TRANSPARENT COMMUNICATION BETWEEN JAVA AND RECONFIGURABLE HARDWARE

Philippe Faes, Mark Christiaens, Dirk Stroobandt
Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41
B-9000 Ghent, Belgium
email: {pfaes,mchristi,dstr}@elis.UGent.be

ABSTRACT

Reconfigurable computing systems are devices where a CPU controls the system and uses a Field Programmable Gate Array (FPGA) for computationally intensive tasks. Whenever such a task is to be started, the FPGA can be reconfigured and instructed to start the calculation. Such communication is generally based on a message passing interface.

As a new alternative we propose a portable and transparent interface between the Java Runtime Environment and a generic FPGA. From the software side, communication with hardware looks like regular method calls, while the hardware can access fields of Java objects, call methods, throw exceptions, etc.

We feel this interface can ease hardware support for existing applications and allows development of new hardware supported applications in the object-oriented programming paradigm.

KEY WORDS

Parallel algorithms and architectures, reconfigurable architecture, Java, hardware/software co-design

1 Introduction

Reconfigurable computing systems contain a general purpose CPU and some Reconfigurable Hardware (RH), e.g. an FPGA. The main application runs on the CPU and the RH can be used for computationally intensive operations. The CPU can easily be programmed in a high-level language with tools that are very stable and well understood.

While some algorithms cannot be executed fast enough on a CPU many control-driven operations are naturally expressed in an imperative, sequential programming language. Other operations, like massively parallel calculations and advanced signal processing, require dedicated hardware support.

One option would be to use one or more Application Specific Integrated Cicuits (ASICs). However, ASICs are very expensive to develop and impossible to update once they are integrated in a customer's system. As an alternative, designers often choose an FPGA-based solution. FPGAs can be configured to have a behavior equivalent to that of the required ASIC but their configuration can be changed easily, even at run-time. The penalty is that FPGAs work at lower clock frequencies and dissipate more energy. In cases where this penalty is acceptable FPGAs can be used to improve performance of a computing system. Functionality will be targeted at the CPU where possible and at the FPGA where needed.

Many methodologies have been proposed for designing dynamically reconfigurable systems. Some integrate RH tightly with the instruction set processor [5, 14], requiring applications to be compiled specifically for a given reconfigurable platform. Others suggest connecting a CPU and RH to a system bus [9, 10, 12], reconfiguring the RH at predetermined times during the execution.

If each operation of the application has at least a software implementation and some operations have a hardware accelerated implementation the RH reconfiguration can also be decided at run time. The decision can be based on real-time requirements, battery power, resource availability, etc. In [11] such an adaptive system is presented. It is capable of assigning tasks to either the CPU or one of the *tiles* that are defined on the FPGA. In this context a *task* is an independently calculating entity that communicates with other tasks through uniform messages.

As an alternative to a system based on parallel tasks and messages in this article we propose an interface between the FPGA and an object-oriented shared memory runtime environment. Activating a hardware supported operation will look like a normal method call to the programmer and parallelism is implemented as concurrent software threads which may call hardware methods. As a runtime environment we choose the Java Virtual Machine (JVM) for reasons explained in Section 2.

This hardware accelerated JVM can be used in two different types of machines. The first type is *portable systems* like cellular phones and portable digital assistants. To decrease battery consumption portable devices use specific low power CPUs, trading battery power for computational power. However, these CPUs may not be able to perform complex calculations fast enough to meet the system's real-time specifications. Some calculations require less energy

when executed on an FPGA compared to a CPU [3, 13]. By choosing a slower CPU plus an FPGA one can improve the power consumption. This is even possible for downloaded applications. Indeed, many applications that run on portable devices today are not known at design time. They are selected by the user, sometimes only a few moments before their execution.

The second type of devices are *high-end computers* that perform large scientific calculations, real-time simulations and biomedical applications. These calculations can be accelerated by means of an FPGA board plugged into the host computer.

In the next section, we will motivate the rationale for choosing Java as our software development language. In Section 3 we consider a simple use case to compare classical message passing with our proposed transparent interface. In Section 4 we explain how we intercept Java method calls in order to delegate them to hardware. The low-level communication and the high-level communication protocol are described respectively in Sections 5 and 6. Finally we present our results in Section 7.

2 Why Java?

We choose Java as our research platform for a number of different reasons.

First of all, the Java platform uses bytecode as an intermediate representation. This bytecode is either interpreted by a JVM or it is translated to native machine code by a Just In Time (JIT) compiler at run-time. We can easily alter the behavior of a Java program by making changes to the JVM or the JIT or even to the bytecode at load time. The behavior of native machine code is much more difficult to manipulate correctly or even to understand. At bytecode level it is guaranteed that there will not be any operations outside the current stack frame, there are no accesses out of array bounds, no pointer arithmetic, no writes to a `final` (i.e. constant) variable, etc. In general: life is a lot easier than it is with native machine code. Many of these guarantees can be made because the classloader verifies new bytecode before linking it with the already loaded classes [8, sec 4.9]. Other guarantees are enforced by the JVM through run-time checks.

Another advantage of bytecode is that objects and methods are clearly visible. In native machine code any word-sized number could be a pointer to an object of some unknown class, and methods might be made invisible by inlining.

Most modern JVMs have a JIT compiler that translates valid bytecode to machine code. The idea is that the machine code will execute faster than the interpreted bytecode. When a class is first loaded, the JVM might first interpret the bytecode. After a while it may decide that the performance gain of a JIT-compiled version of the class will be worth the compilation overhead. Likewise a second optimized compilation may be performed to improve execution speed even more. We suggest it should be possible for the JVM to decide to change the execution from JIT-compiled native code to FPGA-supported execution. Again this will induce an extra overhead of downloading the FPGA configuration from some repository into the FPGA, but it can improve total execution time because the FPGA version will be faster than the JIT-compiled version.

Similar advantages can be expected when the techniques presented in this paper are applied to other virtual machines with an intermediate representation.

3 Transparent communication: a use case

One major design goal is to make hardware support completely transparent to the Java programmer. Mignolet et al. [11] and Ha et al. [4] have proposed special message passing interfaces for communication with hardware. All communication with the hardware accelerated object is done by passing messages. Therefore, the system designer is forced to use a message passing interface for every computation that will ever be accelerated in hardware.

Suppose we have a system that contains a Finite Impulse Response (FIR) filter in RH. With a traditional message passing interface the data that needs to be filtered has to be explicitly wrapped in messages and sent to the filter. After the operation finishes, the resulting filtered data is again wrapped in messages and sent back to the CPU.

As opposed to a message passing interface we propose transparent communication between the CPU and the RH. When the Java programmer wants to start a calculation he just calls a Java method, as in Figure 1. If that calculation happens to have hardware support the method call will be intercepted and instead of executing the method the RH will be activated. The RH will communicate with the hardware support system in the JVM and the data to be filtered and the filter coefficients will be copied to the RH. After the filter operation is complete the JVM will create a new array object and initialize it with the filtered data from the RH. This array is then returned to the calling method.

The decision which data needs to be copied to and from the RH is no longer made by the system programmer. Instead, the hardware support system copies data between CPU and RH according to the needs of the algorithm. If for instance the hardware implementation wants to use only the first n filter coefficients, there is no need to copy all coefficients. We let the RH decide which data needs to be copied thus avoiding unnecessary data transfers.

4 Intercepting calls

As we explained in the use case, the Java programmer uses normal method calls, even if a method will be executed by RH and not by the CPU. We need to intercept those method calls and check whether we can delegate the calculation to hardware.

```
someMethod(int[] data) {
  ...
  // this.fc contains filter coefficients
  filtered = this.firFilter(data);
  // Will be intercepted and executed by RH.
  // RH will request this.fc from the JVM
  ...
}
firFilter(int[]data) {
  // If no hardware support exists,
  // this method will be executed.
  ...
}
```

Figure 1. Transparently calling a FIR filter

One way to accomplish this is to make a derived class that overrides some methods with specific hardware communication. This makes method calling transparent, but object construction still is not transparent. The entire application would have to be adapted to call the constructor of the derived class with hardware support, instead of the parent class without hardware support.

A cleaner solution is to change the behavior of the class at load time. This is done using Aspect-Oriented Programming (AOP) techniques [6]. Whenever a class is loaded, the JBoss-AOP [1] classloader checks if it has to inject the "hardware acceleration aspect" into some of the class' methods. If so, the bytecode is altered. Whenever this altered method is called, a special routine will be started instead of directly executing the original method. This routine checks for hardware availability and starts the calculation in RH. If no hardware support is available the original method will be started.

Any method of any Java class can now be hardware accelerated without changing the Java classfiles. This is important if we want to hardware accelerate classes that cannot be changed e.g. because their licenses do not allow modification or because they are fetched from a server we cannot control.

5 Low level communication

Now that we are capable of intercepting calls to a Java method, we need to determine how we can set up communication between the JVM and the FPGA. The FPGA will be connected to a system bus. Our research platform consists of an Altera Stratix EP1S25 FPGA on a PCI board, plugged into a desktop PC. Some of the CPU's address space will be mapped to registers in the FPGA. This makes it easy for software to communicate with the FPGA; it will only need to write to or read from memory locations.

In machine code or languages like C or C++ this is indeed easy. However, Java does not allow access to raw memory. We can work around this by using the Java Native Interface (JNI)[7], which will allow us to call C functions in Java. The JNI is not very fast because for every word of data we need to access on the FPGA we need to cross the Java/C boundary. As an alternative we can adapt the virtual machine to enable direct Java access to certain memory regions. This is surprisingly easy in the Jikes Research Virtual Machine, formerly known as Jalapeño [2]. Since it is programmed mostly in Java the JikesRVM allows Java to access low level system calls and raw memory. Using the JikesRVM communication between Java and the FPGA will be just as fast as communications between native machine code and the FPGA.

Table 1. Possible requests

JVM to FPGA	FPGA to JVM
HW method invocation	HW method return
answer to request	SW method invocation
	lock
	unlock

The FPGA can react to read or write operations by starting a computation. When the FPGA finishes a computation or wants to request some service from the JVM it can notify the CPU in two different ways. A value can be set in a memory mapped register. The JVM is expected to poll this register from time to time and respond to any request that is encoded in the register.

Alternatively the FPGA can send an interrupt to the processor. As a response to that interrupt the JVM will read one of the FPGA's registers and respond to it. Each alternative has its advantage. Polling is better for low latency but CPU cycles are wasted. Handling interrupts has a higher latency but the CPU can devote all its time to useful calculations.

6 Communication protocol

At this point, a communication protocol needs to be defined. We support six different types of signals, as pointed out in Table 1.

There are two different signals that can be sent from the JVM to the FPGA: a *method invocation*, and an *answer* to a request from the FPGA. We will discuss the method invocation here. The answer will be discussed when we get to the FPGA's requests.

To support hardware method invocation each method that can be hardware accelerated has its own set of registers in the FPGA. Each of the method's arguments has a register and there is an extra register to start the invocation. The object to which the method should be applied (the `this` object) is considered to be the first, implied, argument. Each argument (and return value) of a method is passed through a fixed size register of 64 bits. The largest Java primitive values (`long` and `double`) fit into these registers. Objects are passed by reference, as Java always does.

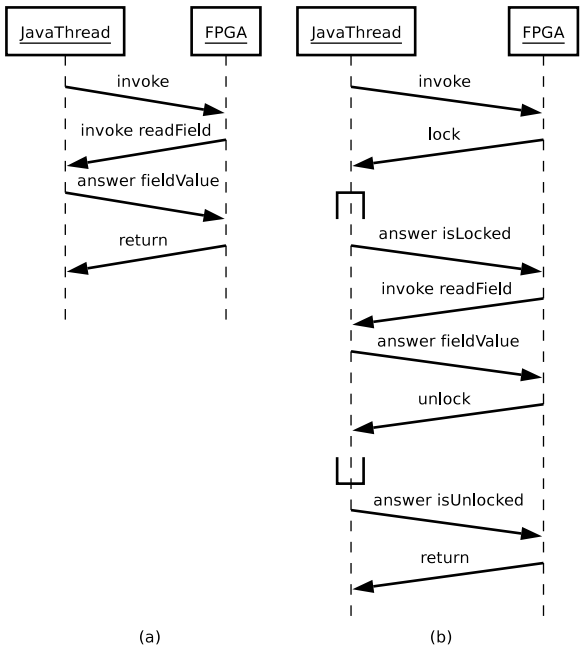

Figure 2. Simple requests from FPGA

When the FPGA finishes its calculation based on the arguments it received it sends a *return* request. This request is passed to the JVM either by polling or with an interrupt, as explained in Section 5.

The *return* request is the first of four primitive requests the FPGA can send. The second request is a *software method invocation*. Any Java method can invoke another method. Therefore we also allow a hardware accelerated method to request a software method call (which may in turn be delegated to hardware). In order to request a software method invocation the FPGA sets some of its registers to encode a reference to the requested method and its arguments, including the implied `this` argument. Since nested Java methods are always executed in the same thread we want the software method invocation to be handled by the thread that started the original hardware method. If we ignore this requirement we cannot guarantee that the thread's local copies of data be the same for the software version and the hardware accelerated version. This violates the Java memory model.

We created a special class that implements a number of interesting methods the FPGA may want to call. They include methods for reading and writing fields, constructing objects and arrays, manipulating arrays, throwing exceptions etc. Whenever we come up with new things that are available in Java, but not in hardware, we just add new methods to make them easily available to the FPGA. Figure 2(a) shows how a method in hardware can read a field from an object.

There are, however, two things that cannot be done in a method: object *locking* and *unlocking*. Java demands that

```
void someMethod() {
  synchronized(o1) { // lock o1
    synchronized(o2) { // lock o2
    doSomethingUsefull();
    } // implied unlock o2
  } // implied unlock o1
}
```

Figure 3. Java locking syntax

locking and unlocking operations are always done in nested pairs. The Java language syntax makes it even impossible to lock an object without unlocking it in the same block, as can be seen from Figure 3.

In order to solve this problem, we do not return from the locking method. Instead, we enter a `synchronized` block and send an answer to the lock request from within that block. All following requests are also handled within the same block, until the FPGA requests an unlock operation. To allow nested locking, we do not actually duplicate code inside the block. Instead we recursively call the method that handles new hardware requests. Now it is possible for a hardware method to lock an object, read a field from it and then unlock it as is depicted in 2 (b).

Notice that it is illegal for the FPGA to send a *return* request when there are still objects locked. The request handling method will check for such illegal requests, and report them as errors.

7 Results

As a first experiment we added instructions to the bytecode as described in Section 4. We used the JBoss-AOP library, and measured the execution time of a method with an empty body. Without modification, one execution of the method takes 20ns, and with modification 7.56 μs. Measurements were conducted on a Sun Hotspot virtual machine 1.4.1 (*blackdown* build) on a Pentium III, 300MHz running the Linux 2.6.3 kernel.

We expect to be able to greatly reduce this overhead by using a different library for the bytecode modifications. However, there will always be a performance penalty which will only be compensated if long methods are selected for hardware acceleration.

In the second experiment we measured the execution times of two methods in software and in hardware. The first method reads the same field one thousand times and the second takes a thousand locks on an object and then releases the locks. Normally one would avoid communication between RH and the JVM as much as possible because it is very slow. In systems which have two CPUs the communication overhead between the two processors is also an important issue. Since we communicate over a PCI bus at 33MHz and the PCI arbiter in our test computer allows us only one read transaction every 20 cycles, communication

Table 2. Execution times per operation in μs

		locking	read
(a) software	N/A	1.5	1.3
(b) hardware simulated in Java	polling	1982	1057
(c) FPGA on PCI bus	polling with extra thread	4069	2009
(d) FPGA on PCI bus	polling	80.9	56.9

is a major bottleneck. We intend to move to a Processor Local Bus (PLB) on a Xilinx Virtex II-Pro. This bus can be clocked as fast as 400MHz and is designed for better throughput and response time than PCI.

Table 2 shows the execution times respectively per lock/unlock pair and per read operation. The hardware simulation (b) runs significantly slower than the real hardware (d) even though the simulation routine is very simple. A lot of the delay is due to switching between the simulated hardware thread and the thread that invokes the hardware method. The Sun JVM uses the operating system's threads which are not very fast in Linux.

We also experimented with a system that has an extra polling thread (c). The threads that call a hardware method will sleep after invoking the method and will be woken up by the polling thread. This experiment gives us an idea of the overhead we can expect when we have one designated thread that handles interrupts and wakes up other threads as required. The measurements show that such indirect notification will be very slow. Perhaps interrupt demultiplexing should be done by a kernel space device driver and not by a Java thread. As another possible course of action the polling could be integrated in the scheduler of the JikesRVM. Obviously, we would need to be careful not to degrade scheduling speed by polling the FPGA too often.

We see that communication between Java and FPGA is expensive, even in our fastest hardware implementation (d). FPGA support is only recommended for longer calculations that require little or no extra communication with the JVM.

8 Conclusion and Future work

We presented a flexible and transparent interface between reconfigurable hardware and the Java Virtual Machine. Even though the communication overhead is significant we believe that for applications with few communications and long calculations a large performance gain can be achieved. We will further optimize the communication overhead and the overhead of the injected aspects.

To demonstrate the usability of this interface we will implement a full medical signal processing application. To facilitate hardware development for Java acceleration we will propose a methodology for converting a Java method to an FPGA configuration.

Acknowledgements

This research is supported by grant 020174 of the Institute for the Promotion of Innovation through Science and Technology in Flanders (*IWT-Vlaanderen*), and by GOA project 12051002 of Ghent University. Philippe Faes is supported by a scholarship of the *IWT-Vlaanderen*.

References

[1] JBoss webpage. http://www.jboss.org .

[2] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] J.-D. Bakker, K. Langendoen, and H. Sips. LART: Flexible, low-power building blocks for wearable computers. In *Proc. Int. Workshop on Smart Appliances and Wearable Computing (IWSAWC), Scottsdale, AZ*, Apr. 2001.

[4] Y. Ha, G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and H. De Man. Virtual Java/FPGA interface for networked reconfiguration. In *Proceedings of the ASP-DAC 2001 Asia and South Pacific Design Automation Conference*, pages 558–563, Jan. 2001.

[5] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97)*, pages 24–33, Apr. 1997.

[6] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland*, pages 220–242. Springer-Verlag, June 1997.

[7] S. Liang. *The Java(tm) Native Interface Programmer's Guide and Specification*. Sun Microsystems.

[8] T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification*. Sun Microsystems, Inc., second edition, 1999.

[9] B. Mei, P. Schaumont, and S. Vernalde. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. Nov. 2000.

[10] B. Mei, S. Vernalde, H. De Man, and R. Lauwereins. Design and optimization of dynamically reconfigurable embedded systems. In *Proc. 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 78–84.

[11] J.-Y. Mignolet, V. Nollet, P. Coene, D.Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 986–991, Apr. 2003.

[12] A. Pelkonen, K. Masselos, and M. Cupák. System-level modeling of dynamically reconfigurable hardware with SystemC. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 174–174, Apr. 2003.

[13] R. Scrofano, S. Choi, and V. K. Prasanna. Energy efficiency of FPGAs and programmable processors for matrix multiplication. In *The First IEEE International Conference on Field Programmable Technology (FPT)*, pages 422– 425, Dec. 2002.

[14] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Filho. MorphoSys: A reconfigurable architecture for multimedia applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.