Hardware/Software Co-Design of a Java Virtual Machine

Kenneth B. Kent
University of Victoria
Dept. of Computer Science
Victoria, British Columbia, Canada
ken@csc.uvic.ca

Micaela Serra
University of Victoria
Dept. of Computer Science
Victoria, British Columbia, Canada
mserra@csr.uvic.ca

Abstract

This paper discusses the initial results of research into the development of a hardware/software co-design of the Java virtual machine. The design considers a complete Java virtual machine with full functionality expected to run with the same capabilities as a fully software Java virtual machine. We address issues such as why a partial hardware implementation is suitable, the challenges in realizing this goal, propose an initial partitioning of the virtual machine between hardware and software, discuss the desired hardware requirements and discuss some details of the hardware and software design.

1 Introduction

The Java platform as introduced by Sun Microsystems in 1994 has spread throughout the computer industry and has reached all domains. As good as Java is for providing "write once, run everywhere" software, the running is not always very good. The Java platform still relies upon software interpreters to execute and accordingly suffers in performance.

Since its introduction, many people have directed resources towards moving Java away from the interpreter and into hardware. The idea is not to decrease the features or functionality of the virtual machine, but to increase its performance. This can be accomplished in any one of three ways: (i) create a general microprocessor that is optimized for Java, yet still functions as a general processor; (ii) make a stand-alone Java processor that runs as a dedicated Java virtual machine; or (iii) create a Java co-processor that works in unison with the general microprocessor.

This project makes advancements towards accomplishing the co-processor approach to work in unison with a general microcontroller to increase Java performance where the co-processor is based on programmable hardware (FPGA). This paper will address the following steps in realizing this goal:

- Justification for choosing the programmable co-processor solution over the other possibilities.
- Research into the Java platform to uncover major problems to overcome during the development.
- Partitioning of the Java virtual machine between software and hardware based upon statistics of what will help increase the performance and make for an easy design.
- Analysis of the partitioning to form a specification of the desired hardware and software requirements to meet the initial partitioning and design, and as well any optional requirements that will help in the development process.
- Description of the hardware and software design for the partition that meets the requirements.

2 Co-processor Idea

Our goal is to provide a full Java virtual machine for a regular windows workstation. For the purposes of this research, we are not proposing to attach a co-processor to the mainboard of a system. Rather, the co-processor should be accessible to the system through one of the many available system busses. This will allow for efficient design research and testing of various configurations and optimizations. This is due to the availability of field programmable gate array (FPGA) cards that can be added to a system. One of these cards can be used to perform verification and testing of the design, while no such hardware is available to allow for a reconfigurable hardware unit on the system mainboard. However, any research results that are obtained will apply just the same if the co-processor is attached to the mainboard directly. The single difference is that the connection between the co-processor and the host processor will be slower. The details of the reconfigurable hardware card are discussed in section 6. An additional advantage of this approach is that the reconfigurable co-processor can be used for other hardware acceleration applications as needed, exploiting indeed the true power of FPGA boards.

Here we focus on Java, but the research will yield a general framework.

3 Design Issues

Some interesting design issues include:

- Addressing multiple Java applications executing within a single Java virtual machine.
- Communication between the hardware and software components of the design.
- Utilizing resources that are available, both resources that are limited and new resources which were not previously available to a fully software Java machine.
- Upholding the Java model, both supporting the full Java API and providing the same level of security to Java programs.
- Managing the scarce memory resources to ensure both consistency of memory across the different levels of cache and correct diligent garbage collection for faster execution.
- Utilizing the potential for concurrent execution in hardware and software.
- Designing and implementing with the realization that the Java model and specification will grow and change to allow the design to grow with the ever-changing specification.

Each of these issues make this project unique and challenging, but most importantly they demonstrate the need for research on a rapidly reconfigurable platform.

4 Justification

Before the justifications for implementing the Java virtual machine as a co-processor, it must be argued as to why we should implement the Java virtual machine in hardware at all. There are several key ways in which Java can be "souped up" [3][9]. The performance enhancements include such techniques as: better source compilers, bytecode optimizers, better Java virtual machines, just-in-time compilers, adaptive compilers, static native compilers, native method calls, Java chips, and better source code. Each of these can bring gains to the performance of Java, and some of them can even work in unison with a hardware Java machine to bring a combined performance increase. Such techniques as writing better source code, using better source compilers and using bytecode optimizers can provide an increase in performance whether a Java application runs on a hardware Java machine or a virtual one.

The use of just-in-time, adaptive and static native compilers, as well as using native method calls, brings performance gains, but it does not provide the overall complete solution that a hardware implementation would provide. In the case of using native method calls or static native compilers, the solution sacrifices the portability feature of Java. Users will have to contend with more than just the "download and execute" paradigm that was one of the original selling points of Java. Any of the compiler solutions will be hard pressed to provide the same level of performance increase that a Java chip can bring due to the fact that current processors today are not based on the same stack based architecture with which Java was designed. This will make for interesting problems in achieving solutions for a mis-matched mapping.

All of these techniques offer potential for increased performance of Java execution, however none of them offer the same potential levels of increase as that of a Java virtual machine implemented in hardware.

4.1 The Hardware Justification

Different hardware solutions have their merits and flaws. The Java co-processor has the most appealing value in that it does not replace any existing technology; instead it supplements current technology to solve the problem. Similar to the math co-processor of old, the Java co-processor upon proving its merit can be integrated into the general-purpose microcontroller.

In comparison to a stand-alone Java processor, this solution provides greater flexibility to adapt to future revisions to the Java platform. Since its birth Java has experienced changes in all areas. The API is constantly changing and, with it, the virtual machine itself has changed and will continue to change (for example, with better garbage collection techniques being devised). With a reconfigurable co-processor, changes in Java can more easily be integrated and made readily available. If the technology were part of the main processor, this would not be as easy a task. In addition, providing a Java-only processor will just change the problem at hand, not fix it. The tables will be turned and Java applications will run fast, while C and other programming languages will be suffering from decreased performance of having to execute through a non-native processor. With the different execution architecture paradigms, a simple solution will not be available.

If Java were to be incorporated into the main processor unit, there would have to be some trade-offs between execution for Java and for legacy programming languages such as C. Surely some of these trade-offs will make it difficult to provide optimizations for execution within the processing unit. Wayner says: "An advantage for Java chip

proponents is how complex it is to design a chip for fast C and Java code performance" [10]. To design a viable chip for both is complex since users will definitely not want to see a decrease in the performance of their current applications to see an increase for Java applications.

The Java co-processor solution also has the benefit of choice. With it available as an add-on card, systems that are not required to provide fast execution of Java can simply continue using a fully software solution. Systems that do require fast Java execution can plug in the card and increase performance without having to replace any of their current components or more drastically having to move to another system all together. As seen with other similar products such as video accelerators, this is the preferred solution for consumers. Finally, the plug-in card can be used as a co-processor for other applications, given its reconfigurability.

5 Partitioning and Design

The Java virtual machine is comprised of two parts: a low level instruction set from which all the Java language can be composed, and a high level operating system to control flow of execution, object manipulation, and device controllers. To partition the Java virtual machine between hardware and software the first step is the realization of what choices are to be made. Since part of the virtual machine is high level operating control, it is impossible to put this work into the hardware partition due to the restrictions of hardware. This leads to investigating the instruction set of Java to determine what is capable of being implemented in hardware.

5.1 Software Partition

The instructions that must remain in software are those designed for performing object-oriented operations. These include instructions for accessing object data, creating object instances, invoking object methods, type checking, using synchronization monitors, and exception support.

Each of these object-oriented instructions requires support that is too extensive to be implemented in hardware since they need class loading and verification. Loading and verification involve locating the bytecode for a class, either from disk or a network, and verifying that it does not contain any security violations. Once the bytecode is verified, if the instruction requires creation of an object then the creation may require accessing the virtual machine memory heap and the list of runnable objects. This process requires complex execution and a significant amount of communication with the host system. As such, it is better to execute the instruction entirely on the host system than within the Java co-processor hardware.

Exceptions are a very complex mechanism to implement in any situation. The reason for this is the effects that an exception can have on the calling stack and the flow of execution. Within the virtual machine it could involve folding back several stack frames to find a location where the exception is finally caught. An exception in Java also involves the creation of an Exception object that is passed back to the location where the exception is caught. This can result in class loading and verifying as part of the exception throwing process. As a result of this potential complexity, the exception instructions should be implemented in software where manipulating the execution stack is more easily performed.

5.2 Hardware Partition

For each instruction it is obvious that if more can be implemented in hardware the better it is, since the overall purpose of this design is to obtain faster execution. Additionally, all instructions can be implemented in software, as shown by current implementations of the Java virtual machine. So for a preliminary investigation, the research entails determining if an instruction can be moved from software to hardware. We look at grouping of instructions to be implemented in hardware with a brief explanation for why the decision was made.

Some of the instructions that exist in the Java virtual machine are instructions that can be found in any processor. As such there is no question that these instructions can be implemented in the hardware partition of the Java machine. These instructions include: constant operations, stack manipulation, arithmetic instructions, shift and logical operations, comparison and branching, jump and return, and data loading and storing instructions. Some of these instructions also include instructions that are typically found in a floating-point unit co-processor.

In addition there are other Java specific instructions that can be implemented in hardware. These instructions are mostly the *quick* versions of the object-oriented instructions of Java that distinguish the hardware co-processor from a general microprocessor. These instructions are used for creating new objects, accessing synchronization monitors, invoking object methods, and accessing object data. Once these instructions are invoked upon an object, subsequent calls can use the *quick* version that does not require class loading or verification. It is the implementation of these instructions in hardware that can contribute to the hardware speed-up of Java.

6 Development Environment

With the partitioning of the Java machine between hardware and software investigated, we can discuss the development environment that will be needed to support the research. From the requirements and specifications it can be seen that our concerns lie with the memory available, the size of the FPGA, and the layout and capabilities of the data paths. We believe that a commercially available board, with minor customization, is able to support the full design, following our specifications below.

6.1 Memory Requirements

The random access memory available on the FPGA card must be used to hold the virtual machines data stack, and its execution stack. These are the minimal items that need to be held in memory. It would be beneficial if the card were able to house several sets of data and execution stacks for different Java processes that are executing. This would significantly speed-up the context switching between processes since no communication with the host system would be necessary to load/store a process context. It would also be beneficial if the memory were large enough so that it could hold Java classes in memory that have already been verified and resolved by the software partition. Thus any references to classes would not result in a stoppage for the class to be retrieved from the host processors memory. From a conservative estimate, our FPGA card contains no less that 4 Mb of memory for holding in the minimal case the data and execution stack of a process.

In addition to RAM we require programmable read-only memory to hold some of the base Java classes that are used most often. Thus, verification and loading of these classes can be skipped altogether when they are referred to in the Java program, as there is no reason to fear a security problem of having the classes corrupted. Obviously, the larger the ROM the more base classes can be fit in. We plan to house the minimal and more common java.lang, java.util, and java.math just to name a few.

6.2 FPGA Requirements

It is hard to determine the required size of the FPGA that will hold the Java hardware design since we are currently only at the design stage. The required size is better determined after some initial implementation time is spent. As such, any estimates that are made on the required FPGA size are just that, estimates. With the amount of knowledge about what we would like to house in the hardware partition, and the possibilities of shifting some features from software to hardware and vice-versa, for the purposes of development we want the largest FPGA we can find. There is a lot of potential for experimentation once the implementation is completed by shifting different functionality to and from hardware and software. By getting a large FPGA it will be possible to test different configurations to find the best solution.

6.3 Data Bus

The size of the bus between the host processor and the FPGA card as well as the layout of how each of the required components on the card communicate with one another is important to our design and implementation. For the data bus between the host processor and the FPGA card, it is desirable to have a bus that is either 32 or 64 bits wide, since the Java virtual machine is a 32-bit architecture and the majority of data within the machine is this width.

Within the FPGA card, our data is traveling mostly from all of the different memory that is available to the FPGA, and then from the FPGA back out to the memory. Having a layout on the card that supports this directional flow of data allows for optimal communication on the card. It is also desirable for the memory on the card to be accessible through the main bus connecting the card to the host processor. This allows for data transfers between the card and the host processor without having to be routed through the FPGA, thus allowing for both data transfer and computation simultaneously.

7 Hardware Design

After some investigation of how Sun Microsystems developed their picoJava processor, some ideas were unveiled regarding features that could just as easily be implemented in the co-processor hardware partition that will allow for increases in performance [5][6][7][8]. The block diagram of the design of the hardware partition implemented on the FPGA is depicted in figure 1. The shaded logical blocks are configurable in size and are not required. Within the block diagram, all connections are 32 bits wide with one exception. This is due to the fact that Java is built on a 32-bit architecture. The exception is the connection between the Stack cache and the arithmetic units that is 96 bits. This allows for long operands to pass from the cache to the arithmetic units in a single cycle.

The logical blocks that are unique to this design are the host interface, and the host controller. The host interface provides communication between the FPGA and the components on the board, as well as communication with the host system. The interface is connected inside the FPGA to both the instruction and data caches as well as the host controller. Instructions for execution flow through the interface and into the instruction cache for execution. Data

from the execution of the program flows bi-directionally through the interface between the data cache and the RAM that is on board the PCI card. Most importantly, the I/O interface provides a means by which the host controller can interact with the host system.

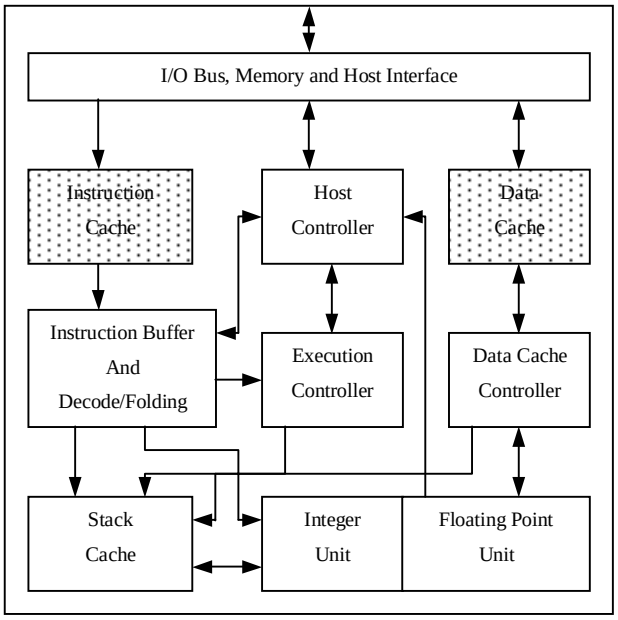

Figure 1: Co-processor hardware design

The host controller maintains the link between the host system and the hardware. It is responsible for halting the hardware in the event that execution needs to stop while the host system carries out part of the execution. In addition, it is responsible for changing the context of the current execution when signaled by the host system. In essence it is the hardware mediator between the hardware and software.

7.1 Memory Interaction

Between the Java co-processor, which is in the FPGA, and the memory that is available on the board, there must be some interaction. The memory on board the card is used as an intermediate memory location between the host system and the Java co-processor. As the host system resolves classes and sets the location of execution with the Java co-processor, it places into the cards RAM, the class itself and the instructions that are located at the address of execution. The Java co-processor through its I/O interface retrieves data from the memory and brings it into the instruction cache on the FPGA for execution. The memory must also contain any data that cannot fit into the data cache and is used too often to be swapped back out onto the host system.

To handle the amounts of data that are being transferred back and forth between the host system and the card, in the event that the memory on board is not sufficient enough to hold all the application and necessary data generated, the memory must be split into manageable blocks to allow for quick and efficient transferring. As blocks are transferred into the memory they are flagged as being in use by the Java co-processor, and as they are used and determined to not currently be in use they are flagged appropriately and cached by the host system the next time data needs transferring to the Java co-processor. Further research may uncover a better solution to marking blocks of memory that can be swapped out. It is worthwhile to note that the garbage collector will free any memory within the blocks that are no longer in use, but is not involved in the caching of data between either of the memory caches.

Blocks of the memory are not labeled to hold either data or instructions. The reason for this is to allow the application the ability to dictate how the memory is used. In the case where an application is very small and uses a lot of data, more blocks will be used for data handling. Conversely, when an application is large and the amount of data storage it uses is small, more blocks of memory will be used to hold the application. This allows for adaptability of the memory to the particular application.

8 Software Design

The software partition is co-designed with the hardware so as to provide a seamless interface between the two partitions in the final implementation. The software partition is designed with the intention of not controlling the execution, but rather supporting the execution of the application by the Java co-processor. Accordingly, the functionalities of the software tend towards passive execution except in the instances where the Java co-processor requests the assistance of the software. Figure 2 depicts the block diagram of the software design with respect to the threads of execution that are necessary in the software to support the Java co-processor. Each of the threads in the figure are standard threads within the Java virtual machine except for the hardware handler that is added for communicating with the hardware partition.

The key component comprising the interface on the software side is the hardware handler. This thread is a passive thread that waits for requests either from the hardware or software partitions so that it can carry the request through the entire Java machine. Requests from the software partition are received from the Scheduler and application threads. For a thread to get a time slice it is passed to the Java machine for execution. During this context

switching, the hardware handler is responsible for saving the context of the previously executing thread and passing the new context to the host controller for loading into the Java co-processor. In addition to invoking context switching within the Java hardware, the hardware handler must also receive events from application threads. These events will often result in context switches to user threads that will handle the event. There is no difference in the handling of the context switch; it is just generated by a different thread. Part of the context switching control also requires the hardware handler to be able to halt and resume the execution of the Java hardware.

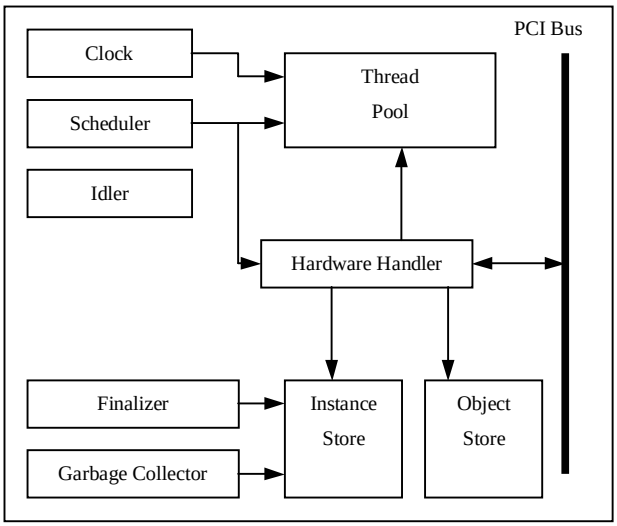

Figure 2: Software partition design of co-processor

The hardware handler is also an essential component in the communication of data between the Java co-processor and the software partition. If the Java hardware were to support a ROM on board the card to hold standard Java base classes that can be trusted, then this thread will be responsible for retrieving the classes from the FPGA card in the event that the class loader in software wants to load one of the base classes. The thread must also be used to maintain the synchronization of data between the memory on the FPGA card and the memory located on the host system. Due to the many important duties of its thread, it must be implemented in a very efficient manner to promote fast execution.

9 Conclusions

This paper discusses the idea of a hardware/software co-designed Java virtual machine. The paper discusses some of the design issues that must be conquered through the project and the justification of why the virtual machine should be implemented in this fashion. Following this we described the partitioning of the design, the development environment for our research work, and lastly some of the finer details of the co-design. This research demonstrates that a Java co-processor is a practical and effective solution to Java's performance problems.

References

[1] El-Kharashi, M.W. and ElGuibaly, F. *Java Microprocessors: Computer Architecture Implications*. PACRIM 1997.

[2] El-Kharashi, M.W.; ElGuibaly, F. and Li K.F. *Quantitative Analysis for Java Microprocessor Architectural Requirements: Instruction Set Design.* International Conference on Computer Design, 1999.

[3] Halfhill, T.R. *How to Soup up Java (Part I).* BYTE, May 1998.

[4] Silberschatz, A. and Galvin, P.B. *Operating Systems Concepts (4th edition).* Addison-Wesley Press, June 1994.

[5] Sun Microsystems. *The Java Chip Processor: Redefining the Processor Market*. Sun Microsystems, November 1997.

[6] Sun Microsystems. *picoJava-I: Java Processor Core.* Sun Microsystems Data Sheet, December 1997.

[7] Sun Microsystems. *picoJava-I: picoJava-I Core Microprocessor Architecture*. Sun Microsystems white paper, October 1996.

[8] Sun Microsystems. *picoJava-II: Java Processor Core.* Sun Microsystems data sheet, April 1998.

[9] Wayner, P. *How to Soup up Java (Part II): Nine Recipes for Fast, Easy Java*. BYTE, May 1998.

[10] Wayner, P. *Sun Gambles on Java Chips*. BYTE. November 1996.