Context Switching in a Hardware/Software Co-Design of the Java Virtual Machine

Kenneth B. Kent and Micaela Serra
University of Victoria
Dept. of Computer Science
Victoria, British Columbia, Canada
{ken,mserra}@csc.uvic.ca

Abstract

This paper introduces the idea of using a field programmable gate array (FPGA) in a hardware/software co-design of the Java virtual machine. The paper will discuss the partitioning of instructions and support for the virtual machine. Discussion will follow concerning the context switching between the two partitions and its importance. Several algorithms are described and analyzed with results from several benchmarks used to assist the discussion. The paper will conclude with a decision on the most suitable algorithm and the justification.

1. Introduction

Since its introduction, many people have directed resources into moving Java away from the interpreter to increase performance. Adding hardware support for the virtual machine has been seen as one means to attain the performance increase. This idea can be accomplished in either 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 [1,2,8-10]. This work makes advancements towards accomplishing the co-processor approach to work in unison with a general microprocessor to increase Java performance [6]. This paper will address the following steps in realizing this goal:

- Justification of why choosing the co-processor solution over the other possibilities is a better solution.
- Describe the partitioning and the instructions that are supported in each of them.
- Discuss the context switching problem and the various algorithms that will be examined for a solution.
- Present a comparison of the results of each algorithm and assess their suitability.

2. Co-Processor Idea

When discussing the approach of implementing the Java virtual machine as a co-processor, what exactly is being implied? Our goal is to provide a full Java virtual machine for a desktop workstation [7]. 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 through one of the many system busses, figure 1. This will allow for efficient design research and testing of various configurations and optimizations. This is easily attainable due to the availability of field programmable gate array (FPGA) add-on cards that can be added to a system. One of these cards can be used to perform verification and testing of the design. Any research results that are obtained will apply just the same as if the co-processor was attached to the mainboard directly. The two significant differences between the co-processor being on an add-on card as opposed to being on the mainboard is the slower communication connection and more importantly the two distinct regions of memory, as opposed to sharing a common memory space. 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, yet the research will yield a general framework for other virtual machines and other applications.

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. This approach also allows for easier research. At this point in time there has not been enough research performed to show that either of the approaches can provide sufficient gains in performance to justify the costs.

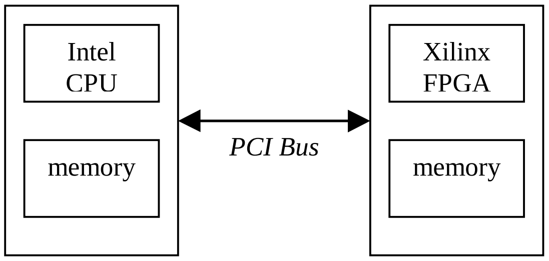

Figure 1: Hardware Architecture

Possibly the greatest advantage of this solution is that there will now be two processing units available for execution. With some scheduling techniques the virtual machine can take advantage of this by using both units together in parallel, potentially rendering better performance than other Java acceleration techniques, such as JIT's. The parallelism may also provide a better platform by which real-time support can be added to the virtual machine, when it is later incorporated into the Java platform.

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 with better garbage collection techniques being devised. With a reconfigurable co-processor, changes in Java can be more easily 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 and not fix it. Simply 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" [11]. 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.

3. 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 transfer this work into hardware due to its restrictions. This leads to investigating the instruction set of Java to determine what is capable of being implemented in hardware.

3.1 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 cannot 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 calling stacks 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 are implemented in software where manipulating the execution stack is more easily performed.

3.2 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 as to 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, type casting, 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. It is these instructions that differ the Java hardware co-processor from other microprocessors. 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.

4. Context Switching

With the addition of a second processing unit, there is the burden of determining when and if the execution should be moved from one unit to the other. Since the architecture has two distinct memory systems for each processing unit, the cost of a context switch from one unit to another is high due to the penalty in transferring the necessary data between memory subsystems. With this high cost, it is necessary to only perform a context switch in instances where the performance gain of making the transition will result in a significant gain that out weights the cost of the context switch. If the penalty for context switching is too high in comparison to the gains of hardware execution, it will be necessary to reduce the context switching latency by moving the FPGA closer to the host system. This can be done by connecting the FPGA to a faster dedicated bus, or by placing the FPGA directly onto the host mainboard. The next section discusses several algorithms that were used to perform a run-time analysis of the Java bytecode to mark appropriate locations where performing a context switch is worthwhile. Currently, this analysis needs to be done at run-time since any changes made to the bytecode at compile time will result in the loss of portability. If the augmenting of the bytecode were to take place at compile time, a more in-depth analysis could take place and a resulting better algorithm could be used. This would completely eliminate the performance hit at run-time.

Several measurements were taken to determine the actual cost of communication between the host and the co-processor connected through the PCI bus. Context switches will vary in cost depending on the amount of data that must be transferred which is dependent upon the current execution state. Our development environment, the HOT-IIXL board, contains 4Mb of user memory, so we tested both extremes of data transfer [5]. For 100 transfers of zero data, i.e. a simple handshake, 4022 cycles were required. For 100 context switches with transfers of 4Mb data in each direction, 71926957 cycles were used. Tests were performed on a 750Mhz Pentium III host, which provides 1193180 cycles/second of computation. This clearly shows the high cost in performing a context switch, especially when a high data transfer is required.

5. Context Switching Algorithms

There are three basic algorithms that were investigated: pessimistic, optimistic, and pushy. Each of these algorithms analyzes the methods found within each of the classes that are requested for loading during the execution of a given Java program. The algorithms insert new opcodes into the methods that result in a context switch from one processing unit to another. With the addition of bytecodes into the methods, the class structure itself is changed to reflect this and make the class legal for classloading. Each of the algorithms work on the basic idea of creating blocks of bytecodes that can be executed within the hardware accelerator. The analysis to create the blocks is done on the bytecodes being executed sequentially. A better analysis is more than likely possible by investigating the branching structure of the bytecode, however such an analysis is too costly to perform at run-time, especially with no predictive branching model for the application [3].

In analyzing the algorithms, not only do the algorithms need to be compared, but also the optimal block size must be considered. If the minimum block size is chosen (size=1) then a context switch to hardware will occur for every instruction that can be executed in the co-processor. This will result in many instances of context switching to execute one instruction. Clearly, this will result in slower performance. If a ridiculously high block size is chosen, then very few, if any, hardware blocks will be found and all execution will take place in software. This is complicated by the fact that branching instructions within a block can result in effectively shortening the block. Thus finding a block size, in addition to an algorithm, to minimize context switching but maximize hardware execution is critical.

The following subsections discuss the various algorithms that were investigated. For simplicity in the discussion, the portion of the virtual machine implemented on the FPGA will be referred to as the hardware side/unit.

5.1 Pessimistic Approach

An approach taken to blocking code for execution in the hardware unit is to assume the worst case scenario. This approach only inserts instructions to context switch to hardware in the event that the next predefined number of sequential instructions it sees are to be executed in hardware. Context switching back to the software partition occurs when an instruction is encountered in hardware that is not supported. Any instructions that are initially software instructions, before being changed into the hardware quick versions, are considered to be software only instructions. This ensures that if no branching takes place in the block of instructions, then the minimum number of instructions will be executed to offset the cost of performing the context switch.

The resulting drawback of this approach is that the execution becomes more software bound than hardware bound. This is due to two different characteristics of the bytecode. First, that there are a minimal number of blocks of sequential instructions made up of these types of instructions. As a result there are few context switch instructions added into the methods and execution tends to stay within the software partition. The second characteristic is that blocks of bytecode that contain instructions that will later be transformed into hardware instructions will never be tagged to be executed in hardware. Once transformed, if the instruction is encountered while executing in hardware, it will be executed there, but the algorithm fails to push the execution to hardware if the instruction is encountered in software.

5.2 Optimistic Approach

The optimistic approach attempts to capture the instances of sequential bytecodes where some of the instructions are initially software instructions, but will later be transformed into hardware instructions. This is accomplished by assuming that this class of instructions is executable in hardware during the augmenting process. This is done with the desire of creating more blocks of instructions, which can be executed in the hardware partition, thus resulting in more context switch instructions. As with the pessimistic approach, execution stays in the hardware partition until an instruction is encountered that requires execution in the software partition. To eliminate useless context switching where a context switch to hardware instruction is immediately followed by a software instruction, a check is made before every context switch to ensure that the next instruction is truly a hardware instruction.

The resulting drawback of this approach is that in some cases this results in fewer context switches to hardware. This is due to instances where previously two blocks were delimited for execution separated by a transforming instruction. Consider figure 2 where previously two hardware blocks may have been created, instead 1 larger block is used. Upon first execution of the block, execution will switch back to the software side on line 3, to change the instruction to its quick form. Thus, first execution of the block will be performed in software, subsequent executions will occur in hardware. In the event that the block is executed only once, then no execution will occur in hardware. More importantly, it is possible that the loop in the block may be computationally intensive. The previous algorithm may have inserted context switch instructions inside the loop and triggered execution in hardware, but the optimistic algorithm wants to create the one large block. The optimistic approach fairs no better in forcing execution into the hardware partition when possible due to a lack of context switch instruction(s) in the appropriate place(s)

```
1:          sw
2:          conshw
3:          sw/hw instruction
4:    label:      hw
5:          sw/hw instruction
6:          …
7:          hw
8:          goto label
```

Figure 2: Bytecode Example

5.3 Pushy Approach

The pushy algorithm attempts to "force" execution back into the hardware partition whenever possible. This is accomplished by modifying the optimistic approach such that whenever an instruction is encountered in the hardware partition that forces execution back into software, the instruction is executed in software as required, but the virtual machine attempts to force the execution back to the hardware partition as soon as possible. A context switch to software instruction signifies any instance where execution is desired to be in the software partition.

This has a positive effect on blocks that are executed multiple times. After the initial execution, the Java instructions that invoke the transition from hardware to software change to become hardware instructions themselves. Thus, if only one explicit context to hardware instruction is seen, the block of byte code will still execute in hardware. This has a negative effect on blocks that are executed only once. In these cases, the execution flow jumps back and forth between partitions as it attempts to force execution in hardware.

Additional improvements were tried with the pushy algorithm to perform a further look ahead when determining to push the execution back to hardware. This was accomplished by looking ahead to verify that the next two instructions were executable in hardware. This used the assumption that the execution flow would follow sequentially and not branch to a different location. The results of comparing the two look ahead techniques showed that the gain was very little. This is due to the infrequent number of instances where execution is pushed back into hardware. The additional penalty for looking ahead further does not outweigh the number of saved context switches.

6. Results

Any numerical results obtained are highly dependent on the benchmarks chosen. To determine the performance and characteristics of the various algorithms, benchmarks from the standard specJVM98 test suite were used [4]. This does not guarantee the results are not skewed based on the benchmarks, but the tests are intended to be a representative set of Java programs. For this work, all benchmarks, which provided source code, were used. Thus, the tests used were *jess*, *raytrace*, *mtrt* (multi-threaded raytrace), *db*, and *compress*. Two other in-house tests, namely *mandel* (calculate the mandelbrot set) and *queens* (calculate 8-queens problem), were also used.

Several interesting characteristics showed with the algorithms and the benchmarks. One interesting characteristic is that in some cases, the optimistic approach performed very poorly in comparison to the pessimistic and pushy approaches. This can be seen in the Mandelbrot test shown in figure 3 where the optimistic had nearly 0% hardware instructions for all block sizes, while pessimistic and pushy reached almost 100%. This is due to the instance where a block of bytecode that is wrapped by context switch instructions, contains a loop that dominates the execution time, but consists of an initial software instruction, as previously described in section 5.2. This effect does not occur in the other two algorithms.

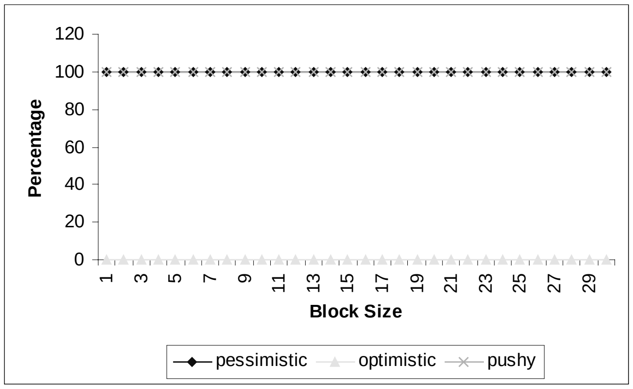

Figure 3: Mandelbrot Percentage Hardware Instructions

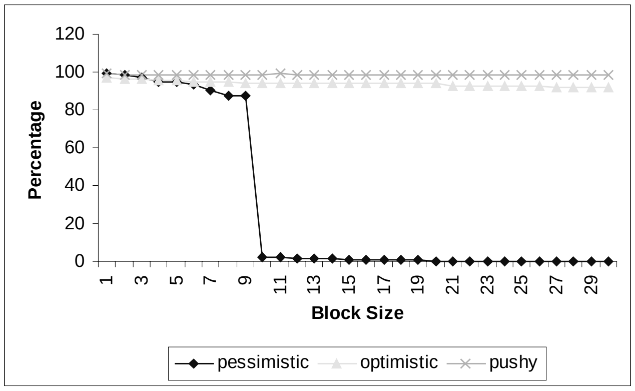

Figure 4: Jess Percentage Hardware Instructions

However, in other cases, the pessimistic approach faired poorly in executing instructions in hardware. This is a result of the dominating execution points having a high concentration of first time software instructions. This high concentration results in very few or no context switch instructions to be added. Thus all of the execution takes place in software. This does not happen with the other algorithms as they desire to push the instructions to hardware on subsequent executions of the bytecode. Figure 4 shows the percentage of instructions executed in hardware for the *Jess* benchmark. As the block size gets bigger, the amount of execution in hardware drops to almost a negligible amount very quickly. This effect happens for all benchmarks with the pessimistic approach, but at different block sizes.

If one examines, the algorithms together over all of the benchmarks, figure 5, it is seen that the pushy algorithm performs best for providing a high amount of execution on the hardware partition. This is very important since the higher volume of execution in the hardware partition will increase performance both by executing in the faster hardware partition, but as well by providing a large window where parallelism can be used. It must also be considered that the pushy algorithm contains a greater level of run-time logic required to determine if execution should return to hardware after a context switch to software.

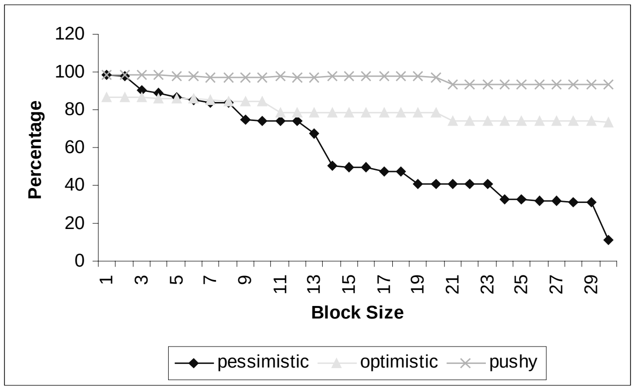

Figure 5: Average Percentage Instructions/Context Switch

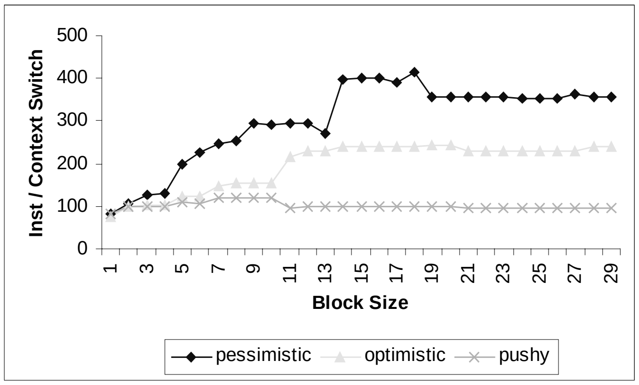

Figure 6: Average Number Instructions/Context Switch

Another concern is: what is the optimal block size? Figure 5 shows that as like other algorithms the percentage of execution in hardware slowly decreases as the block size gets larger. However, for the pushy algorithm the decline is much less dramatic. Figure 6 depicting the average number of instructions per context switch, shows that the pushy algorithm performs best for blocks of size 7-10, with local maximum of 8. Further sampling of the various Java programs may show a better block size.

7. Conclusions and Summary

From the results presented above it can be clearly seen that all of the algorithms are very susceptible to characteristics that vary between applications. As such, any given algorithm could perform best depending on the Java application. If the augmenting of bytecode were to take place at compile time, a better algorithm could be used that could be more adaptive to the characteristics prevalent in the bytecode. This would also eliminate the performance hit at run-time, resulting in even higher performance gains.

Parallelism will be a dominant factor in how much performance increase is attained. The more execution that takes place in hardware will allow for more parallelism! Forcing the execution of bytecode in hardware may not itself contribute to a performance increase once the context switch cost is factored in, but it will provide an opportunity for parallelism that will provide an increase.

The results attained look promising for the percentage of execution that can take place in hardware and the relatively low number of context switches that are necessary to attain it.

12. References

[1] Aoki, Takashi, and Eto, Takeshi. *On the Software Virtual Machine for the Real Hardware Stack Machine*. USENIX Java Virtual Machine Research and Technology Symposium, April, 2001.
[2] El-Kharashi, M. W., ElGuibaly, F., and Li K. F. *An Operand Extraction-Based Stack Folding Algorithm for Java Processors*. International Conference on Computer Design, 2000.
[3] Hecht, Matthew S. *Flow Analysis of Computer Programs*. The Computer Science Library Programming Language Series. Elsevier North-Holland, 1977.
[4] *http://www.spec.org/osg/jvm98*. Standard Performance Evaluation Corporation, November 1997.
[5] *http://www.vcc.com/Hotii.html*. Virtual Computer Corporation, July 2001.
[6] Kent, Kenneth B. and Serra, Micaela, *Hardware/Software Co-Design of a Java Virtual Machine*, 11th IEEE International Workshop on Rapid Systems Prototyping, June 21-23, 2000.
[7] Lindholm, Tim and Yellin, Frank, *The Java Virtual Machine Specification*, Addison Wesley, September 1996.
[8] Sun Microsystems, *The Java Chip Processor: Redefining the Processor Market*, Sun Microsystems, November 1997.
[9] Sun Microsystems. *picoJava-I: picoJava-I Core Microprocessor Architecture*. Sun Microsystems white paper, October 1996.
[10] Sun Microsystems. *picoJava-II: Java Processor Core*. Sun Microsystems data sheet, April 1998.
[11] Wayner, P. *Sun Gambles on Java Chips*. BYTE. November 1996.