Flexible Embedded System Design Using Flowpaths

Darrin M. Hanna, Bryant Jones, Lincoln Lorenz, and Mark Bowers

Abstract—**A typical hardware system can be segmented into components that each define a specific task. Components could be divided among a design team for creation, acquired from another source in the form of legacy work, or an IP core. A finalized system could be composed of several heterogeneous components. Proper integration of these components can often be complex and time consuming for designers. Available tools to aid in rapid hardware design often lack in this area and require confusing pre-compilation procedures. Previous work has been done to create a compiler, called flowpaths, for converting high-level stack-based languages (e.g. Java) to VHDL for use on an FPGA or ASIC. Introduced in this paper is an extension to the flowpaths compiler to allow easier integration of system-components using object-oriented methodologies. System-components can be described by high-level Java classes with methods for interactions with components. These methods are filled with the custom system-components during generation, such that flowpaths acts as the glue logic between separate components. In comparison to handwritten component interconnections, a design integrated with flowpaths shows a decrease in implementation time and design complexity.**

Index Terms—Field Programmable Gate Arrays, Program Compilers, Embedded Systems, Glue Logic

I. Introduction

Often times, large hardware systems are composed using a combination of several heterogeneous components. Usually these components are legacy-based, created by different designers, or acquired from a vendor in the form of an IP core. Integration of several such components into a cohesive system is often referred to as "glue logic". Design of a system using a collection of modules such as these can be quite a difficult task. This can become very time consuming, even for a skilled computer engineer. The need arises for a tool to add flexibility to the design and implementation phases of a multi-component system.

Tools have been created to allow a hardware designer flexibility of design through the use of high-level languages. Examples of such tools include Handel-C and flowpaths; the latter being designed by us. Flowpaths is an architecture described in HDL that can be generated using a stack-based language, rather than a variable/register language which often introduces large fan-out and delay when implemented in hardware. Such stack-based languages include Java bytecode, Common Interface Language (CIL), and Forth. Currently, flowpaths are implemented using Java bytecode, and produce circuits described in VHDL. However, this could be extended to other languages, and different HDLs such as Verilog. Flowpaths are further described in [1, 2]. Table I shows a summary of benchmark results comparing flowpaths to a jStamp microprocessor that natively executes Java bytecode.

Experiment	*Method*	*Time*		*Energy*	
		ms	*ratio*	*mW·ms*	*ratio*
Mandelbrot	JStamp	2.7	1	319	1
	Flowpaths	0.065	0.024	15.55	0.048
FFT	JStamp	237.6	1	44669	1
	Flowpaths	3.84	0.016	714	0.015
Linpack	JStamp	2800.0	1	526,400	1
	Flowpaths	122.2	0.043	34,065	0.064

Table I. Flowpaths versus JStamp performance

These results show improvements in nearly all areas when compared to another embedded system. More results can be found in [1].

These tools allow users to design in a high-level language. Handel-C, for example, uses a subset of C with hardware-specific extensions, while flowpaths uses stack-based languages. Our implementation in Java makes no modification to the standard Java language. On the other hand, Handel-C requires the knowledge of a modified language, allows for design in a high-level language resulting in hardware descriptions that are not practical to modify at the HDL level, and often generates less efficient hardware as described in [2, 3].

Flexible integration of a system is possible in a number of ways. Aside from doing it by hand, tools offer simpler solutions to this problem. Some tools like Handel-C require the pre-compilation of a module for use in a greater system. This can often be confusing and requires many steps. Flowpaths, as we will show, allows for easy integration of custom hardware components into a system through the use of object-oriented design within Java.

This paper describes the flexibility of the flowpath compiler for use in a hardware system. Section 2 outlines how custom VHDL modules can be integrated into an existing flowpath design. Section 3 describes a Mandelbrot fractal explorer using flowpaths to integrate legacy and custom VHDL modules including drivers for VGA, a PS/2 mouse, and user I/O. Section 4 elaborates further with an example of a full robotics system integrated together using flowpaths with custom hardware for GPS, Light Detection and Ranging (LIDAR), a motor controller, a camera with an image processing pipeline,

Manuscript received March 31, 2011.

Darrin M. Hanna, Bryant Jones, Lincoln Lorenz, and Mark Bowers are with the School of Engineering and Computer Science at Oakland University in Rochester, MI (dmhanna@oakland.edu).

and an Inertial Measurement Unit (IMU). The paper closes by providing concluding remarks and future work.

II. Object-Oriented System Design

Flowpaths have the flexibility to utilize custom hand-crafted VHDL components for use in generating hardware. This gives a computer engineer several options for design of a hardware system. Examples of where this can be used include: replacing a complex portion of a generated flowpath with an optimized custom component for greater efficiency, or the integration of several modules with a flowpath as the interconnection fabric. This is described by Fig 1.

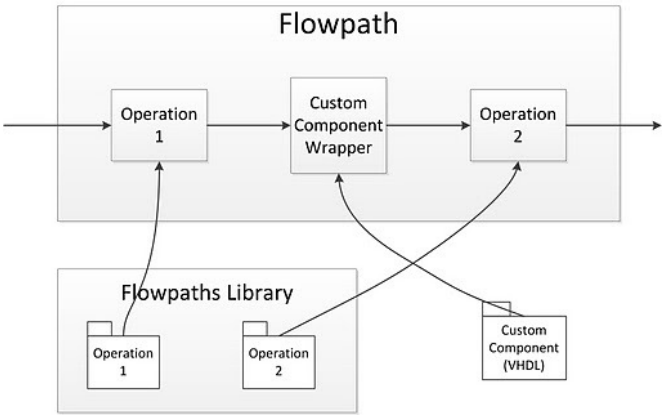

Fig 1. Overview of flowpath component integration

Legacy, custom, or IP core components, can be incorporated easily into an algorithm for an embedded system using flowpaths. Each component can be described using object-oriented methodologies. Components are characterized as objects of the system, with its interactions as functions of that object. This can be easily described in Java by creating a class for each component. Within this class, a method must be made for each function of the component. Alternatively, since a component's interface could be described by a single method, one class could be created with a method for each component. Since the generated flowpath for a method will contain a custom component, the Java method can be left empty. The only parameters that need to be established for a method are the inputs and outputs. These will correspond to stack inputs and outputs of a datapath. The generated flowpath will handle parameter passing into and out of the component. The only change needed to adapt the corresponding hardware component for use in the flowpath, is to make the component conform to a standard operation interface. This interface consists of the changing execution stack (StackIn and StackOut buses) and propagation signals (enable-in and done-out).

Events are usually handled in an embedded system with one of two methods; polling or interrupts. Flowpaths does not currently support the use of interrupts, and therefore handles events using polling. A main execution loop can be created, as in most embedded systems, to allow the generated flowpath logic to interact with the components. When an event occurs within a component it must be registered until the next time it is polled. Once that component is polled again, the event will be handled and reset. Examples given in this paper use this polling method.

III. An Embedded System for Exploring the Mandelbrot Set

A complete Mandelbrot explorer system was implemented using the flowpaths compiler. The Mandelbrot set is a fractal image produced by iterating a quadratic polynomial across points in the complex plane [5]. The system starts at an initial image of the Mandelbrot set, and allows the user to zoom further into a desired portion of the set, recalculating the image and yielding more and more detail. This zooming behavior is shown in Fig 2.

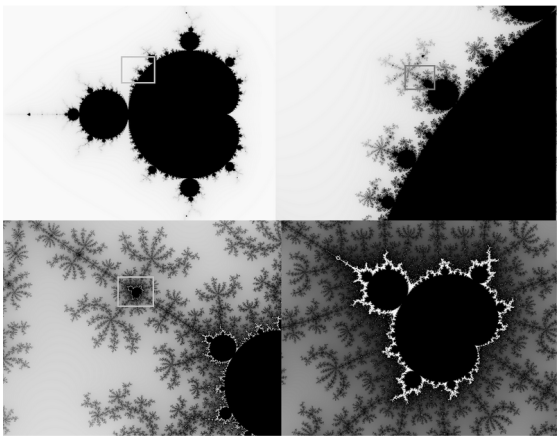

Fig 2. Example Mandelbrot fractal zoom sequence

A VGA driver is used for displaying the Mandelbrot fractal. A PS/2 mouse driver is used for zooming into and out of the fractal. Various calculation parameters are set via switches and buttons using a handwritten component. The VGA and mouse drivers are legacy components. The Mandelbrot calculation core consists of two parallel flowpaths which were generated from a Java algorithm. Each of these components is integrated easily using the object-oriented method presented in this paper.

A class was created containing methods that describe the interface to the existing components. A single method was created for each component. Methods called *PollMouse*, *PollSwitches*, *MandelbrotCalc*, and *PlotPixel* were created. These methods describe the inputs and outputs of the components. The *PlotPixel* and *PollMouse* methods are shown as examples in Listing 1.

```
static void PlotPixel(int addr, int color){
    // write 16-bit VRAM word -> two 8-bit pixels side
    // color(16 downto 8) -> pixel1
    // color(7 downto 0)  -> pixel2
}

static int PollMouse(){
    // returns:
    // 11 downto 0   <= X position
    // 23 downto 12  <= Y position
    // 24 <= r click detected
    // 25 <= l click detected
    return 0;
}
```

Listing 1. *PlotPixel* and *PollMouse* interface methods

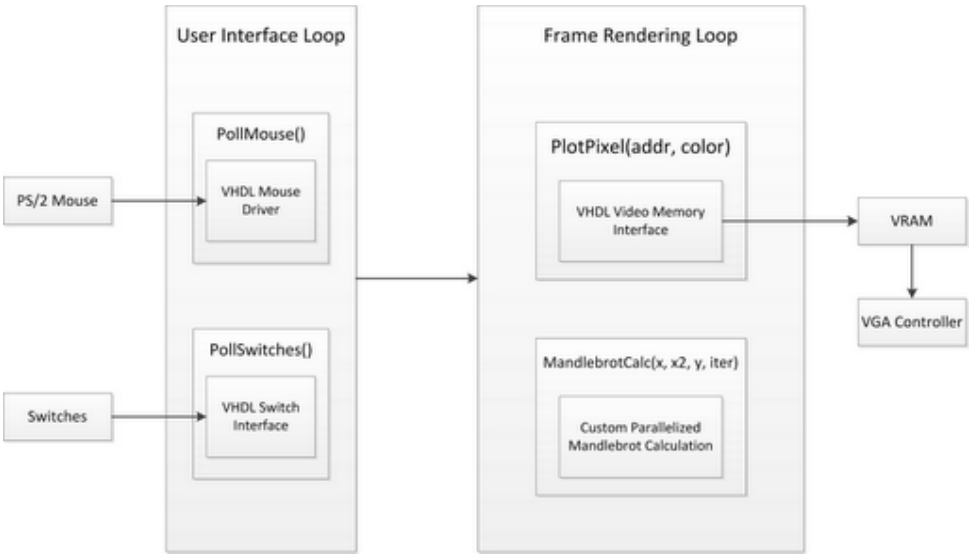

Fig 3. Overview of the Mandelbrot explorer flowpath

The functional interface of these components is shown in Fig. 3. The user interface loop continuously polls the status of the mouse and switches using the *PollMouse* and *PollSwitches* methods, respectively. Momentary events (button presses, mouse clicks) are captured within these methods and handled when the device is polled again.

Mouse clicks trigger the frame rendering loop. This loop recalculates the image based on user input. The calculation is performed in the *MandelbrotCalc* function. This function represents a component consisting of two instances of a previously generated Mandelbrot flowpath, combined in a manner such that they operate upon two Mandelbrot points in parallel. Each instance performs an iterative calculation upon a given point in the Mandelbrot set. The number of iterations taken corresponds to the appropriate pixel color, which is then written to video memory with the *PlotPixel* function. This function writes directly to VRAM, by simply wiring the appropriate stack elements to the memory's address and data lines, and the operation enable pulse to the memory's write enable.

Implementing the frame rendering loop in VHDL would require a complex state machine and take a significant amount of time to implement. The Java version, on the other hand, merely consists of two nested *for* loops and two function-calls, which can be created very quickly. A comparison was performed on several implementations of a Mandelbrot system. The results are shown in Table II.

Calc Core	Glue Logic	Logic Elements Consumed	Draw Time (ms)	Draw Time (ratio)	Approx Dev. Time
Flowpath 32-bit	Flowpath	15,772 (47%)	970	1.000	30 min
Flowpath 32-bit	Manual	8,379 (25%)	860	0.886	3 hours
Manual 32-bit	Manual	3,420 (10%)	414	0.426	1 Day

Table II. Relative performance of Mandelbrot implementations

The Mandelbrot explorer system was developed using several different methods. The systems were subjected to a standard benchmark using the same calculation parameters to produce an image of the Mandelbrot set. The flowpath-generated glue logic had a slight impact on the draw time, but was developed much quicker. The flowpath-generated calculation core yielded performance half as fast, and utilized twice as much logic as a handwritten core, but again, was created in significantly less time.

IV. Robotic System Integration

An FPGA-based system for an autonomous ground robot is currently being developed using flowpaths to implement intelligent algorithms that depend on several components. In robotic systems, there are generally multiple sensors, sensor processing, intelligent processing algorithms, and signal conditioning components, among others. Using a high-level object-oriented programming language to create a robotics control algorithm, streamlines the design of a robotic system. The ability to generate special-purpose hardware from a high-level Java description makes it practical to implement such a system on an FPGA. The easy description allows for future changes to the system to be made quite easily.

The system is based on the simple sense-think-act loop model, where a robot acquires the latest sensor data, makes a decision based on its current state estimate, and sends control signals to act upon its decision. Sensors and other modules are connected to the FPGA through external I/O pins. Each custom component polls its sensor and reads data at the interface update rate. Handwritten VHDL components have been developed for interfacing to a GPS sensor, a LIDAR sensor, and motor controller. Other components such as a camera interface, an image processing pipeline, and an IMU interface are in development. Fig 5 illustrates the interfaces and glue logic that will be used to integrate the system.

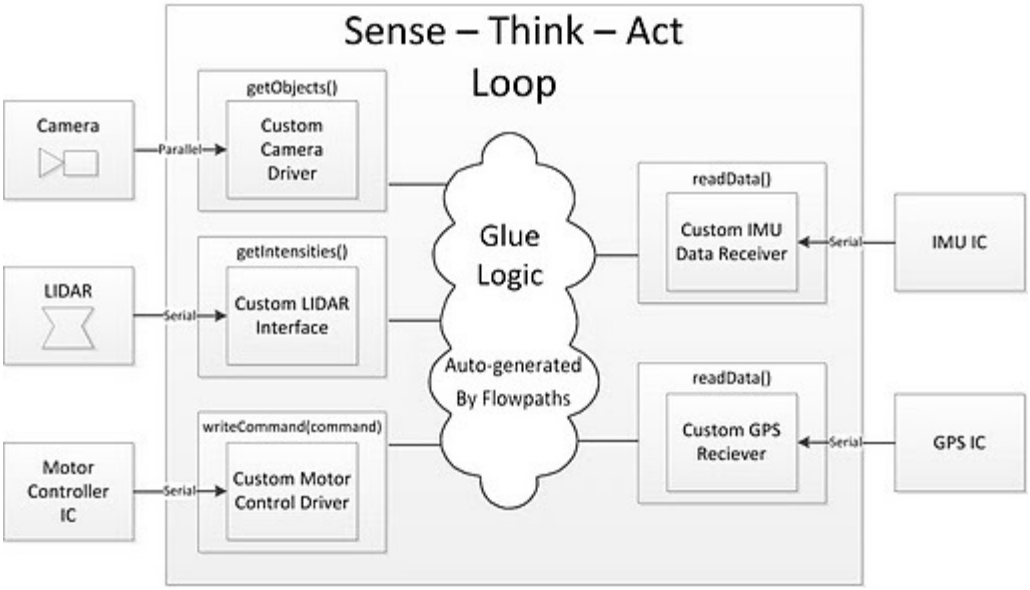

Fig 5. Object-oriented robot design

Individual classes were created to describe the functions of the handwritten components. Each component class contains the methods used to interface with that component and additional helper methods.

The motor controller component utilizes a built-in closed loop control algorithm to control the motors' speeds using encoders for feedback. Its interface requires a system speed and heading angle to compute the velocity for each motor. To interface to the component, a Java class named *MotorController* was created that contains a method called *sendCommand* which takes speed and heading angle as parameters.

A GPS sensor is used for detecting the robot's position on the earth. The GPS component receives the latitude, longitude, altitude, and status information from the GPS sensor. The custom hardware handles all of the details of the interface to the sensor including the parsing of the serial message**.** A Java function, *readData* was created as an interface to the component that reads the latest GPS sensor data. Normally, the *readData* function would return the full GPS message in one read, however, since the Java language limits the number of return values of a function to one, a workaround is needed. It accomplishes this by using an index to select which portion of the data to read. In this way, the data can be read out serially using a call to *readData* for each index. This presents another obstacle since *readData* is called in four places. This normally will generate four instances of the *readData* wrapper including the handwritten hardware. This is redundant and makes connection to an external interface unfeasible. To avoid this, one wrapper file is created and a multiplexer is inserted to control access to the custom hardware from the four different locations. To provide a cleaner external interface, the GPS data is compiled into a *GPSMessage* object. Using the method *readMessage*, the *GPSMessage* passed to it will be populated with the current data. The Java class which provides the interface to the GPS sensor is shown in Listing 2.

```
/**
 * Controller interface to the GPS Sensor
 */
public class GPSDriver{

    public void readMessage(GPSMessage message){
        //populate message
        message.status = readStatus();
        message.longitude = readLongitude();
        message.latitude = readLatitude();
        message.altitude = readAltitude();
    }

    private int readStatus(){
        return readData(0);
    }

    private float readLongitude(){
        return readData(1);
    }

    private float readLatitude(){
        return readData(2);
    }

    private float readAltitude(){
        return readData(3);
    }

    private int readData(int index){
        //get various data from buffer
        // in handwritten component
        return 0;
    }
}
```

Listing 2. GPS interface

To properly navigate the robot, environmental surroundings need to be monitored constantly. A LIDAR and camera are used for this task. The LIDAR operates by receiving laser ranging data in a series of angle-referenced intensity bins which are used to determine surrounding objects. These are read through the Java wrapper method *getIntensities* and added to a LIDAR message created in Java, much like the *GPSMessage*. The camera, on the other hand, acts as the robot's eyes using an algorithm to detect objects in the robot's view. Its corresponding hardware component calculates a list of objects described by their three dimensional location and size. The interface to the camera component reads this list of

objects and combines them into a message containing an array of objects.

Furthermore, the robot must be conscious of its orientation relative to the environment. An IMU is used for calculating the robot's relative position and system's current motion. This component receives orientation, velocity, and acceleration data from the IMU. These data are returned from the component using an interface wrapper, as with the other sensors.

The full system can be pulled together using a sense-think-act loop written in Java. Flowpath algorithms can be written on top of the glue logic used to connect the components. The model can be implemented using a main execution loop with the divisions (sense, think, and act) called in order. Sensing can be done by polling the individual sensor interfaces. On each iteration of the loop, the sensors will be queried for data. The robot will have detailed information about its environment and its position relative to it. Using this sense data, the robot will be able to effectively decide how to react. This can be calculated based on its estimated current state and its goal. Currently, response algorithms are unimplemented in this design, however, the creation should be relatively straightforward using Java. Once the system has chosen its path, the robot can use the *sendCommand* interface of the motor controller and the loop repeats.

V. CONCLUSION

This paper shows how the flowpaths compiler can be used for flexible design of a hardware system which uses legacy, custom, and IP core components. These can be integrated easily into the flowpaths structure through the use of custom classes in Java. By creating a class for custom components, an inter-connection layer can be created in flowpaths to wire the modules together. Using this technique allows for a time savings in development with respect to integration of components.

Two examples were given to demonstrate how using object-oriented design and flowpaths reduced the time and complexity in implementing embedded systems using legacy, IP core, and custom components. A Mandelbrot fractal example demonstrated the integration of a few components with minimal complexity. Also, a real-life robotics system that is underway was described including several components for interfacing outside peripherals such as a camera, LIDAR, a motor controller, an IMU, and a GPS module. Through these examples, we have shown how flowpaths can be used to quickly and easily generate efficient inter-connection logic for a system using object-oriented principles.

VI. FUTURE WORK

Currently in Java, returning an object uses memory. To alleviate this issue, a special reserved class could be implemented to handle this without the use of memory. Work also includes the implementation of automated mapping of external signals into a generated flowpath. This includes the ability to utilize pin resources on a specified FPGA to include the proper connection from within a Java algorithm; similar to the classes implemented by a jStamp embedded processor [6]. Creation of timing libraries, optimization techniques, and process threading are also being explored as possible additions to the flowpath compiler for use with flexible design control.

REFERENCES

1. D. M. Hanna, B. Jones, L. Lorenz, and M. Bowers, "Generating Hardware from Java Using Self-Propagating Flowpaths," Submitted to the International Conference on Embedded Systems and Applications, 2011.
2. D. M. Hanna and R. E. Haskell, "Flowpaths: Compiling Stack-Based IR to Hardware," Microprocessors and Microsystems, vol. 30, pp. 125 - 136, 2006.
3. S. A. Edwards, "The Challenges of Hardware Synthesis from C-like Languages," Proc. of Design Automation and Test in Europe (DATE), Munich, Germany, 2005.
4. D. M. Hanna, M. Duchene, L. Kennedy, and B. Carpenter, "A Compiler to Generate Hardware from Java Byte Codes for High Performance, Low Energy Embedded Systems," The 2007 International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, NV, June 25 - 28, 2007.
5. Benoît Mandelbrot, *Fractal aspects of the iteration of* $z \rightarrow \lambda z(1-z)$ *for complex* λ, z, Annals NY Acad. Sci. **357**, 249/259
6. Systronix, "JStamp: Real-time Native Java Module," 2003.