Exploiting Instruction Level Parallelism for REPLICA – A Configurable VLIW Architecture with Chained Functional Units

Martin Keßler[1], Erik Hansson[1], Daniel Åkesson[1], and Christoph Kessler[1]
[1]Dept. of Computer and Information Science, Linköping University, Sweden

Abstract— *In this paper we present a scheduling algorithm for VLIW architectures with* chained functional units. *We show how our algorithm can help speed up programs at the instruction level, for an architecture called REPLICA, a configurable emulated shared memory (CESM) architecture whose computation model is based on the PRAM model. Since our LLVM based compiler is parameterizable in the number of different functional units, read and write ports to register file etc. we can generate code for different REPLICA architectures that have different functional unit configurations. We show for a set of different configurations how our implementation can produce high quality code; and we argue that the high parametrization of the compiler makes it, together with the simulator, useful for hardware/software co-design.*

Keywords: Scheduling, VLIW, Compiler, LLVM, instruction-level parallelism

1. Introduction

The REPLICA architecture, which is currently under development, is a chip multiprocessor with configurable emulated shared memory (CESM) architecture [1]. Its computation model is based on the PRAM (Parallel Random Access Machine) model [2]. The PRAM model gives a simple deterministic synchronous and predictable model of programming, where parallelism is homogeneous and explicit. The REPLICA core architecture is a VLIW architecture that supports chained functional units so that the result of one VLIW sub-instruction can be used as input to another sub-instruction in the same (PRAM) execution step. The architecture has support for parallel multi-(prefix)operations on the hardware level [3]. It enables the user to access the same memory location from a multitude of parallel threads. There is no need for explicit locking as it would be in conventional parallel programming. By running a high number of threads in parallel, latency at memory accesses is effectively hidden by the architecture. Shared (physically distributed across on-chip memory modules) memory is in PRAM mode accessed in a UMA (Uniform Memory Access) fashion as if it was local memory.

REPLICA also has support for NUMA mode execution, for being able to run sequential and parallel legacy (NUMA) programs.

At the moment a high-level programming language for REPLICA is under development and should be transformed to the REPLICA baseline language using a source-to-source transformer which is currently under development using ANTLRv3 and written in Scala [1].

The baseline language is based on C with some built-in variables to support parallelism, at the moment these are for example `_thread_id`, `_number_of_threads` and `_private_space_start`.

Programs written in the baseline language can be compiled using Clang to LLVM IR and then compiled to REPLICA target code and tested and evaluated on the REPLICA simulator. The key feature of the compiler is the parametrization of the scheduling algorithm. In an earlier version of the compiler [4] there was only support for one basic architecture configuration.

The focus in this paper is to show how the compiler actually utilizes instruction level parallelism for different configurations of the REPLICA architecture in which we have different combinations of chained functional units.

As a proof of concept we have written, in the baseline language, some test programs such as thread parallel blur and threshold filter as well as sequential programs such as multiply & move, discrete wavelet and inverse discrete cosine transformation. We compiled them for different configurations using our parametrized back-end. The compiled programs are run on the simulator. They all show speed-ups from instruction level parallelism.

The rest of the paper is organized as follows. We introduce the REPLICA assembly programming model in section 2. We give an overview of how the dependency graphs of basic blocks are created in section 3, and in section 4 a method of reducing register use is shown. In section 5 we present our scheduling algorithm, and how the compiler is parametrized is shown in section 6. We compare our scheduling algorithm to previous work in section 8. Results can be found in section 7. Finally the conclusion and future work are in section 9.

2. REPLICA assembly programming model

We refer to a single chained VLIW word as a *line*. Sub-instructions on the same line are to be issued at the same time. In contrast to traditional VLIW architectures, the sub-instructions can be dependent due to chained functional units. Different types of sub-instructions are executed in different functional units, in REPLICA we distinguish between the following sub-instruction types[4]:

- Memory unit sub-instructions: Load, store and multi-prefix instructions.
- ALU sub-instructions: add and subtract etc.
- Compare unit: Compare sub-instructions which, set status register flags.
- Sequencer: branch instructions, jump etc.
- Operand: sub-instruction for loading constants, labels etc. into an operand slot
- Writeback: sub-instruction for copying register contents.

In Table 1 different configurations are shown. The simplest configuration, T5, has one ALU before the memory unit and then a compare unit and a sequencer after each other.

When programming on assembly level for the architecture, one has to distinguish between two types of register storage for intermediate results:

- *general purpose registers* `R1` to `R30` can store values persistently;
- *output buffers* `O0` to `Ox` [1], `A0` to `Ax`, `M0` to `Mx` are transient and only valid inside the line.

Both types, however, can be used in the same way:

```
ADD0 R1,R2
ADD1 O2,A0
```

Listing 1: Using registers and output buffers as operands

As shown in Listing 1, every functional unit is bound to one output buffer, denoted by the number following the instruction mnemonic. Only one dedicated functional unit type, the write-back stage, can write values to the register file.

By using the output buffers inside the super-instruction different operations can be chained and the output of one functional unit is fed as an input to another one. This saves a lot of cycles compared to a way where every intermediate result first would have to be written back to the register file. A slight drawback is that results can only be used from the left to the right. Memory operations are an exception. They cannot be chained as latency hiding would not work any longer.

[1]Here, `x` denotes the number of functional units of that type minus 1. For example the T7 configuration has 3 ALUs, and thus the ALU output buffers are named `A0` to `A2`. The Oi buffers are for result of OPerand instr., the Ai of ALU instr. and the Mi of memory unit instr., respectively

```
OP0 16     ADD0 O0,R1 LD0 A0     WB1 A0     WB2 M0
OP0 2      MUL0 O0,R2 ST0 A0,R0
```

Listing 2: REPLICA assembly example [4]

The example in Listing 2 computes at line 1 an address by adding 16 to `R1` using the first ALU, `ADD0`, the result will be available in `A0` and used to load a word. The result (available in `M0`) is copied to `R2` using the writeback instruction `WB2`, at the same time `A0` is copied to `R1`. In the second line (next execution step) `R2` is multiplied with 2; the intermediate result is in `A0` and will be stored at the address contained in `R0`.

3. Dependency Graph

In both this version and the earlier version [4] of the compiler, the support of VLIW is implemented by matching a set of pre-defined combinations of sub-instructions which we call super-instructions. Of course the instruction scheduling and register allocation will not be optimal. We try to solve this problem by splitting each super-instruction into its respective sub-instructions. User-written inline assembly code is also split up. After splitting, any dependency graph between the instructions previously built is no longer valid. We therefore need to build a new one, suited to the requirements given by both the register compression (see section 4) and the ILP scheduling pass (see section 5).

A new graph is constructed by adding one instruction after another in the existing order and determining dependencies to all previous instructions. Two classes are used to store and structure the acquired information, as shown in Listing 3.

```
class MINode {
public:
  MachineBasicBlock::iterator I;
  list<MIEdge> edges;
  unsigned predecessors;
  bool scheduled;
  // [...]
};

class MIEdge {
public:
  int latency;
  MINode* to;
  // [...]
};
```

Listing 3: Definition of the `MIEdge` and `MINode` class

The first one, `class MINode`, is used to encapsulate the `MachineBasicBlock::iterator` to one instruction. Furthermore it stores, among other attributes used later on, a list of edges to other nodes which depend on this

Name	operands	pre ALU	memory	post ALU	compare unit	sequencer unit
T5	2	1	1	0	1	1
T7	3	2	1	1	1	1
T11	7	5	1	2	1	1
T14	7	5	4	2	1	1

Table 1: list of available standard configurations

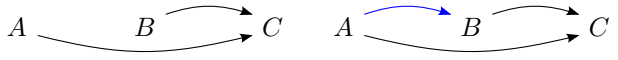
instruction. These edges are formed by instances of the `class MIEdge`.

The attribute `latency`[2] of an edge represents whether the two instructions:

- must stay within the same super-instruction, e.g. `OP0 42` $\xrightarrow{=0}$ `LD0 O0`: The operand `O0` is only valid within a line.
- the depending one *has* to be scheduled in a later super-instruction, e.g. `TRAP R0` $\xrightarrow{>0}$ `ADD0 R1,R2`: We cannot mess with the order before and after a trap.
- or it does not matter ($\xrightarrow{\geq 0}$).

The information what was previously part of one super-instruction is only important to such a degree as to determine how long transient registers are alive. Between the instruction defining the output buffer and the one using it will be an edge with latency constraint $= 0$.

In order to simplify the scheduling pass later on, additional edges are inserted between instructions that are not directly depending on each other.



(a) before (b) after

Fig. 1: Additional edges

This is only necessary for instructions that have to go into the same super-instruction. As shown in Figure 1(b), an additional edge is inserted between nodes A and B. This is required because node A would otherwise only force C into the same super-instruction and leave out B as the edges only provide a forward reference.

Compared to the dependency graph construction in the previous version of the compiler, the methods determining the dependencies have been optimized to report fewer unnecessary conflicts which results directly in fewer edges between the nodes. This in turn enables the scheduling algorithm later on to change the order of the instructions more freely and give a better result.

[2]Latency k of a data dependence edge *(i,j)* is usually defined as if $i \xrightarrow{\geq k} j$ then $t_j \geq t_i + k$ must hold for a correct schedule.

4. Register Compression

The number of slots provided for each functional unit type is limited per super-instruction. Therefore it is quite self explanatory that replacing instructions with shorter versions that do the same job. renders the resulting code more compact. As a result scheduling can pack the same code in fewer super-instructions and thereby speed up its execution. Up to now, the following substitutions are implemented:

- If the constant number 0 is required, use the register `R0` (which is there for that very purpose) instead of defining an operand with `OP 0` and then using the operand. Hence one operand slot is saved and can be used for something else. This is applicable whenever a variable is initialized or reset to zero.
- If an addition at which one operand is a constant zero (`R0`) is performed, the operation can be omitted and instead of the result the non-zero operand can be used. The same holds true for subtractions where the subtrahend is constant zero. This saves one slot in the ALU.
- With the last substitution we might end up with a situation where a register is written back to itself. Such a constellation can of course be removed entirely.
- If a constant zero (`R0`) is written back to a general purpose register, all upcoming usages of that register can be replaced with `R0` up to the next redefinition of that register.

To enforce the compiler to use register compression a flag, `-enable-replica-register-compression`, can be used. While at this point the compiler and simulator still operate under the assumption that we have unlimited write-back slots, this will not hold true in future versions. At that point saving `WB` operations will pay off.

5. ILP scheduling

5.1 Motivation

As REPLICA is a VLIW architecture, the compiler should identify operations that can be executed in parallel when there are no dependencies and resource conflicts between them, and put them in one super-instruction. Additionally the REPLICA architecture offers the possibility to chain instructions. That means that we can

use the output of one functional unit inside a super-instruction as an input to the next one. Thereby we don't have to wait until the result is written back to the register file.

When the LLVM intermediate representation is lowered to the REPLICA instruction set, the created VLIW super-instructions do represent the correct semantics, but they utilize the available functionality quite poorly. An addition operation, for example, will only make use of one ALU and one write-back slot in a super-instruction. All other slots are idling at that time. To make better usage of both the available ILP and the possibility to chain instructions, we have provided an optimization pass that reschedules the instructions, at basic block level, into new super-instructions.

Another important advantage is that this also helps us to generate code for different REPLICA configurations, i.e. providing different amounts of available slots per functional unit type.

Before we start with a description of the scheduling algorithm, some naming conventions that we will use should be introduced. The scheduler is written in a way that we support an arbitrary number of slots per functional unit type. The available slots per functional unit type as a whole will be referred to as a *bucket* (operand bucket, pre-memory-ALU bucket, memory bucket, ...). A *chain* of instructions is a list of instructions that must be scheduled in the same super-instruction. A chain is characterized by edges with latency constraint = 0. We call an instruction *ready* if all its predecessors in the dependency graph are scheduled. We call an instruction *schedulable* when it is ready and all the instructions in the chain are ready as well as there is enough space in the respective buckets to schedule the whole chain. An instruction is *emitted* when it is finally moved from its respective bucket to the output. Instructions that are scheduled but not emitted yet have a special influence during the scheduling.

5.2 Algorithm

As Figure 2 shows we start the scheduling by creating a dependency graph as described in Section 3. By implementation, this directly provides us with a list of instructions that are ready. These instructions are put into the `ready` set.

The main part of the work is split between two major steps "find next schedulable instruction" and "schedule instruction", and a third step "emit instructions" which are explained in the following.

5.2.1 Find next schedulable instruction

In this first step, we try to determine the next instruction that is schedulable. Naturally, this instruction has to be picked from the `ready` set. Hence, `isSchedulable` is

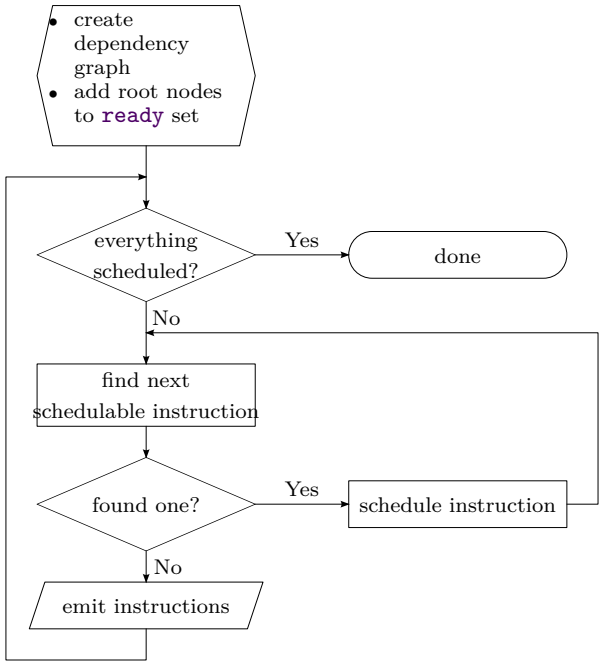


Fig. 2: Flowchart of the scheduling algorithm

called on one element (instruction) after another until we find a result. The challenge here is, that we not only have to consider the instruction itself, but the entire chain; so there has to be enough space in the respective buckets for all the instructions involved.

A goal was to implement this in an efficient way such that we can start with an arbitrary instruction but if find at some point halfway through the chain that there is not enough space, we don't have to do a complicated roll-back of the involved data structures.

Our greedy approach does this by starting with a resource vector of type `struct RemainingSlots` that holds the amount of slots left in each bucket. This resource vector is passed by reference to recursive `isSchedulable` calls.

Every instance of `isSchedulable` now does the following steps:

- It is checked if the instruction has to be stalled because of an instruction that is already scheduled but not yet emitted. This can happen because of sequencing or memory instructions.
- The resource vector is checked if there is enough space in the respective bucket. If so, the corresponding counter is decremented by one.
- `isSchedulable` is called recursively on all depending instructions with latency = 0.

If one of these tests fails the entire attempt on the chain fails and the next ready instruction is tested.

One of the important features of the REPLICA ar-

chitecture is the chaining of instructions. Therefore `isSchedulable` checks if the current instruction is depending on a write-back (`WB`) that is scheduled but not emitted yet. If this is the case and all other prerequisites are met, we save a note that we bypass the write-back and use the output of the functional unit directly that the write-back would otherwise have to save first. Thereby we save one step. If this instruction is the only one that uses the register until it is redefined, the write-back is marked for removal altogether. At the same time we have to be aware of the order inside the super-instruction: E.g. a post-memory-ALU output cannot be used as an operand to a store instruction.

As an example, see the code in Listing 4:

```
OP0 1337        WB2 O0
OP0 42          LD0 R2,O0       WB2 M0
```

This will be rearranged to:

```
OP0 1337        OP1 42          LD0 O0,O1       WB2 M0
```

Listing 4: Skipping a writeback instruction

5.2.2 Schedule instruction

Now that we determined that an instruction plus all recursively dependent ones are schedulable, the instructions are all put in their respective buckets.

Two types of register modifications now have to be applied. Both can be observed in Listing 4.

1) The replacements that were earlier noted because we skip a writeback. `LD0` now uses `O0` instead of `R2`.
2) If we have more than one slot in a bucket, we replace the output buffer that is used. `OP0 42` was moved from slot 0 to slot 1.

All instructions that are pulled in by a latency = 0 constraint are scheduled as well. This goes fine as it was part of the `isSchedulable` check earlier.

All instructions depending on this one have now one predecessor less. Those that have reached zero predecessors are ready. Hence they are put in the `ready` set.

5.2.3 Emit instructions

If "get next schedulable instruction" can't find a suitable instruction, there are apparently not enough slots left. In this case the instructions are emitted as a VLIW and the buckets are emptied. This way of picking the next instruction will eventually cover all instructions because the initial packing into superinstructions derived by the LLVM IR sub-tree matching only imposes minimal requirements (i.e. only one slot per functional unit) and the algorithm will terminate.

6. Parametrization

In order to enforce the compiler to generate target code for a specific configuration we have introduced a set of compiler flags that we will explain briefly. Since the flags are quite many, the compiler is actually run from a script. The first parameter is `-enable-replica-ilp` which tells that the rescheduling should be applied in order to increase instruction level parallelism (ILP); for debugging purposes it can be switched off. The concept of having buckets is mentioned in the previous section, where the idea is to collect instructions that go into the same functional unit. This is intentionally built in a way that there is no restriction as to how many slots there are per bucket. The limitation of the number of slots is only imposed by passing a resource vector from an instance of `isSchedulable` to another. The initial amount (how many slots are there per bucket in a new super-instruction) is not defined in advance but rather given as a command line parameter to the compiler; `-num-ops` tells the number of integer or floating point constants available per super-instruction, `-num-alus` tells how many ALUs are available before the memory unit, `-num-mus` tells number of memory units, finally `-num-alus-post` tells the number of ALUs available after the memory access is done. All these parameters can be shown by calling `llc -help-hidden`.

As explained extensively, rescheduling tries to use the output of one functional unit as an input to another one and thereby bypass write-back operations. Furthermore, the PRAM model implementation by REPLICA requires to hide the latency to memory with different access times. Due to internal matters, this only succeeds when memory accesses are executed in parallel and not sequentially. Therefore chaining data going to and coming from memory is not possible:`-chained-mus` is therefore set to `false` by default. `-enable-replica-inline-integration` tells if an inline assembly string be split up into separate instructions and be subject to following optimization passes. The option to not split and reschedule inline assembly can be useful for debugging purposes. It is important to notice that the dependencies of inline assembly are actually taken care of. For evaluation purposes we have the parameter `-num-reg-read` and `-num-reg-write` to limit the number of general purpose registers that can be read and written respectively in each super instruction, the default is 32 (number of general purpose registers per thread).

7. Evaluation

In order to measure the improvement achieved through the different optimization passes, several test programs were written in REPLICA baseline language. Even though the focus was on instruction level parallelism

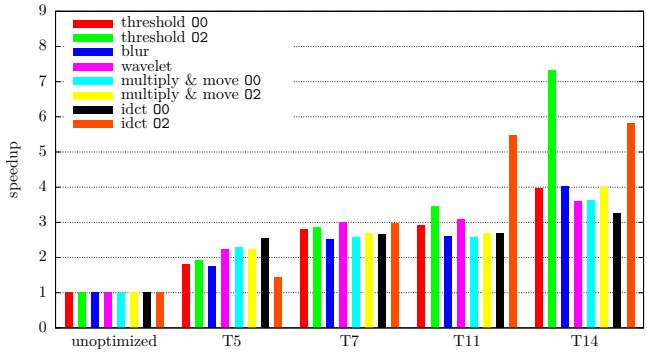


Fig. 3: Instruction level speed-up comparison with different configurations, no constraints on number of read and write ports to reg. file

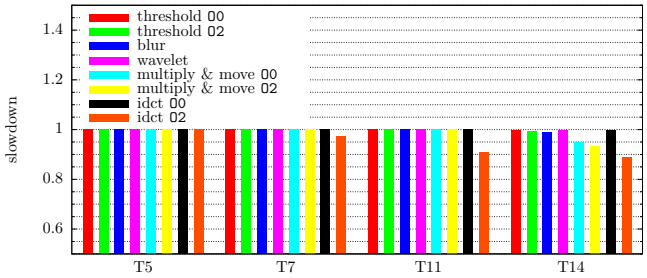


Fig. 4: Relative slowdown when limiting the number of read and write ports to four

we have also used some thread parallel programs as benchmark, to see their characteristics for instruction level parallelism. We have benchmark programs *threshold image filtering* and *blur filtering* which are both thread parallel, moreover a single threaded *discrete wavelet transform* (DWT), where our implementation is based on [5], and one simple test *multiply & move* where elements in an array are multiplied and then moved to another array. We also tested an *inverse discrete cosine transform* (IDCT), this computation kernel was extracted from the Mediabench [6] mpeg2 package.

The baseline case with unoptimized code means that no rescheduling is done, which means that unsplitted super-instructions are used directly from the mapping of the LLVM IR. For each configuration (T5, T7, T11 and T14) we did two test series for each benchmark program. One puts no constraints on the number of register read and writes, which can be seen in Figure 3. In the second test series we limited the number of both read and write ports to the register file to a maximum of four. Figure 4 displays a relative comparison with respect to performance between the two cases. It shows the expected behavior: When we increase the number of functional units without also increasing the number of available registers, speed-up will suffer. For the benchmarks multiply & move and image blur with limited register read and write ports, we will still get almost the same speed-up as in the unlimited case, this is because the functional units can be utilized well since the data dependencies allow us to make use of the buffers in the chained functional units.

8. Comparison to Previous Work

There has been a lot of work done in instruction scheduling, see e.g. [8], [9], [10], [11].

It is well-known that the time-optimal instruction scheduling problem for basic blocks is for most target architectures NP-complete [12]. While smaller scheduling problems can, today, be solved to optimality using integer linear programming and similar techniques, the general case is typically solved by heuristic algorithms.

One classical heuristic solution of the scheduling problem is Graham's greedy algorithm for scheduling tasks [13], known as "list scheduling". It maps tasks to a set of processors, where the tasks can be dependent on each other. The tasks are kept in a list ordered by their dependencies and other priorities so to assign a task to an idle processor it goes through the list and picks the next ready task. If no runnable task is found the processor will idle. Another scheduling algorithm is critical path scheduling [12].

Finlayson et al. [14] show how compiler support can help reducing power consumption without adversely affecting performance when introducing internal registers, that are read and written explicitly, instead of pipeline register. These internal registers reminds a lot of REPLICAs explicit functional unit result registers.

When it comes to VLIW scheduling for architectures with chained functional units we are only aware of the following two, which we compare in detail.

8.1 Original virtual ILP algorithm

Scheduling for a more generalized version of this type of architecture has been proposed by [7]. The algorithm proposed there proceeds in a different way than we do. It is more focused on filling the available memory slots. It sees the next free slot (both memory slots and other ones) and then tries to find a suitable instruction that is independent of all the remaining unscheduled instructions. From our point of view, this has some disadvantages:

- a lot of instructions are inspected which then turn out to have the wrong type.
- it is only taken into account that this instruction depends on other ones, not that other instructions (in this chain) might depend on this one and therefore have to fit into the same super-instruction/ VLIW.

Our algorithm, on the other hand, scans through the set of ready instructions and checks if they can be scheduled.

This seems to be more flexible if dependencies in both directions have to be considered.

8.2 Earlier ILP scheduler for the REPLICA compiler

Together with the general compiler back-end, Åkesson [4] also implemented an optimization pass that aims at exploiting more ILP. The implemented algorithm is based on [7] but looks for the next instruction (as we do) instead of the next slot. As the necessity to manage instructions in a different stage was recognized, different containers exist for holding instructions:

- those that have to be scheduled immediately (within the same super-instruction)
- those that have to be scheduled whenever there is enough space
- those that have to be scheduled at the earliest in the next super-instruction.

One can see that this corresponds to the different latency constraints defined in the dependency graph. This design, however, turns out to be rather cumbersome when we try to use short-cuts and skip write-backs. In order to do this scheduling, a dependency graph was required as well in [4]. The implementation however took in many cases a too conservative approach (e.g., it ignored the fact that output buffers are only valid inside a super-instruction) which resulted in too many unnecessary dependencies that rendered rescheduling more difficult or even impossible. [4] only support one basic REPLICA configuration.

9. Conclusion and future work

We have shown that our implementation of a parametrizable instruction scheduling algorithm for a VLIW processor with chained functional units produces high quality code for different hardware configurations. The high parametrization of the compiler makes it, together with the simulator, useful for evaluating different hardware configurations.

Future work includes the extension of scheduling beyond basic blocks, for example for loops using software pipelining techniques. It can be interesting to evaluate it for the REPLICA architecture since the case of chained functional units is a different one compared to standard VLIW architectures.

Another interesting idea would be to try to formulate and solve the scheduling problem as an integrated code generation problem of instruction selection, scheduling and register allocation together, for instance using *integer linear programming* both at basic block level and beyond. In earlier work Eriksson [15],[16] models integrated code generation for clustered VLIW DSPs using this technique. One example of similarity is that clustered VLIW DSPs have different register files which give constraints on which registers can be used by the different functional units; in our case we have the transient functional registers which are exposed to the programmer and can only be used from left to right and are only valid during one super-instruction step. Both lead to strong coupling between register allocation and instruction scheduling, for which integrated code generation provides higher code quality, see Eriksson [15].

Acknowledgment

This work was funded by VTT, Finland. The authors would like to thank Martti Forsell for his comments.

References

[1] REPLICA, project home-page, http://www.vtt.fi/sites/replica/?lang=en accessed Feb. 10. 2012

[2] J. Keller, C.W. Kessler, and J. Träff. Practical PRAM programming. Wiley series on parallel and distributed computing. J. Wiley, 2001.

[3] M. Forsell. Realizing Multioperations for Step Cached MP-SOCs. Proc. SOC'06, pages 77–82, 2006.

[4] Daniel Åkesson. An LLVM back-end for REPLICA code generation for a multi-core VLIW processor with chaining, LIU-IDA/LITH-EX-A-12/007-SE,Dept. of Computer and Information Science , Linköping University 2012.

[5] G. Pau. Fast discrete biorthogonal CDF 5/3 wavelet forward and inverse transform http://www.embl.de/~gpau/misc/dwt53.c accessed March 9. 2012.

[6] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In International Symposium on Microarchitecture, 1997.

[7] Martti J. Forsell. Using parallel slackness for extracting ILP from sequential threads, In Proc. of the SSGRR-2003s, Int. Conf. on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet, 2003, L'Aquila, Italy 2003

[8] G. Wood, Global optimization of microprograms through modular control constructs, Proc. 12th Annual Workshop in Microprogramming, 1979

[9] J.A Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Trans. on Computers, vol. 30, no. 7, 1981.

[10] M. Tokoro, E. Tamura, T. Takizuka. Optimization of microprograms, IEEE Trans. on Computers C-30:7, 1981.

[11] A. Aiken, A. Nicolau, A Development Environment for Horizontal Microcode, IEEE Trans. on Software Engineering, no. 14, 1988.

[12] C.W. Kessler. Compiling for VLIW DSPs, book chapter, 38 pages, in: S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, eds., Handbook of Signal Processing Systems, Springer, Sept. 2010

[13] R. L. Graham. Bounds for certain multiprocessing anomalies. The Bell System Technical Journal, XLV, November 1966.

[14] I. Finlayson, G. Uh, D. Whalley, G. Tyson. An Overview of Static Pipelining. In Computer Architecture Letters (CAL), accepted 2011.

[15] Mattias Eriksson. Integrated Code Generation Dissertation. Linköping Studies in Science and Technology, Dissertation No. 1375, Linköping University, Sweden, June 2011.

[16] Mattias Eriksson, Christoph Kessler. Integrated Modulo Scheduling for Clustered VLIW Architectures. Proc. HiPEAC-2009 High-Performance and Embedded Architecture and Compilers, Paphos, Cyprus, Jan. 2009. Springer LNCS 5409, pp. 65-79.