelism*

1 Introduction

Performance acceleration of scientific and engineering computations is one of the core interests of the Computer Engineering group at the University of Alabama in Huntsville. Recently, we were asked to examine the problem of building a real-time scene generator for producing infrared (IR) images for verification and validation of IR seekers. Since current IR scene generation techniques [1, 2, 4] operate far slower than real-time, pre-production of IR scene data is required. As part of this effort, we decided to examine the feasibility of using Model Based Design to produce operational implementations of an IR scene generator.

Model Based Design (MBD) is becoming one of the key methodologies used to create scientific and engineering systems as organizations try to minimize system defects. Many opportunities for errors occur when different representations of a system exist. MBD provides Simulink blocks and implementations that subject matter experts can use to model the System. This model can then be realized automatically using synthesis techniques and the implementations. For IR scene generation, we felt that the needed performance gains could only be achieved by exploiting inherent parallelism. We examined several Simulink approaches to accelerating model execution including: replicating individual computation kernels using the VHDL construct *for generate*, using the *for iterator* construct, and replicating individual computation kernels using MATLAB scripts

The fundamental problem is that Simulink is optimized for single-threaded execution. The SME needs to describe a parallel architecture in Simulink in a way that guarantees that the structure is preserved even as the model is synthesized into an implementation. The options for parallel architectures include multi-core processors running multi-threaded applications and field programmable gate arrays (FPGAs).

2 Infrared Scene Generation

IR scene generation has been used for many years in diverse applications. The area studied is typically divided into many identical spatial elements or “pixels”. Each pixel model includes heat exchange with adjacent elements as well as thermal interaction with the environment.

In this paper, a pixel element was created that would observe the temperature of each of its neighbors, provide its temperature to its neighbors in the same clock cycle and update its temperature based on its current temperature and the relative temperatures of each of its neighbors. If neighbors are missing, the pixels own value is used. We started with the following equations that represent the behavior of the pixel.

$$T_{t+1} = T_t + [K * (T_{navg}) / 2 - T_t \qquad (1)$$

T_t = Current pixel temperature

T_{navg} =Average temperature of surrounding pixels

K = Thermal conductivity constant

The average temperature of the surrounding pixels is calculated and multiplied by the thermal conductivity to represent the rate at which heat spreads, and then the previous temperature is subtracted from that average. The formula has the advantage of using only basic math but does not include frictional heating or ablative cooling effects.

The next step was to produce a Simulink model of a pixel. In order to be exported to VHDL, fixed point variables were used. Fixed-point arithmetic will only work if the dynamic range of the data values is computed and each pixel temperature can take on the entire temperature range. The range of the temperatures dictates how many bits we allocate left of the binary point. The precision needed dictates how many bits are allocated right of the binary point.

3 Model Duplication

Simulink has two replicator constructs; *for each* and *for iterate*. We first investigated *for each* as a mechanism for arraying the pixels and interconnecting them. We found out that the *for each* could create multiple simulatable copies of the pixel but only using manual interconnection. Further, when we generated C++ code for the *for each* model, we got three calls to a single routine rather than three routines. The *for each* is also not supported by HDL Coder so we were unable to generate HDL. Our next approach was to generate the VHDL for a single pixel using HDL coder and to replicate the cell using the VHDL *for generate* construct. Approximately 230 IR pixels were able to fit into a Virtex6 VLX 760 part.

To evaluate the efficacy of the parallel hardware approach, we compared its performance to a serial C# implementation of the algorithm. One of the issues encountered comparing FPGAs to microprocessor implementations is that in the end it is easy to get data into and out of a processor while it is difficult to get data into and out of an FPGA. In order to do a comparison between implementations we augmented the FPGA design with a serial IO interface. In order for the comparison to be valid, clock cycles for serial I/O had to be added to the FPGA results.

When the number of iterations between readouts of the 230 processing elements is small, i.e., 128, the FPGA is about 10x faster than the microprocessor. However when the number of iterations between readouts is larger, say 100M, then the FPGA is almost 300 times faster.

Having seen significant performance improvement from the hardware implementation, we attempted to bring the arrayed hardware back into the Simulink environment using EDA Simulator Link. Though we could bring the arrayed hardware in as a black box, and simulate it within the Simulink environment, there is no way to backannotate the timing information found in the VHDL model into Simulink. Thus, there is no way to optimize the Simulink model for the individual pixels for performance tuning. At this point, we turned to another candidate Simulink construct, *for iterator*, to see whether we can produce an architecturally accurate model of the parallel implementation of the IR scene generator.

We decided to experiment with a very simple basic block rather than the full IR scene generator pixel so that it was easy to understand the results. We hoped that the effect of the *for iterator* was to create N copies in the same manner that the *for generate* VHDL construct did. Whereas the Simulink simulation performed as expected, we did not see the hoped for structural replication. The Real Time workshop generated C++ had a loop executed N times, rather than N pieces of code. Further, we tried using HDL Coder to generate VHDL and saw a warning in Simulink. Examination of the VHDL showed that the *for generate* construct was used, however it was used incorrectly, allowing no simulation or synthesis.

MATLAB scripting, our final strategy, did create an architecturally accurate, scalable Simulink model. Unfortunately, Simulink does not guarantee that it will maintain this structure when it generates C++ code. The two *for iterate* constructs in the model are collapsed into a single for loop in the C++ code.

In subsequent experiments, we were able to maintain the structure by using model referencing and thought that we should combine the arraying capability of the MATLAB script and model referencing. We ran the script with model reference blocks and then generated both C++ code and VHDL from the arrayed model with mixed results. The C++ code conformed to our expectations but the VHDL did not. The VHDL did represent the arrayed nature of the model but did not push down into the subsystem to create a lower level representation.

4 Conclusions and Future Work

In an earlier work [5], we presented a first effort to use MBD to build FPGA-based computational systems. We have now realized parallel implementations that more fully utilize the capabilities of FPGA fabrics. A primary goal has been to construct an architecturally accurate representation of the parallel system in a Simulink model so that optimization of the design can be carried out at the model level. Several issues have arisen that have required different approaches to building multithreaded implementations for microprocessors and fully parallel implementations for FPGAs.

We plan to continue this work focusing on generating partitioned models that can be deployed on multiple platform architectures simultaneously. In radar, for example, the input and signal processing might be done on an FPGA fabric while the target identification and tracking might be done on a multi-core microprocessor. Our future work will examine this path as well as developing safety critical designs that require strict traceability between all levels of the system from high level requirements to net list implementations.

5 References

[1] An Approximate Ablative Thermal Protection System Sizing Tool for Entry System Design, John A. Dec and Robert D. Braun, *44th AIAA Aerospace Sciences Meeting and Exhibit 9 - 12 January 2006, Reno, Nevada.*

[2] Towards A Multi-FPGA Infrared Simulator, Vinay Sriram and David Kearney.

[3] VHDL.org

[4] Infrared Scene Generation (IRSG) Developer's Guide, M. Eric Rouleau, *Defence R&D Canada, September 2008.*

[5] A Model-Based Design Approach For Realizing Signal Processing Systems in FPGAs Rhonda Gaede, David Moody, Michael Adderley, Charles Fulks, Laurie Joiner, Jeffrey Kulick, University of Alabama, Huntsville, Alabama, USA, Presented at ERSA 2010, Las Vegas, July 13, 2010

SESSION

LATE PAPERS

Chair(s)

TBA

Cybersecurity: From Engineering to Science

Extended Abstract

Carl E. Landwehr
Institute for Systems Research
University of Maryland
College Park, USA
Landwehr@isr.umd.edu

Abstract— **Engineers design and build artifacts - bridges, sewers, cars, airplanes, circuits, software -- for human purposes. In their quest for function and elegance, they draw on the knowledge of materials, forces, and relationships developed through scientific study, but frequently their pursuit drives them to use materials and methods that go beyond the available scientific basis. Before the underlying science is developed, engineers often invent rules of thumb and best practices that have proven useful, but may not always work. Drawing on historical examples from architecture and navigation, we consider the progress of engineering and science in the domain of cybersecurity.**

Keywords - cybersecurity research; science; engineering; foundations

Over the past several years, public interest has increased in developing a "science of cybersecurity," often shortened to "science of security" [1,2]. In modern culture, and certainly in the world of research, "science" is seen as having positive value. Things scientific are preferred to things unscientific. A "scientific foundation" for developing artifacts is seen as a strength. If one invests in research and technology, one would like those investments to be scientifically based or at least to produce scientifically sound (typically meaning reproducible) results.

This yearning for a sound basis that we might use to secure computer and communication systems against a wide range of threats is hardly new. Lampson characterized access control mechanisms in operating systems in 1971, forty years ago. Five years later Harrison, Ruzzo, and Ullman analyzed the power of those controls formally. It was 1975 when Bell and LaPadula and Walter, et. al., published their state-machine based models to specify precisely what was intended by "secure system" [5,6]. These efforts, preceded by the earlier Ware and Anderson reports [7,8], and succeeded by numerous attempts to build security kernel-based systems on these foundations, aimed to put an end to a perpetual cycle of "penetrate and patch" exercises.

Beginning in the late 1960's, Djikstra and others developed the view of programs as mathematical objects that could and should be proven correct, that is, their outputs should be proven to bear specified relations to their inputs. Proving the correctness of algorithms was difficult enough; proving that programs written in languages with informally defined semantics implemented the algorithms correctly was clearly infeasible without automated help. In the late 1970's and early 1980's several research groups developed systems aimed at verifying properties of programs. Proving security properties seemed less difficult and therefore more feasible than proving general correctness, and significant research funding flowed into these verification systems in hopes that they would enable sound systems to be built.

This turned out not to be so easy, for several reasons. One reason is that capturing the meaning of "security" precisely is difficult in itself. In 1985, John McLean's "System Z" showed how a system might conform to the Bell-LaPadula model yet still lack the security properties its designers intended [9]. In the fall of 1986, Don Good, a developer of verification systems, wrote in an e-mail circulated widely at the time: "I think the time has come for a full-scale redevelopment of the logical foundations of computer security…" Subsequent discussions led to a workshop devoted to "Computer Security Foundations," inaugurated in 1988 that has met annually since then and to the founding of *The Journal of Computer Security* a few years later.

All of this is not to say that the foundations for a science of cybersecurity are in place. They are not. But the idea of searching for them is also not new, and it's clear that establishing them is a long term effort, not something that a sudden infusion of funding is likely to achieve in a short time.

But lack of scientific foundations does not necessarily mean that practical improvements in the state of the art cannot be made. Consider two examples from centuries past.

The Duomo, the Cathedral of Santa Maria Del Fiore, is one of the glories of Florence. At the time the first stone of its foundations was laid in 1294, the birth of Galileo was almost 300 years in the future, and of Newton, 350 years. The science of mechanics did not really exist. Scale models were built and used to guide the cathedral's construction, but at the time the construction began, no one new how to build a dome of the planned size. Ross King tells the fascinating story of the competition to build the dome, which still stands atop the cathedral more than 500 years after its completion, and of the many innovations embodied both in its design and in the methods used to build it [10]. It is a story of human innovation and what we might today call engineering design, but not one of establishing scientific understanding of architectural principles.

About two hundred years later, with the advent of global shipping routes, the problem of determining the east-west position (longitude) of ships had become such an urgent problem that the British Parliament authorized a prize of £20,000 for its solution. It was expected that the solution would come from developments in mathematics and astronomy, and so the Board of Longitude, set up to administer the prize competition, drew heavily on mathematicians and astronomers. In fact, as Dava Sobel engagingly relates, the problem was solved by the development, principally by a single self-taught clockmaker named John Harrison, of mechanical clocks that could keep consistent time even in the challenging shipboard environments of the day [11].

I draw two observations from of these vignettes in relation to the establishment of a science of cybersecurity. The first is that scientific foundations frequently follow, rather than precede, the development of practical, deployable solutions to particular problems. I claim that most of the large scale software systems on which society today depends have been developed in a fashion that is closer to the construction of the Florence cathedral or Harrison's clocks than to the model of specification and proof espoused by Dijkstra and others. The IETF motto asserting a belief in "rough consensus and running code" [12] reflects this fundamentally utilitarian approach. This observation is not intended as a criticism either of Dijkstra's approach or that of the IETF. We simply must realize that while we are searching for the right foundations, construction will continue.

Second, I would observe that the establishment of proper scientific foundations takes time. As we have seen, Newton's law of gravitation followed Brunelleschi by centuries and could just as well be traced all the way back to the Greek philosophers. One should not expect that there will be sudden breakthroughs in developing a scientific foundation for cybersecurity, and one shouldn't expect that the quest for scientific foundations will have major near term effects on the security of systems currently under construction.

What would a scientific foundation for cybersecurity look like? Science can come in several forms, and these may lead to different approaches to a science of cybersecurity[13]. Aristotelian science was one of definition and classification. Perhaps it represents the earliest stage of an observational science, and we see it both in attempts to provide a precise characterization of what security means [14] but also in the taxonomies of vulnerabilities and attacks that presently plague us. A Newtonian science might speak in terms of forces, statics and dynamics. Models of computational cybersecurity based in automata theory and modeling access control and information flow might fall in this category, as well as more general theories of security properties and their composability, as in Clarkson and Schneider's recent work on hyperproperties [15]. A Darwinian science might reflect the pressures of competition, diversity, and selection. Such an orientation might draw on game theory and could model behaviors of populations of machines infected by viruses or participating in botnets, for example. A science drawing on the ideas of prospect theory and behavioral economics developed by Kahneman, Tversky, and others might be used to model risk perception and decision making by organizations and individuals [16].

In conclusion, I would like to recall Herbert Simon's distinction of science from engineering in his landmark book, *Sciences of the Artificial* [17]:

Historically and traditionally, it has been the task of the science disciplines to teach about natural things: how they are and how they work. It has been the task of the engineering schools to teach about artificial things: how to make artifacts that have desired properties and how to design.

From this perspective, Simon develops the idea that engineering schools should develop and teach a "science of design." Despite the complexity of the artifacts humans have created, it is important to keep in mind that they are indeed artifacts. We have the ability, if we have the will, to reshape them to better meet our needs. A science of cybersecurity should help us understand how to create artifacts that provide the computational functions we desire without being vulnerable to relatively trivial attacks and without imposing unacceptable constraints on users or on system performance.

References

[1] Workshop Report, NSF/IARPA/NSA Workshop on the Science of Security, November, 2008. Available at: http://sos.cs.virginia.edu/

[2] Science of Cyber-Security. JSR-10-102. November, 2010. Available at: http://www.fas.org/irp/agency/dod/jason/cyber.pdf

[3] Lampson, B.W. Protection, Proc. Fifth Princeton Symp. on Information Sciences and Systems, Princeton University, March1971, pp. 437-443. Reprinted in ACM SIGOPS *Operating Systems Rev. 8*, 1 pp.18-24, Jan. 1974.

[4] Harrison, M.A, W. L. Ruzzo and J. D. Ullman. "Protection in Operating Systems". *Communications of ACM. 19*(8):461--471, August 1976.

[5] Walter, K. G., Ogden, W. F., Gilligan, J. M., Schaeffer, D. D., Schaen, S. L., and Shumway, D. G., Initial Structured Specifications for an Uncompromisable Computer Security System, ESD-TR-75-82, ESD/AFSC, Hanscom AFB, Bedford, MA (July 1975) [NTIS AD-A022 490].

[6] Bell, D. E., and La Padula, L., Secure Computer System: Unified Exposition and Multics Interpretation, ESD-TR-75-306, ESD/AFSC, Hanscom AFB, Bedford, MA (1975) [DTIC AD-A023588] Available at: http://nob.cs.ucdavis.edu/history/papers/bell76.pdf .

[7] Ware, W., Security Controls for Computer Systems: Report of Defense Science Board Task Force on Computer Security, Rand Report R609-1 (Feb. 1970) Available: http://nob.cs.ucdavis.edu/history/papers/ware70.pdf

[8] Anderson, J. P., Computer Security Technology Planning Study, ESD-TR-73-51, ESD/AFSC, Hanscom AFB, Bedford, MA (Oct. 1972) [NTIS AD-758 206] Available: http://nob.cs.ucdavis.edu/history/papers/ande72a.pdf.

[9] McLean, J. "A Comment on the 'Basic Security Theorem' of Bell and LaPadula," *Information Processing Letters 20* (2), pp. 6770 (Feb. 1985).

[10] King, Ross. *Brunelleschi's Dome*. Penguin Books, New York, 2000.

[11] Sobel, Dava. *Longitude*. Penguin Books, New York, 1995.

[12] The Tao of IETF: A Novice's Guide to the Internet Engineering Task Force. Network Working Group RFC 4677, 2006. Available at: http://www.rfc-editor.org/rfc/rfc4677.txt

[13] Cybenko, George. I am indebted to George Cybenko for this observation and the subsequent four categories. Personal communication, spring, 2010.

[14] Avizienis, A., J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing 1*(1), pp. 11-33, Jan. 2004.

[15] Clarkson, M.R. and F.B. Schneider. "Hyperproperties". *Journal of Computer Security, 18*, (6), pp. 1157-1210, Dec. 2010.

[16] Kahneman, Daniel, and Amos Tversky (1979) "Prospect Theory: An Analysis of Decision under Risk", *Econometrica, XLVII*, pp. 263-291,1979.

[17] Simon, H.A. *Sciences of the Artificial*. MIT Press, Cambridge MA. 3rd Edition, 1996.

Rapid Implementation of Floating-point Computations Using Phase-Coherent Dynamically Configurable Pipelines

D. Rutishauser and R. Shuler
Avionic Systems Division, NASA Johnson Space Center
Houston, Texas, U.S.A.

Abstract - *The Phase-Coherent Dynamically Configurable Pipeline is a concept for the rapid implementation of pipelined computational algorithms in configurable hardware. The approach allows a high level of sharing of floating-point resources among multiple computations. The concept features a simple tag-based control scheme and a sparse-pipeline allocation approach that enables all the stages of an arithmetic pipeline to be processing simultaneously, with multiple computations allocated to the same pipeline. Thus the approach increases hardware resource utilization and reduces power consumption. A framework is presented that implements the concept. The current framework targets a Field-Programmable Gate Array (FPGA), and simplifies the coding phase of the algorithm and troubleshooting. The framework is demonstrated on a technology currently under development by NASA to provide automatic hazard detection and avoidance for spacecraft landing systems.*

Keywords: reconfigurable computing, floating-point, automatic landing, pipeline

1 Introduction

The implementation of computational algorithms is governed by the requirements of the application. In meeting these requirements the designer is also concerned with development time and maintainability of the implementation in the face of changes to the algorithm. Frequent algorithm design changes may occur due to parallel development of the algorithm and its realization.

For 3-dimensional, real-time, dynamic computational applications such as found in space systems, requirements for fast but low power processing often direct the search for implementation options to custom configurable solutions. Field-Programmable Gate Array (FPGA) implementations have been shown to have up to a factor of ten less power consumption compared to microprocessors [1].

This work investigates applying reconfigurable technologies in support of the Automated Landing and Hazard Avoidance Technology (ALHAT) project [2]. ALHAT is developing a system for the automatic detection and avoidance of landing hazards to spacecraft. The system is required to process large amounts of terrain data from a Light Detection and Ranging (LIDAR) sensor, within strict power constraints. Current design environments for configurable hardware development require substantial knowledge and expertise in hardware design, and these are not traditional skills of algorithm designers. Development times for custom hardware solutions are also significantly higher than for a software implementation. These two characteristics can cause projects to prefer software and microprocessor-based solutions despite the performance potential of configurable hardware [3].

The wide dynamic range associated with 3-dimensional transformations in applications such as ALHAT is best suited to floating-point arithmetic, a primary driver of the complexity of configurable hardware design. Operators in High Level Design Languages (HDLs) do not support floating-point arithmetic, as found in most software languages. Many efforts in the field of configurable computing research focus on developing compilers and frameworks to allow the design entry phase to have a similar complexity level as traditional software design [4]. If a framework with the capabilities required is not available, design alternatives include systolic array implementation or the development of a custom processor. In a systolic array, the operations and data path are designed specifically for the desired operation [5], [6]. This approach typically produces the highest performance when compared to other options, but the design is not flexible and changes to the high level algorithm require new iterations of a potentially time-consuming design effort. Often particular computations occur only a small fraction of the time, but the systolic array computational resources are wired for a specific computation. HDL tools will take advantage of if-then-else topology to recognize when in-line fixed point resources can be re-used, but not module based floating-point resources. In addition, real-time dynamic applications that interface with numerous sensor systems are characterized by sparse data arrivals from those systems. In this sparse data environment fully-pipelined designs are not used to their full capability.

Custom soft processors provide more flexibility to algorithm changes, and better accommodate re-use of resources, but have a more substantial initial design effort. These processors may use instructions tailored to the application in order to perform competitively with general purpose Application Specific Integrated Circuit (ASIC) processors [7], [8].

This work addresses the issues of development time for floating-point arithmetic algorithms in configurable hardware and ease of design modification with a framework that is simple in comparison to a software-to-hardware compilation system. The framework enables the definition of dynamically-configured floating-point pipelines in HDL that follow the flow of a software implementation more closely than a systolic array, and is suitable for straightforward translation from an executable software implementation that can be used for verification of the design. The pipelines use a data tag control scheme that avoids the complexity of centralized pipeline control logic. The tag approach allows dynamic configuration of pipeline components on a cycle by cycle basis, and supports traditional fully pipelined data path configurations, and several schemes for re-use of unallocated cycles in sparsely filled pipeline configurations. Re-use of unallocated cycles requires definite knowledge of where those cycles are. In one method of particular interest, resource constraints are addressed with a phase-coherent allocation approach that overloads pipeline stages with multiple operations, without the need for a scheduling algorithm or complicated control logic.

This paper is organized as follows. Section 2 describes research related to the approach described in this work, Section 3 provides details of the dynamically configurable phase-coherent pipeline design, the prototype test application and experimental platform are discussed in Section 4, test results are discussed in Section 5, and a description of plans for future work is provided in Section 6.

2 Related Work

Several examples exist in the literature of research frameworks for the implementation of floating-point computations on configurable hardware. In [9], the Trident compiler for floating-point algorithms written in C is described. The framework represents a substantial development effort with all the functionality of a traditional compiler: parsing, scheduling, and resource allocation. A custom synthesizer that produces Very High Level Design Language (VHDL) code and custom floating-point libraries are also included. The phase-coherent pipeline approach does not require the complex components found in the Trident system, and is suitable for straightforward translation from C code to VHDL defining the pipeline design.

The authors in [10] present a VHDL auto-coder to reduce the development time of floating-point pipelines. A computation is defined in a custom HDL-like pipeline description file, and the code for a single pipeline implementing the computation is produced. The approach requires a user to learn the author's custom HDL, and does not attempt to share resources. A C++ to VHDL generation framework is presented in [11], using an object-oriented approach to VHDL auto-coding of arithmetic operations. All computations are subclasses of a class "operator", and a method of the class "operator" produces VHDL to implement a pipeline for the computation. Again a user must learn the author's syntax to define computations and resource constraints are not addressed.

The resource sharing approach for phase-coherent pipelines has some similarities to those developed for ASIC synthesis algorithms. Hwang et al. [12] define the Data Introduction Interval (DII) as the period in clock cycles between new data arrivals at the input of a pipeline. A DII equal to one represents fully pipelined operations. As discussed in Section 3, a DII greater than one is required for phase-coherent resource sharing. Resource sharing approaches are described in [13] and more recently for FPGAs in [14], where analysis of a data flow graph of the computation and heuristics must be used. The phase-coherent allocation approach is governed by simple algebraic relationships and does not require complex analysis or heuristics. Phase-coherent allocation does not perform dynamic scheduling, and does not require any scoreboarding method [15] or hardware to check for structural or data hazards.

The association of tag values with data for control discussed in Section 3 is similar to the tagged-token dataflow computational model used in the Manchester Dataflow Machine [16]. The local pipeline control in our method is simpler, and does not require a matching unit, token queue, or overflow unit. Tagged-token dataflow concepts have also been used more recently in a configurable hardware implementation for parallel execution of multiple loop iterations [17].

A hand-coded systolic array and MATLAB®-based FPGA implementation of the coordinate conversion stage of processing LIDAR scan data for an automatic landing hazard detection system is compared in [18]. Fixed-point arithmetic is used. As previously discussed, a hand-coded systolic array is not easily adaptable to algorithm design changes. The MATLAB® solution is more easily developed and adapted, but is most suitable to the processing of streaming data and would not be an effective approach for other computations, such as the hazard detection stage of the ALHAT algorithm.

3 Implementation Framework

In this section, the key elements of the phase-coherent, dynamically configurable pipeline framework are described. An example design is used to illustrate how the concepts work together to provide the benefits of the approach.

3.1 Dynamically Configurable Pipeline

In the context of this work, a dynamically configurable pipeline is a pipelined operation with an interconnect configuration that is selectable during circuit operation. This selectable input/output configuration between floating-point operations, temporary storage, and input/output devices is achieved with multiplexers on each port of the pipeline. As stated in [19] the resource overhead of using multiplexers to share computing resources is balanced by the reduction of resources achieved. An additional concern, particularly for reconfigurable computing, is development time [20]. Dynamically configurable pipelines provide a flexible data path that is easier to modify than the fixed data path of a systolic array, reducing overall development and maintenance time.

3.2 Phase Tag Control

A dynamically configurable data path allows the inputs of an operation to be consuming operands and results to be produced for different computations potentially every clock cycle. Further, operations such as floating-point arithmetic typically require latencies greater than one to function at high enough clock frequencies to meet the performance requirements of applications. A control scheme is required to route operands and results between pipelines and other resources with the correct timing. In contrast to using a scheduling algorithm and global control approach, a distributed phase tag control method is used.

In the phase tag control method, a tag word is associated with a set of inputs. The tag is assigned to a buffer that has the same latency as the operation consuming the inputs. The buffer is called a phase keeper. The output of the phase keeper is tested to determine when the outputs of the operation are valid for the inputs associated with the tag. One tag can be used to control the input/output configuration of many floating-point units, providing all units have the same latency. For coding convenience, and to handle occasional units with different latency, phase keepers were built in to all our floating point units. If they are not used, the HDL tool flow removes them automatically. The functional units output two phase tag words. The ready tag (.r), usually with a latency of one clock cycle, is used to signal that a pipeline is ready to accept data. The completion tag (.p) indicates that an operation is finished. The tags are used to control the inputs and outputs of a pipeline. The content of the tag is used by an algorithm developer to indicate which step of the computation is associated with the operands or results. The phase tag control approach supports both dense and sparsely allocated pipelines. Examples of several design cases follow.

Figure 1 is a VHDL code sample of the dynamically configurable pipeline and phase tag control approach. The design is a fully pipelined implementation of the computation A+B+C+D=sum. In this and the remaining code examples, the signal AI(n) is an array of records that bundles data, phase tag, and function (addition or subtraction) for the input of adder unit n. AO(n) is an analogous signal for the output of the unit. The signals PI(n) and PO(n) are arrays of phase tag signals for the input and output, respectively, of phase keeper unit n.

In this example, three adders are connected such that the first unit (0) adds inputs A and B in parallel with the second unit (1) that adds inputs C and D. The outputs of the first two adders are wired to the inputs of the third adder (2), typical of a systolic array. Instead of an explicit state machine to control the output of the pipeline when the result of the computation is valid, the phase tag appearing at the output of the third adder, AO(2).p, is tested to determine when to store the result in registered signal sum1.

```
...
ad0: FADDP port map (clk, AI(0), AO(0));    --adder unit instantiations
ad1: FADDP port map (clk, AI(1), AO(1));
ad2: FADDP port map (clk, AI(2), AO(2));

process (clk) begin
if rising_edge(clk) then
...
 AI(0) <= (ZERO, ZERO, NOPH, ADD);
 AI(1) <= (ZERO, ZERO, NOPH, ADD);
 AI(2) <= (ZERO, ZERO, NOPH, ADD);

 if example1 then                            -- *** static full rate pipe with 3 adders ***
  if data_strobe then                        -- initiate first two adds on data strobe
   AI(0) <= (data_a, data_b, x"10", ADD);
   AI(1) <= (data_c, data_d, x"10", ADD);
  end if;
  AI(2) <= (AO(0).o, AO(1).o, AO(0).p, ADD); -- intermediate sums always wired to final adder
  if AO(2).p = x"10" then
   sum1 <= AO(2).o;                           -- consume result when tag appears at output
  end if;
 end if;

...

end if;
```

Figure 1. Example of dynamically configurable phase-tag control design of a fully pipelined implementation of the computation A+B+C+D=sum.

```
...
process (clk) begin
if rising_edge(clk) then
...
 AI(0) <= (ZERO, ZERO, NOPH, ADD);
 AI(1) <= (ZERO, ZERO, NOPH, ADD);
 AI(2) <= (ZERO, ZERO, NOPH, ADD);

 if example2 then                          -- *** sparse pipe with only one adder and max rate ***
  if data_strobe then
   AI(0) <= (data_a, data_b, x"10", ADD);    -- initiate first add on data strobe
  end if;
  if AO(0).r = x"10" then
   AI(0) <= (data_c, data_d, x"11", ADD);    -- initiate second add as soon as adder ready for input
  end if;
  if AO(0).p = x"10" then
   sum_ab <= AO(0).o;                        -- save sum a+b for one clock cycle
  end if;
  if AO(0).p = x"11" then
   AI(0) <= (sum_ab, AO(0).o, x"21", ADD);  -- initiate final sum
  end if;
  if AO(0).p = x"21" then
   sum2 <= AO(0).o;                          -- consume final sum
  end if;
 end if;
...
end if;
end process;
```

Figure 2. Example of dynamically configurable phase-tag control design of a sparsely pipelined implementation of the computation A+B+C+D=sum.

clock	trigger	logic	adder pipe stage 1	adder pipe stage 2
1	**DATA STROBE**	**A1 & B1 to adder, tag #10**		
2	**ready #10 tag**	**C1 & D1 to adder, tag #11**	**process A1+B1**	
3			**process C1+D1**	**process A1+B1**
4	**done #10 tag**	**save A1+B1**		**process C1+D1**
5	**done #11 tag**	**C+D, A+B to adder, tag #21**		
6	DATA STROBE	A2 & B2 to adder, tag #10	**process final sum**	
7	ready #10 tag	C2 & D2 to adder, tag #11	process A2+B2	**process final sum**
8	**done #21 tag**	**store or forward result**	process C2+D2	process A2+B2

Figure 3. Configuration logic events, event triggers, and processes allocated to adder unit pipeline stages for example of a sparsely pipelined implementation of the computation A+B+C+D=sum.

Figure 2 shows the same computation implemented with only a single adder unit, and assuming a DII that results in sparse data arrival strobes compared to the length of the computation. In this example, the ready tag is tested to determine when the adder unit can accept a second input. This approach allows the single adder to process the first two add operations as quickly as possible without a conflict. Note that intermediate results must be saved for the first addition because the result cannot be consumed immediately. Intermediate results can be stored up to the register resource limit of a particular part.

Using a two stage floating-point adder module, Figure 3 shows the processes allocated to each adder stage, the configuration logic for each clock cycle, and the event which triggers the logic. Data point 1 is shown in bold. A second data point is in gray. The minimum DII is 5 clock cycles, at which time the previous point is finished with the adder input. The pipe is still processing the previous point, but the correct actions will be triggered by tags as they emerge from the pipe, as each carries configuration “knowledge" of what should be done with its associated data.

The phase tag scheme makes call and return logic, similar to subroutine calls in software programs, possible in a configurable hardware design. The re-used adder has effectively become a call and return module. The tag associated with data input indicates where the control logic should resume when the adder is finished.

In addition to tags, an application can use mode variables to control the configuration of the pipeline units. Mode variables can allow use of fewer distinct tags. But the pipe has to be completely empty before changing a mode variable. The examples do not have mode variables, but they were used in our prototype application.

Note that the tags are strobes. Initialization code gives them the default value “NOPH” (no phase) unless they are specifically assigned. If a group of functional units will be re-used in a different module, then this initialization should be contingent upon a mode variable. The functional units are set up with tri-state input signals so that they can be shared between modules. All that is required is to pass the input and output signals for the functional units to the active module, and enable initialization defaults only in the active module. If the DII is not regular, then buffers should be used to ensure that the minimum DII is met.

The example in Figure 2 uses intra-pipeline stage sharing to process more than one computation stage on a single adder unit. Effective resource sharing using this method can be a complicated problem [14]. In the next section, the concept of phase-coherent resource allocation is presented as a straightforward means of accomplishing intra-pipeline sharing. The allocation is constrained by simple relationships based on the DII and minimum pipeline unit latency. These criteria are not restrictive in real-time data processing systems that interface with various sensor subsystems with different latencies. In such systems, fully pipelined computations are not generally required.

3.3 Phase-Coherent Resource Allocation

Phase-coherent pipeline allocation is a simple means to allow pipeline stage sharing that enables different computations to be allocated to the same functional units. The method requires that results associated with a particular computational sequence all emerge at a constant phase, that is, at a constant multiple of a minimum unit latency L. If a unit does not naturally have this latency, it must be padded with enough empty pipeline stages. The multiplex stage is included in the latency value. The DII should be equal to or greater than L[1]. The pipe can be said to have L independent phases. For maximum re-use, successive data inputs are allocated to different phases, until all phases are used, and only then are conflicts possible. Under these conditions, a simple algebraic relationship can be used to compute the period of time that units can be re-used as follows.

Given a dynamically configurable pipelined functional unit with a latency of L, each pipeline stage, $S_p(n)$, can process a datum of an independent computation. The reuse interval, I_R, is defined as the number of clock cycles in which units can be reused freely. This interval is computed as shown in (1).

$$I_R = DII \cdot L \qquad (1)$$

For maximum re-use, the interval can be applied separately to each functional unit. The reuse interval may be applied manually if hand-coding or incorporated into a translator. Figure 4 shows a diagram of phase-coherent pipeline allocation for the computation of the prior code examples. In Figure 4, the DII=3, L=3, and I_R=9. The d_n variables represent a set of input operands for the four add operations at DII=n. As shown, the phase offsets for allocation are implemented with a one or two cycle store of the incoming data value at the input of the unit, as shown in clock cycles 3, 6, and 7. This is required because the input stage of the unit is busy processing prior input data at these cycles. A latency L phase keeper buffer tracks the allocation of available

[1] In real systems, the DII is never regular due to crossing clock domains between the sensor systems and application. First-In First-Out (FIFO) buffering can be used to force an average DII allowing the use of the phase-coherent allocation.

computation phases and is used to control the assignment of inputs to functional units. Also shown in Figure 4 is that by cycle 8 when the third input data set is consumed, each pipeline stage of the unit is processing data. Intermediate results or temporary state variables, for example the intermediate addition result A+B, do not benefit from the phasing scheme and must either be used within the DII or copied every DII clocks. Alternately they could be retained in no-op pipe units.

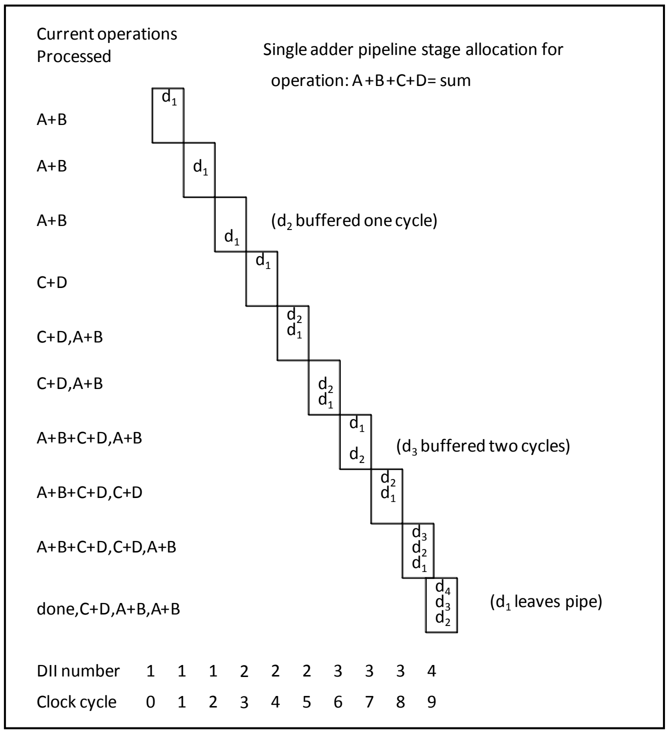

Current operations Processed
Single adder pipeline stage allocation for operation: A+B+C+D= sum
A+B
A+B
A+B
C+D
C+D,A+B
C+D,A+B
A+B+C+D,A+B
A+B+C+D,C+D
A+B+C+D,C+D,A+B
done,C+D,A+B,A+B
d1
d2
d3
d4
(d2 buffered one cycle)
(d3 buffered two cycles)
(d1 leaves pipe)
DII number 1 1 1 2 2 2 3 3 3 4
Clock cycle 0 1 2 3 4 5 6 7 8 9

Figure 4. Phase-coherent allocation of a single adder unit performing the computation A+B+C+D=sum on three input data sets, d. The parameters of the allocation are DII=3, L=3, and I_R=9, all in design clock cycles.

The VHDL that implements the example of Figure 4 is shown in Figure 5. As shown, the code implementing the phase-coherent allocation method is straightforward and suitable for generation by an auto-coder. The phase tags control each stage of the computation as well as the cycle the adder unit is free to accept new data. The dynamically configurable inputs and outputs allow the same unit to process each computation stage within the reuse interval.

4 Prototype Application

Details of the algorithms supporting ALHAT for landing hazard detection and avoidance are provided in [21]. The general approach is to produce a regular grid of surface elevation data in the coordinate frame of the landing site from the LIDAR range samples. This elevation map is then analyzed for surface slope and roughness and compared to thresholds for these parameters to identify hazards. The processing stages for LIDAR scan data are coordinate conversion, re-gridding, and hazard detection. The first two stages are currently demonstrated in the prototype design. The computations implemented are summarized in this section.

```
pk0 : PKEEP  port map (clk, PI(0), PO(0));    --keeper unit instantiations
pk1 : PKEEP  port map (clk, PI(1), PO(1));
...
process (clk) begin
if rising_edge(clk) then
PI(0) <= RDYPH;                                --constant indicating units ready for data
waiting <= FALSE;
...
AI(0) <= (ZERO, ZERO, NOPH, ADD);
AI(1) <= (ZERO, ZERO, NOPH, ADD);
AI(2) <= (ZERO, ZERO, NOPH, ADD);

if example3 then                              -- *** phase coherent pipe with DII*L re-use interval ***
  PI(0) <= PO(0);                             -- default to a re-circulating phase tag
  if data_strobe or waiting then
   if PO(0) = RDYPH then                      -- wait for an available pipe phase
    PI(0) <= x"10";                           -- phase tag indicates 1st operation this re-use interval
    AI(0) <= (data_a, data_b, x"10", ADD);    -- send a and b to adder inputs
   else
    waiting <= TRUE;                          -- if pipe not available, come back and try again
   end if;
  end if;
  if PO(0) = x"10"  then                      -- when a+b completes, save and start next
   PI(0) <= x"20";
   AI(0) <= (data_c, data_d, x"20", ADD);
   sum_ab <= AO(0).o;                         -- this state variable valid  for L clocks or DII/L stages
  end if;
  if PO(0) = x"20"  then                      -- when c+d completes, initiate final sum
   PI(0) <= x"30";
   AI(0) <= (sum_ab, AO(0).o, x"30", ADD);
  end if;
  if PO(0) = x"30"  then                      -- when final sum completes ...
   PI(0) <= RDYPH;                            -- vacate this phase
   sum3 <= AO(0).o;                           -- consume result
  end if;
end if; -- end example3

...
end if;
end process;
```

Figure 5. Example of dynamically configurable phase-tag control design of a phase-coherent pipeline implementation of the computation A+B+C+D=sum.

4.1 Coordinate Conversion and Re-gridding

As described in [21], the coordinate conversion stage converts each LIDAR range sample from scanner angle and range coordinates to Cartesian coordinates. The computation is shown in (2), where $pr_{x,}$ $pr_{y,}$ $pr_{,z}$ are components of the converted point, $t_{x,}$ $t_{y,}$ $t_{,z}$ are the components of the sensor position vector, $p_{x,}$ $p_{y,}$ $p_{,z}$ are the components of the range sample, and $q_{1,}$ $q_{2,}$ $q_{3,}$ q_4 are components of a quaternion vector for the coordinate rotation.

$$pr_{x,y,z} = 2\begin{bmatrix} (q_3q_3 + q_0q_0 - 0.5)p_x + \\ (q_0q_1 + q_3q_2)p_y + \\ (q_0q_2 + q_3q_1)p_z \end{bmatrix} + t_{x,y,z} \tag{2}$$

In the re-gridding stage of the computation, converted range samples are projected into a grid cell of the elevation map, and a bilinear interpolation scheme is used to update the elevation of each vertex of the cell containing the projected point. The elevation of the projected point weighted by the distance from the point to the vertex is added to the current weighted elevation for that vertex. Updates to the weighted elevations and the weights for each vertex of the grid cell containing a projected point are made using the computation shown in (3). In (3) *r* and *c* are the row and column numbers of the elevation map grid cell vertices, respectively.

$$\begin{aligned}
&u = r - \lfloor r \rfloor; \\
&v = c - \lfloor c \rfloor; \\
&W(r,c) += (1-u)(1-v); \\
&E(r,c) += (1-u)(1-v)pr_z; \\
&W(r+1,c) += u(1-v); \\
&E(r+1,c) += u(1-v)pr_z; \\
&W(r,c+1) += (1-u)v; \\
&E(r,c+1) += ((1-u)v)pr_z; \\
&W(r+1,c+1) += uv; \\
&E(r+1,c+1) += (uv)pr_z;
\end{aligned} \quad (3)$$

4.2 Experimental Setup

The prototype design is tested on a Xilinx® Virtex™-5 FX130T FPGA hosted on an Alpha Data ADM-XRC-5TZ mezzanine card. The Virtex™-5 family has a radiation tolerant version, providing a path to space flight certification of the design. The ADM-XRC-5TZ board has 48 megabytes of SRAM across six banks, used for storing the elevation and weight data for the elevation map. The prototype design uses two SRAM interfaces that bundle two SRAM banks each. The interfaces are designed to use one bank for even addresses and one for odd. This approach makes it possible to run the SRAM interface at twice the design rate to reduce memory latency. Currently all interfaces operate at the same clock frequency.

A Gigabit Ethernet interface is included for command and data input and output. The prototype is designed to run at the Ethernet clock speed of 125 MHz. To avoid buffering of the input data, the prototype is designed with a DII of 12 clock cycles. The 12 cycles is derived from each LIDAR sample consisting of three single-precision floating-point components of four bytes each.

With this prototype platform, the coordinate conversion and re-gridding computations were implemented within a few days using the phase-coherent pipeline approach. A full-rate elevation map computation is verified (input LIDAR samples are converted and re-gridded within 12 clock cycles) on this prototype. These results show that the LIDAR data can be processed in real time, or faster than real time.

5 Results

A comparison of resources used between various implementations of the example computation presented in Section 3 is shown in Table I. The resource values are reported from the Xilinx® synthesis tools. The static implementation is a direct wiring of the adders and data path to realize the computation. The dynamically configurable/phase tag control implementations are designed as presented in Figures 1, 2, and 4. The phase-coherent implementations are designs applying the phase-coherent method to each case represented by Figures 1, 2, and 4. The floating-point units are implemented using the Xilinx® CORE Generator™ tool. DSP slice resources are used in the multiplier units but not shown in Table I. Comparing the Lookup Table (LUT) resources between the static and phase-coherent implementations shows the phase-coherent pipeline method yields an 85% reduction in resources. This means a given FPGA can hold the equivalent of about seven times as many source lines of floating point application equivalent code using the phase-coherent method, as using traditional data path methods. If a particular design does not approach resource limits, phase-coherent reuse reduces design size resulting in faster place and route.

TABLE I. RESOURCE COMPARISON BETWEEN DIFFERENT IMPLEMENTATIONS OF COMPUTATION SUM=A+B+C+D.

Resource	**Implementation- 3 Instantiations of sum=a+b+c+d**		
	Static	***Dynamically Configurable/Phase Tag Control***	***Phase-Coherent***
Slice Registers	1198	753	342
LUTs	4768	1755	705
Slices	1653	694	309
LUT/FF Pairs	4706	2010	834
Min. Period	6.6ns	6.6ns	6.6ns

6 Conclusions

The method described in this work achieves substantial improvements in the ease of both development and resource reuse for pipelined computations on configurable hardware. Using this HDL method, declarations and wiring are simplified, and operand/result assignments are easily mixed with other synchronous code. The HDL reads like and corresponds closely to a software specified algorithm. This allowed rapid design of the prototype, and should allow fast response to algorithm changes. The HDL is suitable for straightforward translation from an executable software definition that can be used for algorithm verification. This reduces the gap between the expertise required to design configurable implementations and that of typical algorithm designers.

The difficulty of resource reuse is reduced with a data tag control scheme and phase-coherent allocation method that replace the need for complex global scheduling, heuristics or cycle dependent logic. Sparse data arrival in real-time is efficiently allocated to pipeline stages, reducing design size and place and route times.

Further examination of the utility of the approach is planned with the full implementation of the initial hazard detection algorithms in the ALHAT project [21]. Unlike coordinate conversion and re-gridding, hazard detection is computation-bound with high potential parallelism, exercising the generality of the approach. The algorithm has been updated several times since this initial version [3] providing a relevant case to test the difficulty of design modification using the phase-coherent framework. The approach will also be considered for application to dynamically configurable ASICs.

7 References

[1] G. Govindu, L. Zhuo, S. Choi, P. Gundala, V. Prasanna, “Area, and Power Performance Analysis of a Floating-point based Application on FPGAs”, In Proceedings of the Seventh Annual Workshop on High Performance Embedded Computing (HPEC 2003), http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.157.9977.

[2] C. Epp, E. Robertson, T. Brady, "Autonomous Landing and Hazard Avoidance Technology (ALHAT)," Aerospace Conference, 2008 IEEE,pp.1-7,March. 2008,doi: 10.1109/AERO.2008.4526297.

[3] C. Villalpando, A. Johnson, R. Some, J. Oberlin, S. Goldberg, "Investigation of the Tilera processor for real time hazard detection and avoidance on the Altair Lunar Lander," Aerospace Conference, 2010 IEEE, pp.1-9, March, 2010, doi: 10.1109/AERO.2010.5447023.

[4] M. Cardoso, P. Diniz, and M. Weinhardt, “Compiling for reconfigurable computing: A survey,” ACM Comput. Surv. 42, 4, Article 13, June 2010, pp. 1-65,doi:10.1145/1749603.1749604.

[5] K. Sano, T. Iizuka, S. Yamamoto, "Systolic architecture for computational fluid dynamics on FPGAs," Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on, pp.107-116, April 2007,doi: 10.1109/FCCM.2007.20.

[6] S. Qasim, S. Abbasi, B. Almashary, "A proposed FPGA-based parallel architecture for matrix multiplication," Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on, pp.1763-1766, Nov. 2008, doi: 10.1109/APCCAS.2008.4746382.

[7] D. Goodwin and D. Petkov, “Automatic generation of application specific processors,” Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '03). ACM, New York, NY, USA, pp. 137-147, doi:10.1145/951710.951730.

[8] J. Yu, C. Eagleston, C. Han-Yu Chou, M. Perreault, and G. Lemieux, “Vector processing as a soft processor accelerator,” ACM Trans. Reconfigurable Technol. Syst. 2, 2, Article 12, June 2009, pp. 1-34, doi:10.1145/1534916.1534922.

[9] J. Tripp, K. Peterson, C. Ahrens, D. Poznanovic, M. Gokhale, "Trident: an FPGA compiler framework for floating-point algorithms," Field Programmable Logic and Applications, 2005. International Conference on, pp. 317-322, Aug. 2005, doi: 10.1109/FPL.2005.1515741.

[10] G. Lienhart, A. Kugel, R. Manner, "Rapid development of high performance floating-point pipelines for scientific simulation," Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, pp.8 April, 2006, doi: 10.1109/IPDPS.2006.1639439.

[11] F. de Dinechin, C. Klein, B. Pasca, "Generating high-performance custom floating-point pipelines," Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on , pp.59-64, Sept. 2009, doi: 10.1109/FPL.2009.5272553.

[12] K. S. Hwang, A. E. Casavant, C. Chang, M. d'Abreu, "Scheduling and hardware sharing in pipelined data paths," Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on, pp.24-27, Nov. 1989, doi: 10.1109/ICCAD.1989.76897.

[13] S. Wakabayashi, N. Ohashi, J. Miyao, N. Yoshida, "A synthesis algorithm for pipelined data paths with conditional module sharing," Circuits and Systems, 1992. ISCAS '92. Proceedings., 1992 IEEE International Symposium on, vol.2, pp.677-680, May 1992, doi: 10.1109/ISCAS.1992.230161.

[14] S. Mondal, S. Memik, "Resource sharing in pipelined CDFG synthesis," Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific, vol.2, pp. 795- 798, Jan. 2005, doi: 10.1109/ASPDAC.2005.1466464.

[15] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco: Morgan Kaufmann Publishers, 2003.

[16] J. R Gurd, C. C Kirkham, and I. Watson, “The Manchester Prototype Dataflow Computer”, Commun. ACM 28, 1, January 1985, pp.34-52, DOI=10.1145/2465.2468.

[17] H. Styles, D.B. Thomas, W. Luk, "Pipelining Designs With Loop-Carried Dependencies," Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on, pp. 255- 262, 6-8 Dec.2004doi: 10.1109/FPT.2004.1393276.

[18] K. Shih, et al., “Fast real-time LIDAR processing on FPGAs,” http://www.informatik.uni-trier.de/~ley/db/conf/ersa/ersa2008.html (accessed 6/3/2011).

[19] W. Sun, M. Wirthlin, S. Neuendorffer, "FPGA pipeline synthesis design exploration using module selection and resource sharing," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,vol.26,no.2,pp.254-265,Feb.2007,doi: 10.1109/TCAD.2006.887923.

[20] J. Villarreal, A. Park, W. Najjar, R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, pp.127-134, May, 2010, doi: 10.1109/FCCM.2010.28.

[21] A. Johnson, A. Klumpp, J. Collier, A. Wolf, “Lidar-based hazard avoidance for safe landing on Mars,” http://trs-new.jpl.nasa.gov/dspace/ (accessed 6/3/2011).

NAC: A lightweight intermediate representation for ASIP compilers

Nikolaos Kavvadias and Kostas Masselos
Department of Computer Science and Technology, University of Peloponnese, 22100 Tripoli, Greece

Abstract— *ASIP processors are tuned for optimized mapping of narrow application sets in heterogeneous platforms. Their successful development relies on compiler-based design space exploration. The careful design of the compiler intermediate language is a necessity, due to its dual purpose as both the program representation and an abstract target machine. Its design affects the complexity, efficiency and ease of maintenance of all compilation phases.*

In this work, an extensible typed assembly intermediate language, NAC, is presented. It can be used for processor exploration, optimizing intermediate representation (IR) transformations and SSA compilation. Minimal SSA construction algorithms are thoroughly presented for the first time. [1]

Keywords: compilers, intermediate representation, SSA, ASIP

1. Introduction and related work

Recent compilation frameworks provide linear IRs for applying analyses, optimizations and as input for backend code generation. GCC [1] supports the GIMPLE IR. Many GCC optimizations have been rewritten for GIMPLE, but it is still undergoing grammar and interface changes. GCC supports backends for ASIP processors such as baseline Xtensa [2] but it is not suitable for rapid retargeting to non-trivial architectures. LLVM [3] uses a register-based IR named LLVM bitcode, targeted by a C/C++ companion frontend (clang). It is written in better coding style than GCC, but similarly the IR infrastructure and semantics are excessive.

In this paper, the NAC (N-Address Code) IR is introduced. NAC supports semantic-free n-input/m-output mappings, user-defined data types, and specifies a virtual machine architecture. NAC's strength is its simplicity: it is inherently easy to develop a CDFG (Control/Data Flow Graph) extraction API, apply graph-based IR transformations for domain specialization, investigate SSA (Static Single Assignment) construction algorithms and perform other compilation tasks for ASIPs.

Specifically, this paper investigates minimal SSA construction schemes [4], [5] that don't require the computation of the iterated dominance frontier [6]. For the first time, detailed implementations are illustrated to ease their adoption in new projects.

[1]The presented research work was co-funded by the European Union in the frame of the ENOSYS project (FP7-ICT-248821) (`www.enosys-project.eu`).

2. Representing programs in NAC

In this section, the NAC typed-assembly language is described. NAC provides arbitrary n-to-m mappings allowing the elimination of implicit side-effects, a single construct for all operations, and bit-accurate data types. It supports scalar, single-dimensional array and streamed I/O procedure arguments. NAC statements are labels, n-address instructions or procedure calls.

An n-address instruction is actually the specification of a mapping from a set of n ordered inputs to a set of m ordered outputs. Also termed as (n, m)-operation, it is formatted as follows:

```
outp1, ..., outpm <= op inp1, ..., inpn;
```

where: `op` is a mnemonic referring to an IR instruction, `inp1, ..., inpn` are its n inputs, and `outp1, ..., outpm` its m outputs.

All declared objects have an explicit static type specification: "globalvar" (a global scalar or array variable), "localvar" (a scalar or array local), "in" (an input argument to the given procedure), or "out" (an output argument).

NAC supports bit-accurate data types for (signed/unsigned) integer/fixed-point and floating-point arithmetic. Data type specifications are essentially strings that can be easily decoded by a regular expression scanner; typical examples are `u32, s11, q4.4u, q2.14s, f1.8.23`, respectively.

The EBNF grammar for NAC is shown in Fig. 1 where it can be seen that rules "nac" and "pcall" provide the means for the n-to-m generic mapping for operations and procedure calls, respectively. It is important to note that NAC has no predefined operator set; operators are defined through a textual mnemonic.

For instance, an addition of two scalar operands is written as: `a <= add b, c;`. Control-transfer operations include conditional and unconditional jumps explicitly visible in the IR. An example of an unconditional jump would be: `BB5 <= jmpun;` while conditional jumps always declare both targets: `BB1, BB2 <= jmpeq i, 10;`. This statement enables a control transfer to the entry of basic block BB1 when i equals to 10, otherwise to BB2.

Procedures are supported as non-atomic operations by using a similar form to operations. In `(y) <= sqrt(x);` the square root of an operand `x` is computed; procedure argument lists are indicated as enclosed in parentheses.

```
nac_top = {gvar_def} {proc_def}.
gvar_def = "globalvar" anum decl_item_lst ";".
proc_def = "procedure" [anum] "(" [arg_lst] ")"
                "{" [{lvar_decl}] [{stmt}] "}".
stmt = nac | pcall | id ":".
nac = [id_lst "<="] anum [id_lst] ";".
pcall = ["(" id_lst ")" "<="] anum ["(" id_lst ")"] ";".
id_lst = id {"," id}.
decl_item_lst = decl_item {"," decl_item}.
decl_item = (anum | uninitarr | initarr).
arg_lst = arg_decl {"," arg_decl}.
arg_decl = ("in" | "out") anum (anum | uninitarr).
lvar_decl = "localvar" anum decl_item_lst ";".
initarr = anum "[" id "]" "=" "{" numer {"," numer} "}".
uninitarr = anum "[" [id] "]".
anum = (letter | "_") {letter | digit}.
id = anum | (["-"] (integer | fxpnum)).
```

Fig. 1: EBNF grammar for NAC (N-Address Code).

Table 1: A set of basic operations for a NAC-based IR.

Mnemonic	Description	(N_i, N_o)
`ldc`	Load constant	(1,1)
`neg, mov`	Unary arithmetic op.	(1,1)
`add, sub, abs, min, max, mul, div, mod, shl, shr`	Binary arithmetic op.	(2,1)
`not, and, ior, xor`	Logical	(2,1)
`setzz`	Comparison for `zz`: (`eq,ne,lt,le,gt,ge`)	(2,1)
`muxzz`	Conditional selection	(3,1)
`load, store`	Load/Store from/to mem.	(2,1)
`sxt, zxt, trunc`	Type conversion	(1,1)
`jmpun`	Unconditional jump	(0,1)
`jmpzz`	Conditional jump	(2,2)

2.1 Encoding NAC information

A NAC program incorporates the complete information of a translation unit of the original program comprising of a “globalvar” definition list and a procedure list. A single NAC procedure is defined by the following set of lists: ordered input (output) arguments, “localvar” definitions, NAC statements and basic block (BB) labels.

Statements are organized in the form of a record. Lists *opnds_in* and *opnds_out* collect operand items, following the definition of an “OperandItem” record. This record is comprised of an identifier name, a data type specification, an operand type (*otype*) representation and an absolute operand item index. *otype* can take one of the following values: {INARG, OUTARG, LOCALVAR, GLOBALVAR, CONSTANT} and {INVAR, OUTVAR} as an additional def/use specifier for NAC statements.

3. Uses and extensions of NAC

3.1 A basic NAC implementation

A basic operation set for RISC compilation is summarized in Table 1. N_i (N_o) denotes the number of input (output) operands for each operation.

The memory access model defines dedicated address spaces per array, so that both loads and stores require the array identifier as an explicit operand. For an indexed load in C (`b = a[i];`), a frontend would generate the following NAC: `b <= load a, i;`, while for an indexed store (`a[i] = b;`) it is `a <= store b, i;`.

Table 2: CI characteristics for hand-optimized ANSI C implementations of *crcsp* and *crcdp*.

GA operator	Bit-level oper.	N_i/N_o	Cycles (seq.)	CI cycles	CI area (MAU)
crcsp	No/Yes	4/1	76-13	–	–
crcsp	No/Yes	8/1	41-6	3-1	0.977-0.142
crcsp	No/Yes	8/2	5-1	3-1	1.867-0.153
crcdp	No/Yes	4/1	111-18	–	–
crcdp	No/Yes	8/1	58-8	3-1	1.466-0.147
crcdp	No/Yes	8/2	5-1	3-1	2.800-0.164

3.2 IR extensions

We have defined three custom IR operators, `bitins`, `bitext` and `concat`, for bitfield insertion/extraction from a word and concatenation of two or more subwords. As motivational examples, the single- (*crcsp*) and double-point (*crcdp*) crossover operators are examined. C code for the genetic algorithm operators was passed to Machine-SUIF [7] for IR generation using a peephole matching-based code selection pass for the ByoRISC ASIP [8]. ByoRISC supports CIs with up to 8 inputs and 8 outputs.

crcsp reads four inputs: two parent chromosomes (father, mother), crossover point (location), and chromosome length (len) and produces two independent outputs; the (son, daughter) chromosomes for the next generation. *crcdp* defines two crossover points for bitfield exchange.

In Table 2, with bit-level operators unused, the minimum number of cycles required for *crcsp* are 76 for a sequential schedule and 12 for an ASAP, while for the *crcdp* these limits are 111 and 14, respectively. When the bit-level operators are used, the sequential schedules without CIs require 13 and 18 cycles for *crcsp* and *crcdp* respectively with ASAP schedules of 5 cycles for both. When the $N_i/N_o = \{8/2\}$ constraint is used, a single-cycle multi-input, multi-output (MIMO) CI is identified for each crossover operator. The area requirement is estimated relatively to the area (multiplier area unit or MAU) of a 32-bit single-cycle multiplier characterized for a Virtex-4 FPGA (XC4VLX25).

3.3 CDFG construction

A novel, fast CDFG construction algorithm has been devised for both SSA and non-SSA NAC forms producing flat CDFGs (Fig. 2). A CDFG symbol table item is a node (operation, procedure call, globalvar, or constant) or edge (localvar) with user-defined attributes: the unique name, label and data type specification; node and edge type enumeration; respective order of incoming or outgoing edges; input/output argument order of a node and BB index. Further attributes can be defined, e.g. for scheduling bookkeeping.

3.4 Application profiling with NACVM

NAC programs can be either interpreted or translated to low-level C for performance evaluation on the corresponding

```
NACtoCDFG()
  input List NACs, List variables, List labels, Graph cfg;
  output SymbolTable st, Graph cdfg;
begin
  Insert constant, input/output arguments and global
  variable operand nodes to st;
  Insert operation nodes;
  Insert incoming {global/constant/input, operation} and
  outgoing {operation, global/output} edges;
  Add control-dependence edges among operation nodes;
  Add data-dependence edges among operation nodes,
  extract loop-carried dependencies via cfg reachability;
  Generate cdfg from st;
end
```

Fig. 2: CDFG construction algorithm accepting NAC input.

Table 3: Application profiling with a NAC framework.

App.	LOC (NAC)	LOC (dot)	$P/V/E$	#ϕs	#Instr.
atsort	155	484	2/136/336	10	6907
coins	105	509	2/121/376	10	405726
cordic	56	178	1/57/115	7	256335
easter	47	111	1/46/59	2	3082
fixsqrt	32	87	1/29/52	6	833900
perfect	31	65	1/23/36	4	6590739
sieve	82	199	2/64/123	12	515687
xorshift	26	80	1/29/45	0	2000

abstract machine, NACVM. A set of realistic kernels has been selected: *atsort* (an all topological sorts algorithm by Knuth), *coins* (compute change with minimum amount of coins), multimode *cordic* computation, *easter* (Easter date calculations), *fixsqrt* (fixed-point sqrt), *perfect* (perfect number detection), *sieve* (prime sieve of Eratosthenes) and *xorshift* (100 calls to G. Marsaglia's PRNG).

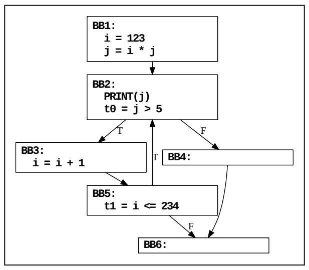
Static and dynamic metrics have been collected in Table 3. For each application (App.), the lines of NAC and resulting CDFGs are given in columns 2-3, number of CDFGs (P: procedures), vertices and edges (for each procedure) in column 4, amount of ϕ statements (column 5) and lastly the number of dynamic instructions for the non-SSA case.

4. SSA construction algorithms

This paper argues that rapid prototyping compilers, would benefit from straightforward SSA construction schemes which don't require the use of sophisticated concepts and data structures [4], [5].

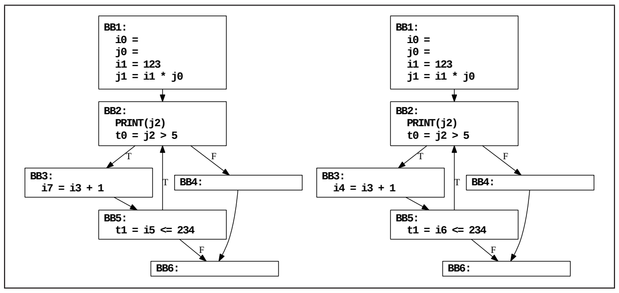
Algorithm P presents a "really-crude" approach for variable renaming and ϕ-function insertion [4]. In the first phase, every variable is split at BB boundaries, while in the second phase ϕ-functions are placed for each variable in each BB. Variable versions are actually preassigned in constant time and reflect a specific BB ordering (e.g. DFS). Thus, variable versioning starts from a positive integer n, equal to the number of BBs in the given CFG.

Algorithm H does not predetermine variable versions at control-flow joins but accounts ϕs the same way as actual computations visible in the original CFG. Due to this fact, ϕ-insertion also presents dissimilarities. Both methods share common ϕ-minimization and dead code elimination phases.



```
BB1:
  i = 123
  j = i * j
BB2:
  PRINT(j)
  t0 = j > 5
BB3:
  i = i + 1
BB4:
BB5:
  t1 = i <= 234
BB6:
```

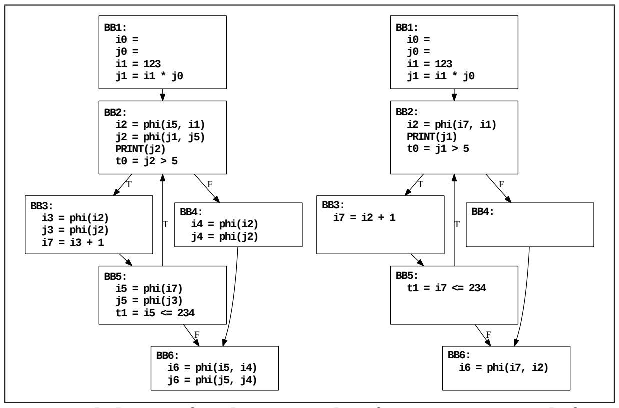
Fig. 3: CFG of the example subprogram from [5].



```
BB1:
  i0 =
  j0 =
  i1 = 123
  j1 = i1 * j0
BB2:
  PRINT(j2)
  t0 = j2 > 5
BB3:
  i7 = i3 + 1
BB4:
BB5:
  t1 = i5 <= 234
BB6:
```

```
BB1:
  i0 =
  j0 =
  i1 = 123
  j1 = i1 * j0
BB2:
  PRINT(j2)
  t0 = j2 > 5
BB3:
  i4 = i3 + 1
BB4:
BB5:
  t1 = i6 <= 234
BB6:
```

Fig. 4: Incomplete SSA for the example following variable numbering with algorithm P (left) and H (right).



```
BB1:
  i0 =
  j0 =
  i1 = 123
  j1 = i1 * j0
BB2:
  i2 = phi(i5, i1)
  j2 = phi(j1, j5)
  PRINT(j2)
  t0 = j2 > 5
BB3:
  i3 = phi(i2)
  j3 = phi(j2)
  i7 = i3 + 1
BB4:
  i4 = phi(i2)
  j4 = phi(j2)
BB5:
  i5 = phi(i7)
  j5 = phi(j3)
  t1 = i5 <= 234
BB6:
  i6 = phi(i5, i4)
  j6 = phi(j5, j4)
```

```
BB1:
  i0 =
  j0 =
  i1 = 123
  j1 = i1 * j0
BB2:
  i2 = phi(i7, i1)
  PRINT(j1)
  t0 = j1 > 5
BB3:
  i7 = i2 + 1
BB4:
BB5:
  t1 = i7 <= 234
BB6:
  i6 = phi(i7, i2)
```

Fig. 5: Valid SSA for the example after ϕ-insertion (left) and ϕ-minimization and dead code elimination (right).

4.1 Motivating example

The motivating example from [5] is shown in Fig. 3, with incomplete SSA following variable numbering in Fig. 4.

Valid unoptimized and minimal SSA are shown in Fig. 5 involving the maximum and minimum possible number of ϕs, respectively, as generated by P. H presents only lexicographic and not semantic differences to this result. Both algorithms achieve the generation of minimal SSA involving the two ϕ statements in BB2 and BB6.

4.2 Analysis of algorithms P and H

Variable numbering in algorithm P is given in Fig. 6. Only arrays and maps (key-indexed) are used for sequences of same-type elements. Vectorized assignments to arrays/maps are allowed, copying a scalar to all elements. A single iterative form is used for iterating over a set or sequence. Lists can have subset updates, member insertions and deletions

```
VariableNumbering(List NACs, List vars):
  ssa_vars = empty; var_reads = zeroes;
  var_writes = ones; set_writes = 0;
  curr_bb = 0; prev_bb = -1;
  bbnum = get number of basic blocks from NACs;
  for stmt in NACs do
    if stmt.bb != curr_bb then
      prev_bb = curr_bb; curr_bb = stmt.bbix;
      if curr_bb > 1 and set_writes == 0 then
        var_writes = bbnum; set_writes = 1;
      var_reads = curr_bb;
    for input operand (opnd) in stmt do
      if opnd is a localvar and is scalar then
        ssaopnd = opnd ## var_reads['opnd'];
      update input operands of stmt;
    for output operand (opnd) in stmt do
      get opnd_ix = index of opnd in vars;
      if opnd is a localvar and is scalar then
        if stmt.bb > 1 then
          var_writes['opnd'] += 1;
        var_reads['opnd'] = var_writes['opnd'];
        ssaopnd = opnd ## var_writes['opnd'];
        insert ssaopnd to ssa_vars list;
      update output operands of stmt;
    update stmt in NACs;
  delete localvar scalars from vars;
  merge ssa_vars with vars;
```

Fig. 6: Variable numbering in algorithm P.

and can be merged. A key-based retrieval operation named *get* is also used. GNU C concatenation ## is used.

P alters in-place the NAC statement list and replaces the non-SSA variable list by a versioned one, *ssa_vars*. *var_-reads* and *var_writes* define maps for keeping the version numbers of CFG variables. *set_writes* is a flag array for controlling proper initialization of *var_writes*. *curr_bb* and *prev_bb* are BB markers for the current and previous BB accessible in a single pass over NAC statements. *bbnum* is the number of BBs in the CFG. For each NAC, *var_writes* is set to *bbnum* for all except the entry BB. *var_reads* is set to *curr_bb*. Then, for each NAC input operand, which is a local scalar, a versioned variable is created using the entry in *var_reads*. For output operands, *var_writes* is incremented for a non-entry BB and the *var_reads* entry for the same operand is updated for future uses. A new SSA variable is defined according to the value of *var_writes* entry and is inserted in *ssa_vars*. After updating each NAC accordingly, local scalars are deleted from the initial list and *ssa_vars* is merged with *vars*.

Variable numbering in algorithm H uses a 'visited' map, *var_bb_id*. After some preprocessing, input operands are numbered in the exact same way as in P. Output operands are associated to defined SSA variables: for unvisited variables of entry blocks, *var_writes* is incremented by two, otherwise by one. Then unvisited variables are marked as visited. Afterwards, *var_reads* is updated as in P.

ϕ-insertion according to P reads both the non-SSA and SSA variable lists as shown in Fig. 7. ϕ statements are collected in *phi_stmts*. *bb_preds* is the list of all preceding BBs for a given block and *bb_preds_num* keeps their number. All BBs are scanned to update *bb_preds* and *bb_preds_num*, then for each non-SSA variable active in the given BB (k), the ϕ statement destination operand is created as the k+1

```
PhiInsertion(List NACs, List vars, List labels,
  List nonssa_vars):
  phi_stmts = empty; bb_preds = zeroes; bb_preds_num = 0;
  (ST, G) = create CFG from (NACs, labels);
  for k in BBs(ST) do
    insert predecessor BBs of k in bb_preds;
    bb_preds_num = get number of predecessor BBs of k;
    for sopnd in nonssa_vars do
      if sopnd is localvar scalar, has def/use in k then
        phi_opnds_in = empty; phi_opnds_out = empty;
        if bb_preds_num > 1 then
          ssaopnd_out = sopnd ## k+1;
          insert ssaopnd_out to phi_opnds_out;
          insert ssaopnd_out to vars;
        ix = 0;
        for n in bb_preds_num do
          if bb_preds[n] != -1 then
            ix = SSA ver of sopnd at last def in BB #n;
            if ix == 0 then
              ix = bb_preds[n] + 1;
            ssaopnd_in = sopnd ## ix;
            insert ssaopnd_in to phi_opnds_in;
        if k == 0 and BB #k does not define sopnd then
          phi_stmt = LOADCONST(phi_opnds_out);
        elsif BB #k has predecessors then
          phi_stmt = PHI(phi_opnds_out, phi_opnds_in);
        insert phi_stmt to phi_stmts;
  merge NACs with phi_stmts;
  update absolute addresses (addr) in NACs, labels;
```

Fig. 7: ϕ-insertion in algorithm P.

version. Determining input SSA operands for each NAC requires scanning all predecessors, and if any exist, to assign the version *bb_preds*[n]+1. Then, either a constant load or a ϕ statement is created, the latter for non-entry BBs.

ϕ-insertion in H examines all parsed BBs for determining subsequent variable versions for each ϕ output operand. If a def of this operand is found, its SSA version is incremented by two over the current index, otherwise by one. Source operand version is defined by a similar process, without the additional version increment.

5. Conclusions

In this paper, a semantic-free IR, named NAC, was presented, for use in rapid prototyping ASIP compilers. Its applicability is illustrated through cases of rule-based transformation for better CI generation, application profiling and self-contained description of minimal SSA construction algorithms.

References

[1] GCC. [Online]. Available: http://gcc.gnu.org

[2] Tensilica. [Online]. Available: http://www.tensilica.com

[3] LLVM. [Online]. Available: http://llvm.org

[4] A. W. Appel, "SSA is functional programming," *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, Apr 1998.

[5] J. Aycock and N. Horspool, "Simple generation of static single assignment form," in *Proc. 9th Int. Conf. in Compiler Construction*, 2000, pp. 110–125.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Prog. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct 1991.

[7] Machine-SUIF. [Online]. Available: http://www.eecs.harvard.edu/hube/software/

[8] N. Kavvadias and S. Nikolaidis, "The ByoRISC configurable processor family," in *Proc. IFIP/IEEE VLSI-SoC*, Oct. 2008, pp. 439–444.

Automatic Tailoring of Configurable Vector Processors for Scientific Computations

D. Rutishauser[1] **and M. Jones**[2]
[1]Avionic Systems Division, NASA Johnson Space Center, Houston, Texas, U.S.A.
[2]Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia, U.S.A.

Abstract—*Re-hosting legacy codes optimized for a platform such as a vector supercomputer often requires a complete re-write of the original code. This work provides a framework and approach to use configurable computing resources in place of a vector supercomputer towards the implementation of a legacy code without a long and expensive re-hosting effort. The approach automatically tailors a parameterized, configurable vector processor design to an input problem, and produces an instance of the processor. Experimental data shows the processors perform competitively when compared with a diverse set of contemporary high performance computing alternatives.*

Keywords: reconfigurable, high-performance vector, scientific computing

1. Introduction

A legacy code application tailored for execution on a vector computer is the assumed target for acceleration for this work. A custom vector computer is developed on an Field-Programmable Gate Array (FPGA) to run the application, and this computer is customized to the particular application. Configurable processing resources enable a wider range of vector processing architectures than found in traditional vector computers, and allow an architecture to be tailored to a specific application, instead of the traditional approach of tailoring the application to the computer.

This paper highlights contributions of the work described in detail in [1]: (1) a formulation of the problem of determining a tailoring of an architecture to a computation, (2) an algorithmic approach to solve the problem that includes a parameterized architectural framework for vector processing, (3) a scheduling/mapping algorithm that effectively uses established performance-enhancing practices in vector computing, and (4) the VectCore processor design that provides a low-overhead implementation of the approach. The approach is evaluated using experimental data including data produced from an end-to-end implementation in hardware.

2. Related Work

The following is an abbreviated survey of related work, a more complete discussion can be found in [1]. The viability of supercomputing on FPGA systems has been assessed [2]. The potential for performance gains has resulted in numerous studies ranging from implementations of basic matrix computations [3], to the porting of a full scientific application to a production supercomputing system with configurable hardware capabilities [4]. Hybrid general purpose, configurable High Performance Computer (HPC) systems have also been developed as commercial and research systems [5],[6].

Vector processing remains a relevant processing paradigm in the domain of scientific computation. The Convey hybrid HPC [6] includes a vector processing functionality as an example of its application-specific "personalities." The Vector Instruction Set Architecture (ISA) Processors for Embedded Reconfigurable Systems (VIPERS) [7] is a "soft" vector processor approach that provides a configurable feature set that can be tailored to an application. The Vector-Extended Soft Processor Architecture (VESPA) [8] features a vector co-processor also with a configurable architecture that can be tailored to input problem requirements. These systems do not include an automatic tailoring of the vector processing resources to a specific problem.

3. Approach

The system targeted by this approach is a hybrid consisting of an FPGA tightly-coupled with a general-purpose processor. Computations identified as candidates for FPGA implementation are input to tools that determine an architecture tailored to each input problem. A mapping of each input computation to its tailored architecture is also produced. Custom tools produce the High-Level Design Language (HDL) representation of the tailored architecture, and a microcode program to run on the architecture. Vendor tools produce the FPGA configuration file. All the custom and vendor tools run on a development workstation. A software program consisting of the original application augmented with code that provides an interface between the software application and the FPGA runs on the general-purpose processor.

Details of the problem formulation can be found in [1]. An overview of the algorithmic approach is shown in Figure 1. The inputs are a representation, G, of a set of computations

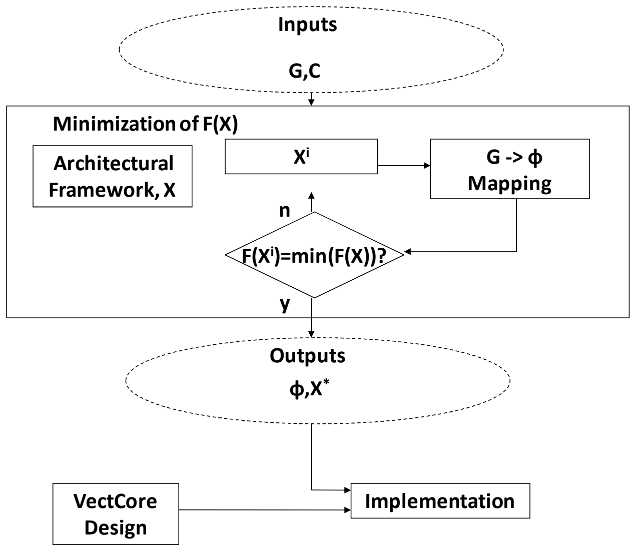


Fig. 1: Algorithmic approach overview.

and an overall resource constraint, C, for the FPGA. An integer minimization algorithm is executed for an objective function F. The minimization algorithm includes an architectural framework that is the target for mapping the input computation to processors and supporting resources. The template is parameterized allowing different specifications for the quantity and type of processing resources. A vector X specifies the parameters for a given architecture. A scheduling and allocation algorithm maps operations in G to a set of starting times, ϕ, and resources in a particular instance, X^i of the architectural template. The outputs of the minimization algorithm are the architecture specification, X^*, that minimizes the scheduled execution time of G without exceeding the resource constraints, and the associated schedule, ϕ. The VectCore processor design provides an interface to the general-purpose processor and controls the execution of the resources specified by X^*. An integrated implementation consisting of the tailored VectCore instance and a microcode program to run the input computation completes the solution.

3.1 Architectural Template

The architectural template was introduced in [9] and is a design framework for a configurable vector processing core. The template provides a target for the allocation of input computations and supports performance enhancing vector computing techniques. The following notation is used for the components included in the VectCore implementation designed for this research.

- L = vector load/store units
- V = vector registers
- A = vector adder units
- M = vector multiplier units
- B = functional unit buses

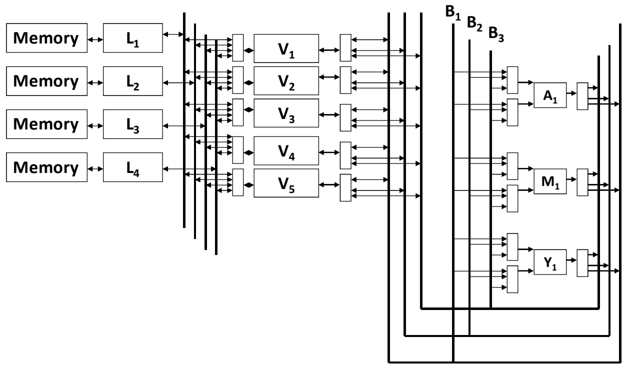


Fig. 2: Example VectCore architecture template instance. Specification includes $4L$, $5V$, $1A$, $1M$, $3B$, $1Y$, and $0I$.

- Y = vector $SAXPY$ units
- I = vector inner product units

Other floating-point functional units could be defined in the approach. Figure 2 shows a specific example of a template instance. In the VectCore topology, U is the total number of vector functional units and B denotes the number of independent links between input/output pairs in the network. The number of inputs, q, to the VectCore network is $q = V + U$. Analysis in [1] shows the wiring complexity for the VectCore is proportional to Bq. Therefore, the VectCore interconnect network performance and cost can be approximately matched to topologies ranging from a bus $(B = 1)$ to a crossbar $(B = q)$ depending on the selection of the parameter B. The specific instance of the template is found by solving the minimization problem for a given target computation constrained by the available FPGA resources.

3.2 VectCore Design

The VectCore microcode is a representation of the schedule and allocation determined by the minimization algorithm. The format for a microcode word is the VectCore Schedule-Packet (S-PAK). An S-PAK contains a start time and resource configuration for a schedule event. The size of each S-PAK word is fixed for a given maximum number of resources and number of resource types [1]. The S-PAK design minimizes overhead with a compact and flexible format. For example, the size of an S-PAK is low because the architecture does not require complex routing information or unique bit fields for each resource.

Figure 3 shows a block diagram of the interface and S-PAK dispatch architecture. S-PAKs are written by the general purpose processor to the S-PAK First In First Out (FIFO) buffer interface of the VectCore. The S-PAK router forwards S-PAKs to resource FIFO buffers for each computing resource. When all the S-PAK configurations for a given schedule event have been forwarded, a control word is written to the event FIFO for consumption by the global clock

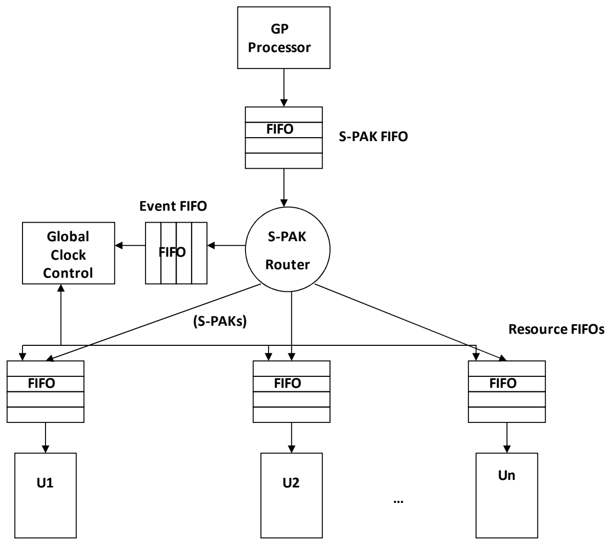


Fig. 3: VectCore interface and S-PAK dispatch scheme.

control logic. This logic sends a start signal to independent state machines for each resource. Proper execution timing is ensured by including a capability for all the resource pipelines to be stalled until all necessary resources are ready to support a given global clock cycle. Comparisons of the schedule length to actual schedule execution time measured by hardware counters show this approach adds a very low (3-4%) overhead to the schedule execution time.

4. Experimental Design

The VectCore approach is evaluated with a fully operational end-to-end implementation using a Xilinx®Virtex II Pro™vp70 FPGA. The VectCore design is also targeted to a Virtex™5 part. The VectCore is tested with the matrix multiplication and back-substitution step of the LU matrix decomposition problems, and a portion of an application tailored to a Cray architecture. Matrix-by-matrix multiplication provides a computation-bound test case with high available parallelism. The back-substitution problem is characterized by a long dependency chain between the operations. The Cray benchmark also has a high available parallelism and is memory-bound. A range of problem sizes, operation orderings, and architecture resource sizes are tested to characterize the scaling and degree of tailoring possible with the VectCore approach [1].

5. Results

Table 1 shows the Floating-Point Operations Per Second (FLOPS) performance for each tailored problem implementation running each problem type. The first three rows are matrix multiplication problems with different operation orderings. The next two rows are two different implementations of the back-substitution problem, and the problem labeled “tass" is the Cray application. For the $mm1$, $mm2$, and $tass$ problems, the maximum FLOPS performance occurs for the implementations running the problem that matches its tailoring. As expected, the long dependency chains and low available parallelism in the back-substitution problems show that the performance of the architecture is limited in comparison to problems such as matrix-by-matrix multiplication. Nevertheless, VectCore is able to effectively and correctly execute such problems, an important requirement for many applications.

Table 1: VectCore (Virtex II Pro™)performance of each tailored problem implementation for each benchmark problem.

Problem	**Architecture Tailoring Target**					
	MFLOPS					
	$mm1$	$mm2$	$mm3$	$ts1$	$ts2$	$tass$
$mm1$	1989	0	0	243	0	0
$mm2$	0	1155	496	0	0	0
$mm3$	0	267	400	0	0	0
$ts1$	0	0	0	164	0	0
$ts2$	0	108	108	0	108	0
$tass$	0	0	0	149	0	763

The VectCore performance is compared to alternatives including hybrid general-purpose/configurable processor HPC systems, a systolic architecture, a server-grade General Purpose Processor (GPP) system, and a traditional supercomputer. For additional architecture comparisons see [1]. Table 2 shows the GFLOPS per processor performance and the system cost per GFLOPS for the particular workload types and sizes for each alternative. [1]

The Virtex™2 Pro VectCore exceeds the performance of the Cray SV1, and a VectCore substitution for this type of system is the original motivation for this research. The VectCore provides this performance at approximately 17 times lower price per GFLOPS than the SV1. The VectCore targeted to the Virtex™5 yields a better GFLOPS performance than previous generation hybrid HPCs, and an AMD GPP running optimized code.

Systolic implementations and HPCs outperform the VectCore on similar FPGA device families. Systolic implementations require a dedicated design effort, and a different computation can easily accrue a similar design effort. An HPC's supporting resources for the FPGA cores are fixed. Optimization to a new FPGA target may require substantial redesign for either approach. The VectCore approach is general for a class of computations, and can be re-targeted easily to newer FPGA families to realize large performance gains without a specific optimization to the device.

The overhead of the VectCore approach lies in layering a vector architecture on the existing architecture of an FPGA. The amount of FPGA resources not used directly for floating-point computations impacts the FLOPS performance. The components that dominate the overhead are the

[1]The system prices are reported at initial release, in U.S. dollars, because the literature supporting each non-VectCore option spans several years.

Table 2: VectCore performance and price per performance comparison to implementation alternatives.

System	**Workload**	**Size**	**Clock (MHz)**	**FPGA**	**Device**	**GFLOPS per proc.**	**System Price**	**Price per GFLOPS**	**FP GFLOPS Limit**	**% GFLOPS Limit**
Hybrid HPC										
Convey HC-1 [6]	Matrix Mult.	order 16K	150	Virtex™5	LX 330	19.0	$13K	$171	14.4	132
Cray XD-1 [10]	Matrix Mult.	order 16K	110	Virtex™2 Pro	vp 50	2.0	$100K [5]	$8.3K	8.8	23
Direct Hardware Implementation										
Systolic Array [11]	2D Cavity Flow	48 X 48 grid	106	Stratix®2	EP 2S180	18.0	$7.7K [12]	$425	20.4	88
Configurable Vector Processor										
VectCore	Matrix Mult.	order 16K	133	Virtex™2 Pro	vp70	1.3	$7.5K	$5.8K	14.9	9
VectCore	Matrix Mult.	order 16K	350	Virtex™5	LX 330	6.5	$7.5K	$1.2K	86.8	7
GPP										
AMD (ACML) [10]	Matrix Mult.	order 2K	2200			3.9	$500	$128		
Legacy Vector Supercomputer										
Cray SV1 [13]	Matrix Mult.		300			1.0	$375K [14]	$100K		

vector control for each functional unit and the functional unit bus interconnect. To characterize the performance impact of this overhead, the FLOPS performance using the maximum number of basic floating-point operations (the operation type is problem-dependent) supported by the resources of a given FPGA is used as an upper bound. The last two columns of Table 2 show this maximum theoretical performance and the percent of this maximum achieved for the configurable options. The VectCore approach achieves the lowest percentage of the theoretical performance, but the analysis does not use optimized functional unit designs for a particular FPGA architecture. Lower per-unit resource usage and higher operating frequencies are possible with device-specific optimized designs [15]. Optimized designs can be incorporated into the existing VectCore framework to improve performance and reduce the approach overhead. In addition, the VectCore interconnect is a straightforward, non-optimized design, and its resource usage estimates are pessimistic compared to what is possible using optimizations such as partially-connected subnetworks.

6. Concluding Remarks

The VectCore is compared with a diverse set of contemporary high performance computing alternatives. Problem-specific VectCore implementations are shown to exhibit higher FLOPS performance than several alternatives when targeted to recent FPGA technologies. The VectCore approach is more flexible to design changes than the systolic or HPC implementations that exhibit higher FLOPS performance. This flexibility balances the utility of the VectCore approach with its inherent overhead, which is also characterized in this work.

References

[1] D. Rutishauser, "Implementing Scientific Simulation Codes Tailored for Vector Architectures Using Custom Configurable Computing Machines," Ph.D. dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, U.S.A., 2010. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-04132011-174232/

[2] S. Craven and P. Athanas, "Examining the Viability of FPGA Supercomputing," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 13 –13, 2007.

[3] S. Qasim, S. Abbasi, and B. Almashary, "A Proposed FPGA-Based Parallel Architecture for Matrix Multiplication," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, 30 2008.

[4] V. Kindratenko and D. Pointer, "A Case Study in Porting a Production Scientific Supercomputing Application to a Reconfigurable Computer," Apr. 2006, pp. 13 –22.

[5] internetnews.com, "Cray Unleashes XD1 Opteron/Linux Supercomputer," http://www.internetnews.com/ent-news/article.php/3417221/Cray-Unleashes-XD1-OpteronLinux-Supercomputer.htm, (accessed Feb. 5, 2011).

[6] J. Bakos, "High-Performance Heterogeneous Computing with the Convey HC-1," *Computing in Science Engineering*, vol. 12, no. 6, pp. 80 –87, 2010.

[7] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, "Vector Processing as a Soft Processor Accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 12:1–12:34, June 2009. [Online]. Available: http://doi.acm.org/10.1145/1534916.1534922

[8] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors," in *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2008, pp. 61 –70.

[9] D. Rutishauser. (2006) Implementing Scientific Simulation Codes Highly Tailored For Vector Architectures Using Custom Configurable Computing Machines. MAPLD International Conference. Washington, DC, U.S.A. (accessed Jun. 16, 2011). [Online]. Available: http://klabs.org/mapld06/

[10] L. Zhuo and V. Prasanna, "High-Performance Designs for Linear Algebra Operations on Reconfigurable Hardware," *Computers, IEEE Transactions on*, vol. 57, no. 8, pp. 1057 –1071, Aug. 2008.

[11] K. Sano, L. Wang, Y. Hatsuda, and S. Yamamoto, "Scalable FPGA-Array for High-Performance and Power-Efficient Computation Based on Difference Schemes," in *High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on*, 2008, pp. 1 –9.

[12] The Dini Group, "Hardware for ASIC Prototyping & FPGA Systems," http://www.dinigroup.com/pages/3/files/2006-03-30_7000K10PCI_press_release.pdf, (accessed Mar. 13, 2011).

[13] Cray Inc., "The Benchmarkers̓ Guide for CRAY SV1 Systems," http://parallel.ksu.ru/ftp/computers/cray/sv1_bmguide.pdf, (accessed Mar. 13, 2011).

[14] IDC Inc., "The Cray CX1 Supercomputer: Leveraging the Cray Brand in the HPC Workgroup Market," http://www.cray.com/Assets/PDF/products/cx1/IDC%20whitepaper-CrayCX1.pdf, (accessed Mar. 14, 2011).

[15] K. S. Hemmert and K. D. Underwood, "Fast, Efficient Floating-Point Adders and Multipliers for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, pp. 11:1–11:30, September 2010. [Online]. Available: http://doi.acm.org/10.1145/1839480.1839481

Optimizing the Costs of Communication Infrastructure in Message-Based Multicore

Lars Middendorf[1], **Christian Haubelt**[2], **and Christophe Bobda**[3]
[1]Institute of Computer Science, University of Potsdam, Potsdam, Germany
[2]Institute of Applied Microelectronics and Computer Engineering, University of Rostock, Rostock, Germany
[3]College of Engineering, University of Arkansas, Fayetteville, Arkansas, USA

Abstract—*A software routing approach is proposed as a means to reduce the cost of communication infrastructure in multicore based on message passing. Those costs are generally incurred by the redundancy of routers, designed to deal with all kind of configurations. Our approach consists of eliminating the hardware routers and moving the message routing tasks to general purpose processing elements. For computationally intensive applications, the ratio between computation and routing can be adjusted dynamically. With the use of complex examples, we show that our solution reduces the resource consumption of the communication infrastructure significantly, with only a marginal decrease of the overall performance.*

Keywords: A maximum of 6 keywords

1. Introduction

While the increasing demand on computational power is being fulfilled with the increasing number of processors in multicore systems, the problem of communication between the individual processing units is still an issue. With the increasing amount of cores per chip, message passing approaches are likely to become the dominant paradigm due to the memory bottleneck related with SMP-based approaches.

Network on Chip (NoC) was introduced as an attempt to provide the communication support in message-based multicore [1]. A NoC consists of a set of routers attached to the computing element, with the goal of transporting a message to destination without the intervention of the processor. NoCs are crucial for the performance as well as the cost of the whole system [2]. Most of the systems designed so far rely on a general structure, usually a mesh, that will allow all kinds of connections in the network. This approach is highly inefficient since it does not match the topology of the executing application and incurres high resource costs. The routers are usually designed to deal with all possible cases, thus remaining inefficient for the running applications. Figure 1 shows a 3x3 mesh implementation of a NoC with three processing elements (TC, VGA, LV) as it appears in many system. Clearly the amount of space consumed by the routers ((1,1) - (3,3)) is more than 90% of the total area, not being worth the design efforts. It is therefore not surprising that NoCs have so far not found their way in commercial devices, despite the broad adoption of the multicores in the industry.

The *dynamic network-on-chip* (DyNoc) introduces the concept of coarse-grained components implemented on top of a grid of reconfigurable processing elements [3] [4] [5]. The main innovation of the DyNoc is the possibility to adapt the communication infrastructure to the running application at run-time. Unused routers can therefore be dynamically assigned to processing elements in order to increase their computation power at run-time leading to optimized topologies for the running applications. The DyNoC however remained a concept for which only a case study was implemented in FPGA, but there have been also other approaches to employ unused router components in a DyNoC.

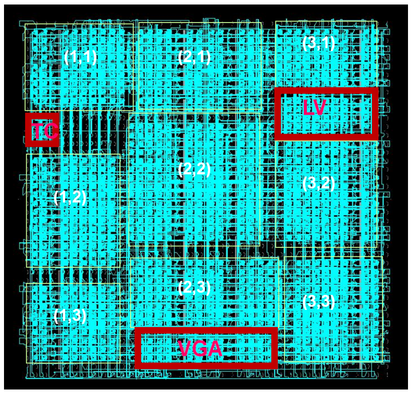


Fig. 1: A 3x3 network on chip for graphic processing

The traffic of an application usually depends on the input data, so that it varies during runtime and does not distribute evenly on the chip. As a result, only a fraction of the total on-chip bandwidth is used. Therefore, a fixed partitioning between computational and communication elements always leads to a waste of resources. The approach provided in this paper offers a pragmatic way to implement dynamic NoCs.

2. Software-Assisted Routing

Software-assisted routing takes place completely inside the processing element. Since the hardware routers are omitted, the processing elements are connected directly to the surrounding nodes. Each processor does not only receive its own packets, but also becomes responsible for routing packets that are specified for other nodes. Hence, the processor must be able to suspend the execution of the main program at any time to handle incoming packets. Thus, an interrupt line is connected to all input ports, so that the software router can be started immediately. Upon receiving a request to route a packet through an interrupt, the processor begins with an arbitration of the input channels. In the simplest case, a round-robin scheduling is used to select the correct input port. The header of the packet is then analyzed to determine the target location.

In comparison to a hardware router, which is able to forward a packet of *n* words in *n* cycles, the software router is several times slower. Most of the delay is caused by the loop that copies a packet between ports. On RISC architectures like the Xilinx's MicroBlaze, this loop takes at least five cycles per word to copy. We therefore propose to extend the processor with a command to transfer data between ports or memory. It should be executed in parallel to the normal control flow to reduce the effective time for copying a packet. Moreover, the time spent in the interrupt handler depends less on the packet's size. However, the decision of the target channel is still completely done in software, so that the performance of a hardware solution and the flexibility of software routing are combined. Since an interrupt can occur at almost any instruction, the current state of the processor must be saved and restored in the interrupt handler. Duplicating the register file is a well-know technique to reduce this overhead [6]. Both the application and the interrupt handler can have their own set of registers. A switch between both banks is performed instantaneously at the entry and exit of an interrupt.

3. Results

We have implemented two types of mesh-based NoCs (4x4 and 5x5) with XY routing (hardware-based and software-assisted) for the sake of performance comparison. First, the RTL code of the design has been simulated to measure the overhead of the software-based solution and to analyze the traffic within the network (Table 1). The first test case simple outputs an internal buffer into the main memory to measure the maximum bandwidth. The second test produces a checkerboard pattern on the screen. The following applications use ray-tracing to draw two scenes with increasing complexity. While the copy example is 30% faster, the more expensive ray-tracing test achieves a speed-up of only 5%. For more complete and realistic examples, we can expect the difference to decrease even more.

In addition to the tests using a simulator, the design has been also evaluated on a Virtex-6-LXT240T. Table 2 compares the implementation properties of the hardware router and a simple processing core to the extended processing unit with ISA extensions used for software routing. It can be seen that the extended PE occupies less resources than the combination of hardware router and simple PE. Further, we can conclude for the non-trivial tests that the software advantage of area outweighs the hardware advantage of performance from Table 1.

Table 1: Cycles required by hardware and software routing.

Test	HW	SW	HW Advantage
4x4			
Copy	25,938	33,614	1.30
Checkerboard	25,482	33,660	1.32
Plane	61,543	66,805	1.09
Plane, 3 Spheres	88,768	92,831	1.05
5x5			
Copy	26,800	33,743	1.26
Checkerboard	26,444	33,986	1.29
Plane	50,270	53,830	1.07
Plane, 3 Spheres	66,459	68,464	1.03

Table 2: Required hardware resources on Virtex-6 LX240T.

Property	Router & Simple PE (HW)	Extended PE (SW)	SW Advantage
Registers	1,380	1,232	1.12
Lookup Tables	3,768	2,747	1.37

4. Conclusion

The experiments within the simulator and on the FPGA have shown that software routing might be a viable solution for a certain class of applications. However, omitting the routers must save enough resources or provide extra flexibility to justify the overhead of the software solution. Therefore, this technique might offer its advantages on large arrays of relatively simple processing elements like RISC cores. As a result, we expect software-assisted routing to be most useful for resource constrained and dynamic reconfigurable platforms like FPGAs.

References

[1] L. Benini and G. De Micheli, "Networks on chips: A new soc paradigm," *Computer*, vol. 35, pp. 70–78, January 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=619071.621885

[2] U. Y. Ogras, J. Hu, and R. Marculescu, "Key research problems in noc design: a holistic perspective," in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '05. New York, NY, USA: ACM, 2005, pp. 69–74. [Online]. Available: http://doi.acm.org/10.1145/1084834.1084856

[3] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, and J. Teich, "A dynamic noc approach for communication in reconfigurable devices," in *IN PROCEEDINGS OF INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL.* Springer, 2004, pp. 1032–1036.

[4] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[5] C. Mahr, P. Bobda, "Reconfigurable router for dynamic networks-on-chip," in *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on*, 2010.

[6] H.-G. Kim and H.-C. Oh, "A dsp-enhanced 32-bit embedded microprocessor," *J. Embedded Comput.*, vol. 3, pp. 19–28, January 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1516712.1516715

A Reconfigurable Computing Multiple Processor Framework with Hybrid Pipeline Scheduling

D. Cook[1], K. Ferens (Contact), and B. Mcleod
[1]Space Engineering, Bristol Aerospace; Electrical and Computer Engineering, University of Manitoba
[1]darcy.cook@magellan.aero, ferens@ee.umanitoba.ca, mcleod@ee.umanitoba.ca

Abstract- This paper presents a framework for a reconfigurable computing system, consisting of cohesive hardware and software architectures. The framework allows customization of the hardware and software to fit a class of applications, while narrowing the design space to a manageable set of design parameters. The framework features a novel hybrid static and dynamic pipeline scheduling technique, which dynamically reconfigures the hardware to boost performance.

Keywords: Reconfigurable computing, hybrid pipeline scheduling, task scheduling.

I. INTRODUCTION

PERFORMANCE requirements of embedded systems are becoming more demanding. For example, biologically inspired and artificially intelligent computational algorithms have been applied to topological self-organization, routing, and data aggregation for mobile ad-hoc wireless sensor networks. The distributed implementation of these computational algorithms on the underlying hardware of each node of such networks requires, at times, near real-time performance to mitigate the already high rate of disruptions and long delays [1]. Furthermore, the underlying hardware is constrained to consume minimal energy, since grid power may not be available, and power harvesting can only be performed at scheduled times. As a result, the choice of hardware platforms for embedded systems is constrained to implementations which are dynamically reconfigurable and energy consumption aware.

II. FRAMEWORK OVERVIEW

The proposed reconfigurable computing, multiple-processor framework consists of both hardware and software architectures, which may be individually configured to fit a specific application [2].

A. *Hardware Architecture*

The hardware architecture consists of multiple-processors connected together with a uniformly accessible global shared memory. Each processor in the system can access the global memory space to share data, and all processors have an equal average global memory access time. The number of processors can be changed, statically or dynamically, according to the needs of the application. The main components of the system are the soft processors (including local data and instruction memories), global shared memory, global memory controller, and task controller.

Soft Processor

A soft processor is implemented in configurable logic within an FPGA. The framework allows both homogenous and heterogeneous processors, offering the option of creating processors with the same or different hardware and software resources. Each of the processors requires its own local data and instruction memory, which allows each processor to run an independent program.

Task Controller

The task controller performs scheduling and can be customized for each application. The task controller is implemented entirely in hardware, and is independent of the soft processors. This allows the task switching overhead to be much less than it would be if the scheduling was done in software by a processor. The DFG task dependencies are entered as parameters in the task controller hardware description language. Also, the task allocation for each processor is specified in the hardware description of the task controller. This allows the task controller to control which task should be executed on which processor.

Hybrid Pipeline Scheduling

This paper presents a novel pipeline scheduling algorithm that is a hybrid between traditional static pipeline scheduling and dynamic pipeline scheduling [1]. Static pipeline scheduling involves allocating the tasks to the processors at design time, and also determining the schedule in which the tasks will be executed. Dynamic pipeline scheduling involves dynamically assigning tasks to available processors during run-time.

The novel pipeline scheduling algorithm developed for the proposed framework reduces the idle times significantly by changing the number of pipeline stages to ensure that, if there are any tasks assigned to a processor that can be executed, they are executed, rather than waiting for other tasks to finish. There is no longer a global processing period (Tp); now each processor will have its own processing period, that is independent of the other processors.

```
// assign 1st task which has not completed all its iterations
pipe_idx ← min. pipeline index i where $Q[\,N_L[i]\,] < K_{ds}$
```

```
// while there are still tasks (not finished their iterations)
while Q[ N[i] ] < $K_{ds}$ for any task i that is in $N_L$ {
   /* if the predecessors to the current task have more iterations complete than the current task, then execute this task, otherwise skip over it to the next task*/
   if Q[$N_L$[pipe_idx]] < Q[ $M_L$[N[pipe_idx]][j] ] for all j then
   {
      Allow task $N_L$[pipe_idx] to be executed
      Wait until task $N_L$[pipe_idx] is finished executing }
   /*move to the next task, and if the end of the list is reached, then start again at the beginning*/
   pipe_idx ← pipe_idx + 1
   if pipe_idx = $\beta_{np}$ then {
      pipe_idx ← min. pipeline index i where Q[ $N_L$[i] ] < $K_{DS}$ }
}
```

The hybrid pipeline scheduling algorithm adapts to the variable length of task E as shown in Fig. 1. This essentially increases the number of pipeline stages for the system, but results in a shorter total execution time for the entire system.

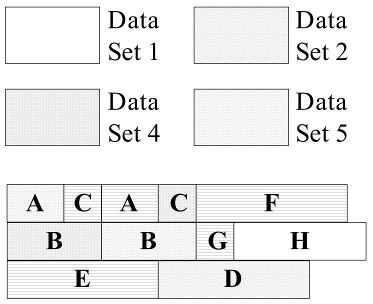


Fig. 1. Dynamic pipeline scheduling with variable task time.

III. EXPERIMENTAL FRAMEWORK

In order to demonstrate the effectiveness of the proposed framework, the framework was implemented in a Xilinx Virtex-II Pro FPGA, on the Xilinx XUPV2P development platform [3]. Xilinx MicroBlaze soft processors were used for the experimental implementation, each processor running with a clock speed of 100 MHz. Each processor had 16 kB of local memory to be used for instruction and data memory. The global memory size used for the example application was 16 kB of RAM internal to the FPGA. The task controller used the novel task pipelining algorithm as described above to dynamically change the length of the pipeline and optimize processor utilization with the goal of minimizing the program execution time. Experiments were conducted using 1, 2, 3, and 4 processors (due to FPGA limitations).

The example application developed to evaluate the proposed framework simulated a green screen video system.

IV. EXPERIMENTAL RESULTS

Overall, it was observed that adding more processors to the system slightly increases the speedup because the program execution is dominated by accesses to global memory. Fig. 2 shows the execution times of each of the tasks with a different number of processors in the system.

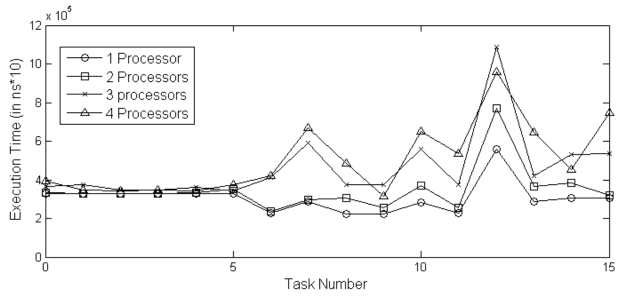


Fig. 2. Original Application Task Execution Time.

V. CONCLUSIONS

This paper proposed a reconfigurable computing multiple-processor (RCMP) framework, which specifically targeted stream-oriented applications. Reconfigurable components: he task allocation, pipelining algorithm, features and resources of a soft processor (local memory, clock speed arithmetic and/or floating point unit), number of soft processors, task controller interface peripheral; memory controller type (synchronous or asynchronous); global memory type and size; and detailed design of the memory controller interface peripheral.

This framework is best suited to stream-oriented problems that are computationally intensive but have common data that is required by most tasks.

ACKNOWLEDGEMENTS

We acknowledge the support of CMC for the initial use of the Xilinx workstations. We would also like to acknowledge the support of the Real-time Embedded laboratory for the use of the Xilinx XUPV2P development platform.

REFERENCES

[1] S. Cui and K. Ferens, "Energy Efficient Clustering Algorithms for Wireless Sensor Networks" in Proc. of ICWN'11 - The 2011 International Conference on Wireless Networks, Las Vegas, NV, 2011, July 18-21.

[2] D. Cook, "Development of a Field Programmable Multiprocessing System Framework for Stream-Oriented Applications", COMP 7850 *Course Project Report*, Winnipeg, MB: Dept. of Electrical and Computer Engineering, University of Manitoba, May 2009.

[3] Xilinx Inc., Xilinx University Program Virtex-II Pro Development System – Hardware Reference Manual. UG069 V1.1, Apr 2008.