

Java Flowpaths: Efficiently Generating Circuits for Embedded Systems from Java

Darrin Hanna^{a,*}, Michael DuChene^a, Girma Tewolde^b, Jay Sattler^a

^aDept. of Computer Science & Engineering, Oakland University, Rochester, MI 48307 (speaker)

^bDept. of Electrical and Computer Engineering, Kettering University, Flint, MI 48504

*dmhanna@oakland.edu

ABSTRACT

The performance of software executed on a microprocessor is adversely affected by the basic fetch-execute cycle. A further performance penalty results from the load-execute-store paradigm associated with the use of local variables in most high-level languages. This paper describes creating Java flowpaths, a general method of generating hardware directly from Java bytecodes that support basic operations, class instantiation, events, and multi-threaded applications. Performance increases occur because flowpaths created from Java programs require zero clock cycles for load, stack manipulation, and class instantiation operations. In addition, multithreaded flowpaths do not suffer from traditional processor bottlenecks such as context switching, stack manipulation and the traditional load, execute and store operation. Results show that these improvements yield several orders of magnitude improvement in terms of speed for both sequential and multithreaded programs.

Keywords

FPGA, Java, Flowpaths, Multithreading, Event-driven

1. INTRODUCTION

FPGAs promise the flexibility of software running on a microprocessor together with increased performance in terms of throughput. However, this flexibility usually requires one to be a hardware designer using a hardware description language such as VHDL or Verilog. Several methods for using a language like C as a hardware description language or to generate hardware from software have been researched over the past 15 years [1-8], but in most cases the designer is still thinking in hardware terms and solutions like these have seen little commercial interest [9].

There is an inherent performance penalty in using any microprocessor. A microprocessor contains a general-purpose datapath and control unit that executes a sequence of instructions representing a software program that is stored in memory. A dedicated datapath and

control unit that executes only a particular software program or algorithm can be more efficient in terms of both speed and area. It would save development time to take a traditional software program and compile it directly to hardware (FPGA or ASIC) in the same way as a software program is compiled to a particular microprocessor. The goal is therefore to generate this dedicated datapath and control unit automatically from a standard software program. This datapath and control unit can be described by a hardware description language such as VHDL or Verilog to synthesize to an FPGA. Note that this is not the same as writing the software program directly in Verilog or VHDL because although the program might be able to be simulated it would not, in general, be synthesizable. For example, a while loop that depends on unknown inputs cannot be synthesized in VHDL or Verilog.

Flowpaths have proven to be useful in automatically converting software programs written in a stack-based language such as Java bytecodes or Forth to hardware [10, 11]. This is accomplished by representing the software program as a stack-based algorithm that leads to an efficient datapath and control unit that can then be synthesized using VHDL or Verilog. Results show that flowpaths can perform within a factor of 2 of a minimal hand-crafted direct hardware implementation and orders of magnitude better than compiling the program to a microprocessor optimized for Forth or bytecodes [12].

Concurrency is fundamental to hardware and is not well-represented by most software languages. C and C++, for example, are optimized for expressing sequential algorithms and have no language-level support for concurrency [9]. Many of the languages that support software-to-hardware compilation have added constructs to support parallelism that complicate the language and software development process. Java is a multithreaded language that provides built-in support for threads that makes parallel programming with threads easier.

Java is one of several software-programming languages that compiles to an intermediate representation (IR) that is stack based. The Java Virtual Machine (JVM) is stack-based. Each thread has a JVM stack that stores frames created each time a method is invoked. The frame contains

an operand stack, an array of local variables, and a reference to the runtime constant pool of the current method's class [13]. To execute instructions, variables are loaded onto the stack, the instruction is executed, and the answer is re-stored again. In this fashion, a combination of stack and memory are used in this "stack-based" JVM.

This load-execute-store feature of Java bytecodes maintains the normal microprocessor paradigm. Indeed microprocessors have been developed that execute these Java bytecodes directly [14]. A popular area of research involves executing instructions on a microprocessor core in an FPGA. One such area of work is in developing a processor core to execute Java bytecodes on an FPGA [15-17]. These implementations attempt to overcome the slow performance of traditional interpretive Java virtual machines (JVMs) by introducing new methods for executing Java bytecodes as native processor instructions. However, the performance of these microprocessors still suffers from the excessive use of local variables in the original software program.

A programming language that inherently minimizes the use of local variables is Forth. A Forth program consists of a sequence of words, each of which is a sequence of other Forth words with a parameter stack for passing data between words. In this way Forth programs typically use far fewer local variables than more traditional languages. The tradeoff is that Forth requires stack manipulation words to manipulate elements on the stack. Many of the Java processors are stack machines that have been derived from Forth processors [17].

High-level Forth programs and basic Java bytecode operations can be converted to flowpaths that can be synthesized directly to hardware using VHDL or Verilog as shown in [12]. Stack manipulation words are represented in flowpaths as simple wire connections that take zero time to execute. In this sense the stack has completely disappeared once the software program is converted to flowpaths, avoiding the load-execute-store penalty without incurring the stack manipulation penalty of traditional Forth programs. In a similar way the load-execute-store construct in Java bytecodes is represented in flowpaths as a single sequential operation. Rules for converting all Forth words to flowpaths, implementations, and results can be found in Hanna (2003) [10]. We have implemented a basic compiler using most of these rules.

The input to the flowpath generator is either a standard Forth program or standard Java bytecodes. This means that high-level software programs written in either Java or Forth can be directly compiled to hardware. The hardware can be implemented in an FPGA resulting in significant improvement in performance compared with executing the same software using a microprocessor.

Such a microprocessor-less architecture is capable of executing multiple tasks using a single clock, truly in

parallel, while a single processor can only task switch or expensive multi-processor architectures must inefficiently synchronize. This paper introduces Java flowpaths for generating hardware directly from Java bytecodes supporting basic operations, class instantiation, event handling, and multithreading. Section 2 describes instantiating class instances in hardware. Event handling is covered in Section 3. Section 4 describes elements for creating parallel flowpaths from multithreaded Java bytecode. Finally, conclusions are given in Section 5.

2. CLASS INSTANTIATION FROM BYTECODES TO FLOWPATHS

In addition to general operations described in *Hanna, et. al.* [11], Java flowpaths must support a framework for instantiating Java classes from bytecodes. In this framework we have captured the essential features to create object instances from class specifications, uniquely identifying the objects using an object registry system, creating the execution logic for the methods in the class, providing a method registry system, and incorporating a mechanism for handling nested method calls. Two general approaches are presented and compared for performance vs. logic and routing resource needs. The framework is further demonstrated using an example application.

In Section 2.1 the JVM's specification with regard to class instantiation is summarized to provide an understanding of this whole process in a traditional software implementation of the JVM. In Section 2.2 our general framework for extending flowpaths to support class instantiation in hardware is presented. Two approaches are discussed and compared.

2.1 Class Instantiation in JVM

The Java Virtual Machine (JVM) Specification gives an abstract specification of the internal behavior of the JVM leaving the implementation details for the designer. We will use Venners [18] to present some relevant information about the internal workings of the JVM that relates to our work. The major internal architectural components of the JVM are a *class loader subsystem*, an *execution engine* and *runtime data areas*. The class loader subsystem provides a mechanism for loading types (classes and interfaces) from given fully qualified names. The execution engine is responsible for executing the instructions contained in the methods of loaded classes. When a JVM runs a program, it uses the runtime data areas to store many things, including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables and intermediate results of computations.

Each instance of the JVM has one *method area* and one *heap*, which are both shared by all threads running in the

virtual machine. Type information from classes that the JVM loads is stored in the method area, while all objects that the program instantiates during its run-time are placed in the heap. Each executing thread has its own context that is made up of a *program counter (PC) register* and a *Java stack*. The *PC* register indicates the next instruction to execute (for non-native methods) and the thread's Java stack stores the state of Java method invocations for the thread. State information of a Java method includes its local variables, method input parameters, its return value and intermediate calculations.

The Java stack is composed of *stack frames*. A stack frame contains the state of one Java method invocation. When a thread invokes a method, the JVM pushes a new stack frame onto that thread's Java stack. When the method completes, the JVM pops and discards the frame for that method. This stack-based architecture of the JVM simplifies implementation on a wide range of processor architectures and facilitates code optimization by Just-in-time (JIT) and dynamic compilers that operate at run-time in some implementations.

According to the JVM Specification, the class loader subsystem is responsible for loading, linking and initialization of types. The loading requires finding and importing the binary data for the type. During the linking phase the JVM performs verification to ensure correctness of the imported type; prepares memory for the class variables and initializes the memory to default values; and transforms symbolic references from the type into direct references. Finally, in the initialization phase the JVM invokes Java code that initializes class variables to their proper starting values.

For each type it loads, a JVM must store the following kinds of information in the method area: the fully qualified name of the type and its direct superclass (unless the type is an interface or class `java.lang.Object`), whether or not the type is a class or an interface, the type's modifiers (some subset of `public`, `abstract`, `final`), an ordered list of fully qualified names of any direct superinterfaces, the constant pool for the type, field information, method information, all class (static) variables declared in the type (except constants), a reference to class *ClassLoader*, and a reference to class *Class*.

The *constant pool* is an ordered set of constants (string, integer, and floating constants) and symbolic references to types, fields, and methods. Entries in the constant pool are referenced by index. For each field declared in the type, the *field information* stored in the method area includes the field's name, type and modifiers (some subset of `public`, `private`, `protected`, `static`, `final`, `volatile`, `transient`). For each method declared in the type, the following *method information* is stored in the method area: the method's name, return type (or `void`), number and types (in order) of parameters, and its modifiers

(some subset of `public`, `private`, `protected`, `static`, `final`, `synchronized`, `native`, `abstract`). For each method that is not `abstract` or `native`, the method area also stores the method's bytecodes, size of the operand stack and local variable sections of the method's stack frame, and an exception table. *Class (static) variables* are shared among all instances of a class and can be accessed, even in the absence of any instance; hence each non-`final` class variable is part of the class data and must be allocated memory in the method area.

Direct implementations of the entire process of the class instantiation as described above in the JVM Specification are available in many different platforms where the full JVM is implemented. For our purpose, however, it may not be feasible or necessary to incorporate all aspects of the process because of the embedded purpose of our execution architecture and interest in conserving target computing resources. At this point, we allow only static instantiation of types with all the types and the objects instantiated from classes known at compile time. Therefore the actual creation of the objects on the FPGA and all the methods required for the lifecycle of the application is done at initial load time. The next section provides details of our class instantiation framework.

2.2 Framework for Class Instantiation in Hardware

To better understand the internal mechanics of class instantiation consider a simple example class *Foo* that has a constructor for initializing objects of its type and a simple instance method as shown in the Java source code in Listing 1 and its compiled bytecode shown in Listing 2.

Listing 1: A Class *Foo* and Instantiation in Java

```
class Foo {
    int a, b, c;
    Foo(int b, int c) {
        this.b = b;
        this.c = c; }
    void imethod1() {
        a = b+c; }
}
public class TestFoo {
    public static void main(String args[]) {
        Foo tf = new Foo(2, 3);
        tf.imethod1();
        System.out.println("(" + tf.a + ", " + tf.b + ", " + tf.c + ")");
    }
}
```

Although we haven't explicitly extended any class in our Java source code for *Foo*, all Java classes derive from `java.lang.Object` as shown at the start of the bytecode file. The file then lists all the fields of the class as defined in the source. When the class is loaded, the loader uses this

information to allocate space for the fields of the class variables as the class is first referenced. Instance variables are allocated only when an object instance is instantiated from the class. Following the fields list is the *Foo* constructor. The constructor starts by calling (**invokespecial**) its parent constructor to instantiate static and instance fields of the parent class. Note that a reference to the object instance being instantiated (*aload_0*) is passed as parameter on the stack when invoking **invokespecial**. All constructors and instance methods require the object instance's reference (termed as "**this**" in a Java source code and available in the local variable array of all instance methods) to be passed to them as their first parameter. The rest of the code in the constructor's bytecode simply initializes the object's fields using the input parameters passed to it. The "**putfield #n**" instruction is used for this purpose with the object's reference and the integer value to be stored in the field passed as parameters to it on the operand stack. The "*n*" in **putfield** identifies the field index in the array of memory holding the object's fields.

Listing 2: Java Bytecodes for the Java Code in Listing 1

```

Compiled from "TestFoo.java"
class Foo extends java.lang.Object {
int a; int b; int c;

Foo(int,int);
Code:
0: aload_0
1: invokespecial #1; //Method java/lang/Object.<init>:()V
4: aload_0
5: iload_1
6: putfield #2; //Field b:I
9: aload_0
10: iload_2
11: putfield #3; //Field c:I
14: return

void imethod1();
Code:
0: aload_0
1: aload_0
2: getfield #2; //Field b:I
5: aload_0
6: getfield #3; //Field c:I
9: iadd
10: putfield #4; //Field a:I
13: return
}

public static void main(java.lang.String[]);
Code:
0: new #2; //class Foo
3: dup
4: iconst_2
5: iconst_3
6: invokespecial #3; //Method Foo.<init>:(II)V
9: astore_1
10: aload_1
11: invokevirtual #4; //Method Foo.imethod1:()V
.
.

```

In this simple example the instance method "method1()" operates on the two field values entered via the constructor to generate a value for the first field. The "**getfield #n**" bytecode instruction is used to retrieve the instance fields; and after computing the result the "**putfield #n**" stores it in the specified field. A **getfield** leaves the field value read from the object instance on the operand stack.

In the **main()** method the "**new #n**" bytecode creates an instance of the *Foo* class using its reference indexed by *n* in the constant pool. The actual initialization of the object's fields is carried out by the constructor that is invoked by the "**invokespecial #n**" at line 6 to which are passed the object's reference and two integer parameters. **Invokespecial** returns on the operand stack the reference to the newly created and initialized object. The instance method **imethod1()** is then called by "**invokevirtual #n**" which also needs a reference to the instance object as a parameter on the stack.

A class definition may contain several constructors with different number and types of parameters. However, the hardware implementation need only contain the relevant ones that are used in the instantiation of objects in the application. By parsing through the **main()** method in the bytecode of the application and all the other methods that are invoked by **main**, and all methods invoked by those methods, and so on, we can figure out the actual constructors and other methods that are used in the life of the application. Therefore, we need allocate resources and download only those relevant objects and methods on the FPGA hardware. A registry mechanism has been developed for keeping record of all the instantiated objects and all the installed methods in the system. This mechanism facilitates easy access of objects and instance and class methods.

Two different approaches of hardware instantiation of classes have been investigated in this study:

- 1) Sharing of methods by all instance objects of a class
- 2) Inlining methods where they are required

In the first approach only single copies of the instance and static methods needed for the application's execution are implemented on the FPGA. A structured routing mechanism is designed to allow each object instance access to these methods. In this class instantiation framework we are considering a single thread of execution hence there will be no issue of concurrent execution of methods, since at any one point in time only one method is actively running.

At the other extreme of the class instantiation techniques one can consider handling all method invocations in the Java applications by inlining the whole method body at the point it is referenced. In this case, a single method will be implemented on the FPGA as many times as it is referenced in the application.

Each approach has its own implication on hardware resource requirements for their implementation, specifically logic vs. routing resources. The method-sharing approach demands more routing resources but less logic resources than the method-inlining approach. Comparing execution performances, the method sharing technique is expected to introduce some performance penalty due to the multiplex logic required to switch signals between the different object instances and the methods in the system.

There is no hard rule one can follow in choosing one approach over the other, or to give some “typical” scenario where one will be preferable over the other. Depending on the number of objects and total number and complexity of methods referenced by each individual object one or the other technique may be more efficient. If the application instantiates just a single object, both techniques will be the same. If the application creates several objects (say 5 to 10) and each object references a large number of methods, then the method sharing technique will be more efficient in resource usage.

Ideally, a hybrid approach combining both method where appropriate on an instance-by-instance bases can be used; simple and short methods would be implemented inline to conserve routing, while complex and long methods may be shared by some or all object instances to conserve space. In this manner, the benefits of both approaches may be realized.

3. EVENT HANDLING IN FLOWPATHS FROM JAVA BYTECODES

Java is an event-driven language. This is another aspect that makes Java useful for describing hardware; it is easy to write an algorithm and call it from an event handler registered to be notified on rising, falling, or other activity on a pin, signal, or bus.

When parsing Java bytecodes event sources must be identified to create the necessary flowpath hardware for it. All event sources are derived from an *empty* EventSource class so that the header information in the Java bytecode distinguishes event sources from other code. When an application creates (with **new**) and instantiates an *EventSource* class, an *EventMgr* flowpath component is included in the flowpath project. The event manager is responsible for registering the particular event source, managing the registration of interested event listeners, and dispatching of events to the registered listeners.

Although it is possible to have a single event manager take care of multiple event sources, our current implementation designates an event manager for each individual event source, keeps records of all interested listeners and takes care of dispatching the events. For example, if an application contains pin events that may

occur on two or more pins, an instance of a separate event manager is generated for every pin event source instead of managing all pin events by a single event manager.

When a listener object registers with a particular event source, it executes an “**addxxxEventListener**” method on the event source and provides its own object ID for the registry. The *EventMgr* will then include the object ID in its list of listeners. The listener can also remove itself from the registry by executing “**removexxxEventListener**”, which is handled accordingly by the *EventMgr*.

In Java, when events occur the source object executes a “**notifyxxx**” method that cycles through the list of its registered listeners and initiates the execution of the listeners’ callback functions to handle the event.

Although the JVM software implementation of the “**notifyxxx**” method goes through the registered listeners serially, one at a time, our flowpath hardware realization enables execution of all listener objects simultaneously in one clock cycle. At the same time the main flowpath algorithm can be allowed to continue executing while the event handlers are running, although in many applications the system works by responding to external events with little done in the main function.

If the main flowpath and the event handler(s) share some common data, a shared access controller is put in place to synchronize the data access. This shared-memory access controller is detailed in Section 4 for multi-threaded applications. Data access operations from different locations (or methods) in the main function and from the event handler is routed to the access controller that regulates the access on a priority basis. Higher priority is given to access operations coming from the event handler. Figure 1 shows the event handling architecture with a single event source, managed by a single “EventMgr” and three interested event listeners.

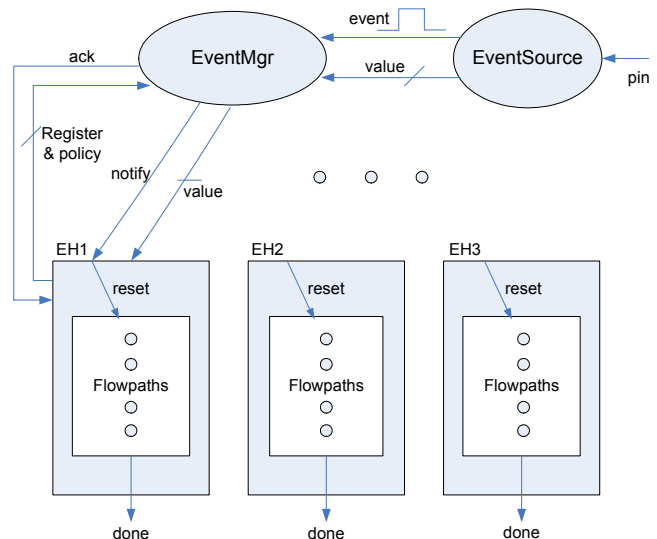


Figure 1: *EventMgr* for a single event source and multiple listeners

When registering with the *EventMgr* the listeners send a register request signal along with a two bit policy that specifies the activity that the handler will listen to. The policy value definitions are disabled, rising edge, falling edge, and both edges. Sending a “00” for policy along with the request to register will disable the listener and it will no longer receive notify signals from the event manager.

After registering a handler, the *EventMgr* sends an acknowledgment (*ack*) signal to the event handler. The *EventMgr* can receive simultaneous requests to register as event handler from multiple handler objects. Since the requests are made on separate lines, the manager can identify each registering listener and send back *ack* signals.

When an event is fired by the event source, the event signal along with the value of the event (one or multiple bit values) is passed to the *EventMgr*. The manager then examines the policy of each event handler and passes along the event notify signal and the value to those handlers whose policy matches the event type. Those handler methods are executed in the normal flowpath fashion using *reset* and *done* signals.

4. PARALLEL FLOWPATHS FROM MULTITHREADED JAVA

Java threads can become quite involved. Although Java is platform independent, it is very difficult to write a truly platform independent multithreaded program. The purpose of this section is to provide general information to familiarize readers with Java threads who have limited experience programming multithreaded software applications. We are particularly focused on aspects of threading that are most applicable to developing algorithms that execute in parallel for embedded applications. For more information regarding Java threads, we recommend reviewing the Java-related references [19-21].

There are two ways of implementing threads in Java. The first way is to develop a class that extends the *Thread* class. The second is to implement the *Runnable* interface. In either case, the class needs to implement a method called *run()*. The *run()* method is the code that is executed when a thread is started.

Threads have priorities that help determine which thread will receive access to system resources such as CPU time and shared data. Resource management is dependent upon the particular JVM on which the thread is executing.

Access to shared data is controlled with the use of a monitor. A *monitor* is a collection of shared resources and a methodology for scheduling access to these

resources [22]. A monitor is necessary to avoid obvious read/write conflicts with shared memory. In the context of the JVM, monitors exist as a sort of mutex to control access to shared data.

Access to CPU time between multiple threads depends upon the JVM and how it schedules threads. The Javasoft specification for threading does not require any particular type of preemptive time slicing. However, time slicing is available on common platforms such as win32 and Sun.

Another important concept in Java threading is the critical section. A *critical section* is a section of data that is to be shared and, therefore, must be locked and treated as an atomic unit with respect to data manipulation [19]. Java uses the *synchronized* keyword to mark critical sections. The *synchronized* keyword can be used in either a block form or on an entire method.

Java threads can be coordinated through the use of the *notifyAll()* and *wait()* methods. The *wait()* method is similar to the *sleep()* method with the exception that the *wait()* method can be interrupted by a call to the *notifyAll()* method. Additionally, the call to *wait()* will make the thread temporarily leave the monitor.

When threads compete for a lock, the thread that obtains the lock will continue executing, while the threads that did not will continue to compete for the lock. There is also a *notify()* method that will only resume a particular thread and leave the rest in a waiting state. We have implemented the *wait()* and *notifyAll()* methods. These constructs allow programmers to develop multithreaded applications that share resources while synchronizing the process.

4.1 Flowpath Threads

The following is an architecture for implementing multiple tasks in hardware. Table 1 shows a list of important definitions. The trivial case of implementing threads in a flowpath is to have multiple flowpaths that have no common dependency executing multiple independent algorithms simultaneously. The flowpaths can be independently synthesized and implemented into partitioned regions of the FPGA, perhaps even using a modular design flow. In this case, each flowpath would actually be referred to as a task according to the definitions in Table 1.

A more challenging case is one in which tasks share common dependencies, shown in Figure 2. The Access Controller will control access to the shared memory by granting or denying a lock on the shared memory. The Access Controller will wait for a lock request, which could come from any number of tasks on the same rising edge of the clock (in parallel) in the form of asserting the *REQx* signals (Figure 3). The Access Controller will grant the lock to the Task *m* with the highest priority. In this case,

the task number corresponds to the task priority, where the lowest task identifier represents the highest priority. The grant occurs in the form of an acknowledgement signal, ACK_m , which gets passed back to the Task m requesting the lock. At this point, Task m will continue execution while the other tasks waiting for the lock will block. The task that holds the lock must explicitly release the lock to allow another task to obtain the lock. This is a cooperative model. If a task never releases the lock, then all access to the shared memory by any other task will block forever. Releasing the lock is accomplished by asserting the $RELM$ signal.

Table 1: Definition of Terms

<p><i>Flowpath</i> – A data path and corresponding control unit that can be synthesized directly to hardware using VHDL or Verilog [11].</p> <p><i>Task</i> – A flowpath and any data that does not share a common dependency with another Task. If the task will be included in a Task Frame then locking logic will be included in the Task</p> <p><i>Access Controller</i> – A device that controls access to a shared memory.</p> <p><i>Shared Memory</i> – A RAM that stores data with a common dependency between two or more tasks.</p> <p><i>Task Frame</i> – A set of Tasks that share a common data dependency, an Access Controller, and a Shared Memory.</p> <p><i>Parallel Flowpath</i> – A set of one or more Task Frames</p>

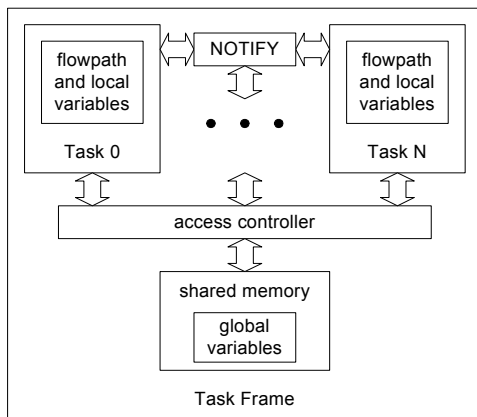


Figure 2: N Tasks with Data Dependency

Three new flowpath OPs are introduced to implement this model: $ReqLock$, $Block$, and $RelLock$. Figure 3 shows the flowpath components for acquiring the lock.

The other tasks will block only if they are waiting to acquire the lock. In any other case, the tasks will continue to execute normally and in parallel. Task synchronization is only required when accessing shared resources. Placing multiple task frames on the FPGA

would result in parallel flowpaths. Figure 4 shows the implementation at the FPGA level.

Given a multithreaded algorithm, each thread must be converted to a flowpath after which the logic for requesting and releasing locks to shared memory would be added to each flowpath. Each flowpath would then be connected to the access controller to facilitate inter-thread communications. Synchronization of the threads occurs with the $NOTIFY$ logic.

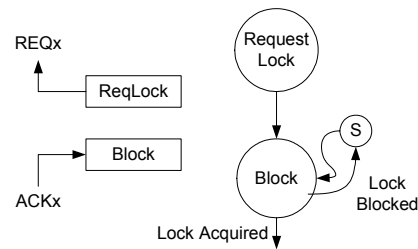


Figure 3: Flowpath Components for Acquiring a Lock

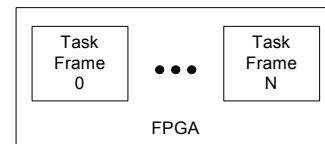


Figure 4: Multiple Task Frames on the FPGA

An example using a typical Producer/Consumer threaded application shows how java bytecodes generated by multithreaded applications can be converted directly to hardware. This is accomplished by adding appropriate OPs to standard Java flowpaths for requesting locks, releasing locks, blocking, and shared-memory access control. The Producer/Consumer example was run on a Personal Computer (PC) with a 600 MHz Pentium M processor running Windows XP with the Sun JVM and a Systonix JStamp with the Ajile-80 Java Processor to compare a CISC and an embedded controller optimized for executing Java bytecodes with flowpaths. Table 2 shows the results from these experiments. The components developed to extend flowpaths to support multithreaded algorithms require minimum space.

This hardware executes multiple tasks in parallel supporting both synchronized and unsynchronized shared memory access. Performance increases occur, in general, since flowpaths created from multithreaded Java programs do not suffer from traditional processor bottlenecks such as context switching, stack manipulation and the traditional load, execute and store operation.

Table 2: Performance of the Producer/Consumer Test

Experiment	# clock cycles*	Absolute Run time (μ s)	Clock (MHz)
Pentium M	314,400,000	851,000**	600
JStamp	80,000	1,085	73.728
Flowpath	51	0.51	100

*clock cycles have been rounded to the nearest thousand
 ** Run time provided by Vtune as absolute time including non-program execution overhead.

5. CONCLUSIONS

This paper has shown how standard Java bytecodes can be synthesized directly to hardware using flowpaths. This includes class instantiation, multithreading, and event handling. These flowpaths are implemented in VHDL and can be realized on an FPGA. Java lends itself well to hardware design; a high-level language that provides broad native support for describing parallel algorithms.

The load-execute-store feature of Java bytecodes maintains the normal microprocessor paradigm. Traditional interpretive Java virtual machines (JVMs) still suffer from the excessive use of local variables in the original software program. Drastic performance increases occur, in general, since flowpaths created from Java programs do not suffer from traditional processor bottlenecks such as stack manipulation, the traditional load, execute and store operation, and context switching for multithreaded applications.

Flowpaths can realize a performance increase of several orders of magnitude when compared to an optimized Java processor directly executing Java bytecodes and execution on a PC. All of the work presented in this paper uses standard Java syntax without any special language annotations.

References

- [1] J. M. P. Cardoso and M. Weinhardt, "From C Programs to the Configure-Execute Model," presented at IEEE Design, Automation and Test in Europe (DATE) Conference and Exhibition, Munich, Germany, 2003.
- [2] D. Soderman, "Implementing C Designs in Hardware," presented at DesignCon98, 1998 International Verilog Conference & 1998 FPGA Configurable Computing Machine Conference, 1998.
- [3] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts, "Handel-C Language Reference Guide," Oxford University Computing Laboratory 1996.
- [4] F. Boussinot, "Reactive C: An Extension of C to Program Reactive Systems," *Software Practice and Experience*, vol. 21, pp. 401-428, 1991.
- [5] Celoxica, "The Technology Behind DK1." Milton Park, Abingdon, Oxfordshire, United Kingdom, Application Note, AN 18 v1.0.
- [6] A. Ye Zhi, N. Shenoy, and P. Banerjee, "A C compiler for a processor with a reconfigurable functional unit," presented at Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2000.
- [7] L. Semeria, "Applying Pointer Analysis to the Synthesis of Hardware from C," Stanford University, 2001.
- [8] J. M. P. Cardoso and H. C. Neto, "Compilation for FPGA-Based Reconfigurable Hardware," *IEEE Design & Test of Computers Magazine*, vol. 20, pp. 65-75, 2003.
- [9] S. A. Edwards, "The Challenges of Hardware Synthesis from C-like Languages," presented at Design Automation and Test in Europe (DATE), Munich, Germany, 2005.
- [10] D. M. Hanna, "A Novel Method for Generation Microprocessor-less Systems with Applications in Bioengineering," in *Department of Computer Science and Engineering*. Rochester, MI: Oakland University, 2003.
- [11] D. M. Hanna and R. E. Haskell, "Flowpaths: Compiling Stack-Based IR to Hardware," *Microprocessors and Microsystems*, vol. 30, pp. 125 - 136, 2006.
- [12] D. M. Hanna and R. E. Haskell, "Implementing Software Programs in FPGAs using Flowpaths," presented at Proceedings of the 2004 International Conference on Embedded Systems Architecture, Las Vegas, NV, 2004.
- [13] P. Haggar, *Practical Java Programming Language Guide*, 1 ed: Addison-Wesley Professional, 2000.
- [14] Systronix, "JStamp: Real-time Native Java Module," 2003.
- [15] "Digital Communications Technologies Lightfoot 32-bit Java Processor Core," Xilinx Alliance Core, 2001.
- [16] U. Brinkschulte, C. Krakowski, J. Kreuzinger, R. Marston, and T. Ungerer, "The Komodo Project: Thread-based Event Handling Supported by a Multithreaded Java Microcontroller," presented at 25th EUROMICRO Conference, Milano, 1999.
- [17] M. Schöberl, "JOP: A Java Optimized Processor for Embedded Real-Time Systems," in *Institut für Technische Informatik*. Vienna: Der Technischen Universität Wien, 2005, pp. 253.
- [18] B. Venners, *Inside the Java Virtual Machine (Java Masters Series)*: McGraw-Hill Companies, 1997.
- [19] M. Campione, K. Walrath, and A. Huml, *The Java Tutorial*, 3rd ed: Addison-Wesley Professional, 2000.
- [20] A. Holub, *Taming Java Threads*, 1 ed: Apress, 2000.
- [21] S. Oaks and H. Wong, *Java Threads*, 3rd ed: O'Reilly, 2004.
- [22] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17(10), pp. 549-557, Oct 1974.