Hardware Architecture for Java in a Hardware/Software Co-Design of the Virtual Machine

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

Abstract

This paper discusses the hardware architecture used in the hw/sw co-design of a Java virtual machine. The paper briefly outlines the partitioning of instructions and support for the virtual machine. Discussion concerning the hardware architecture follows focusing on the special requirements that must be considered for the target environment. A comparison is performed between this design and that of picoJava, a stand-alone processor for Java. The paper concludes with benchmark results for this architecture compared with software execution.

1. Introduction

This paper discusses the idea of improving Java performance by creating a hardware co-design that works in unison with the general microprocessor to form the virtual machine. Specifically, this paper briefly describes the concept of the co-designed Java virtual machine, outlines the partitioning of the virtual machine between software and hardware, and describes the hardware architecture and its components. One of the innovative features of this scheme is that a field programmable gate array (FPGA) is used to realize the hardware partition, thus providing scope for extended reconfigurability. Subsequent sections of the paper compare the proposed architecture with that of picoJava, and give conclusions and a summary.

2. The Co-Design Idea

Our goal is to create a hardware co-design that works in unison with the general CPU of a desktop workstation to provide a full Java virtual machine [5]. This co-design strategy is undertaken with the goal of providing a performance improvement over a software only designed solution. For ease of development and flexibility in partitioning and design we are using an FPGA to implement the hardware design. Initially, this FPGA is located on a board connected to the host computer via the PCI bus. Our research maintains a clear vision of potentially providing the hardware support directly on the mainboard. Higher communication costs and two distinct regions of memory are the trade-offs we are paying for the simpler development and added flexibility of the FPGA at this time. These result in slower performance scores for the empirical results, but do not affect the overall design process. The choice of an FPGA also allows future reconfigurability for many other applications.

3. Partitioning

As with all co-designed systems, the partitioning of the design between hardware and software is a vital step in the process.

3.1 Software Partition

The software partition is intended to provide the support required by the hardware partition and is capable of executing all the virtual machine instructions not implemented in hardware.

The extra support that the software partition must provide includes transferring data during context switches between the hardware and software partitions, performing class loading and verification, garbage collection, type checking, exceptions, as

well as thread and memory management. These operations are all needed by the hardware partition, but cannot be performed in hardware due to their complexity and the limited design space.

The actual virtual machine instructions that are provided in software rather than hardware are instructions that involve the above software supported features. These include instructions such as *new*, *anewarray*, *checkcast*, and *instanceof*. The software partition is also capable of executing *all* virtual machine instructions to allow for smart context switching between the hardware and software partitions [4].

3.2 Hardware Partition

For the hardware partition it is desirable to include any virtual machine instructions that can be implemented in the given design space that is available. Many of the instructions that can be implemented in the hardware partition are those that can be found in traditional processors such as: constant operations, stack manipulation, arithmetic, shift and logic operations, type casting, comparison and branching, jump and return, as well as data loading and storing.

In addition there are other Java specific instructions that can be implemented in hardware. Some of these instructions are the *quick* instructions that perform a given operation knowing that the object or class used is already resolved and available for use. It is these instructions and the stack architecture that gives the hardware design a different look from traditional co-processors.

4. Development Environment

As with all systems, the design is constrained by the target environment. In this case, we are targeting a desktop workstation that has an FPGA available through the PCI bus. More specifically we are using a Hot-II card available from VCC, that has the following configuration [3]:

- XC4062XL Xilinx FPGA with 2304 CLBs.
- 4Mb of user memory.
- 32-bit PCI interface.

The development environment choice is particularly important since it affects the hardware design due to factors such as available number of combinational logic blocks (CLBs), memory, and communication bandwidth.

5. Hardware/Software Interface

The interface between the hardware and software has a direct effect on the design of both. In this case, it was decided to maintain a minimal amount of communication between the two partitions when either of them are computing. Thus, the design is handed addresses to locations where the necessary data exists within memory on the FPGA card. Once the hardware partition is finished execution, it signals the software using an interrupt. The use of an interrupt allows the software partition to perform some other computation in parallel with the hardware. At that point, the software partition retrieves the current state of the virtual machine from hardware and continues execution. This results in a very simple communication protocol between the hardware and software that can be implemented using a single data bus and a control line.

The data required by the hardware design consists of the method's bytecode, local variables, constant pool, object store, and execution stack. Most data structures, with the exception of the object store and constant pool, are accessed frequently and require transfer to the FPGA's local memory. Of these, the hardware design is capable of changing only the local variables and the execution stack. This reduces the communication when returning data and execution back to software. The object store and constant pool are accessed less frequently, and are potentially much larger than the local memory available to the FPGA. This requires that the larger host memory be accessible to the hardware design. If this is not possible, then bytecode instructions, which utilize this data, can be omitted from the hardware design.

6. Design Implications

Due to the target environment, there are several implications to the requirements of the hardware design. One primary consideration is the availability of resources. With restricted design space, there is a

need to have a design that has minimal space and maximal support of bytecode instructions. The architecture must also be flexible to promote easy implementation on different hardware environments. Having a design that can be quickly modified to fit on a smaller FPGA is an important requirement. Additionally, the hardware co-design must be active to compensate for the slow communication speed between the software and hardware partitions. This active design will assist in the communication between the hardware and software instead of waiting for the software to push data onto the hardware partition.

7. Hardware Design

The hardware design is comprised of 4 main units: the host interface, instruction buffer, execution engine, and data cache controller. The architecture is based upon a 3-stage pipeline that

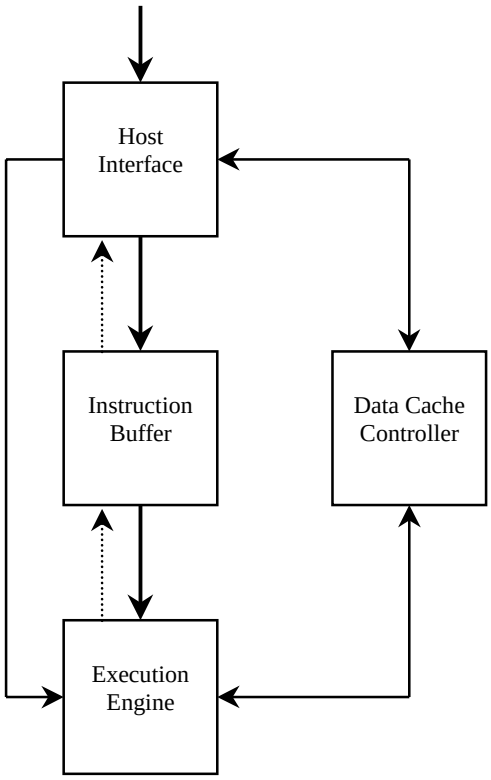

Figure 1: Overview of Hardware Architecture

funnels instructions through the first three units respectively. Figure 1 shows the interconnections between the components and the pipeline between them. The pipeline works in the traditional "fetch-decode-execute" method. The parallelism of this pipeline contributes to any performance gains obtained from the hardware design.

The dark connections show the direction in which the instructions travel through the pipeline, the dashed connections show control lines that are used to feed information back to its feeding unit. The feedback information is primarily used to indicate that the unit is ready for the next instruction. This is useful for instructions that take a variable amount of execution time. Thus, it is not easy to structure the pipeline to provide automatic forwarding of instructions.

The following subsections discuss each of the units in detail and their purpose in the architecture.

7.1 Host Interface

The host interface is the central point within the architecture for communication with both the on-board memory and the host computer. It is involved in retrieving instructions and data as well as handshaking with the software partition in performing context switches. Bytecodes are retrieved from memory and are pipelined to the Instruction Buffer. Any requests from the Instruction Buffer to change the address of instruction fetching results in a delay of execution.

The Data Cache Controller only requests data when the current instruction is waiting for the data, thus any requests from the Data Cache Controller for data take precedence over the fetching of instructions. As such, a variable delay in processing is possible due to the transaction requests of other components. This is necessary since execution cannot continue until the higher precedence requests, which include data accesses, and stack over/under flows are fulfilled.

7.2 Instruction Buffer

The Instruction Buffer acts as both a cache and a decoder for instructions. The cache can be a variable size, and is primarily determined by the amount of design space that is available. A larger cache is preferable as it provides a higher probability that upon executing a branch instruction, the next instruction will already be in the cache and not require a delay as the instruction is retrieved from

on-board memory. There is no real disadvantage to having a large cache, other than the area required to support it. Since the target environment is an FPGA, using the remaining area available is not costly. When the cache has room for more instructions, or the instruction required by the Execution Engine is not located in the cache, then the instruction buffer requests the next instruction from the host controller.

Instruction decoding is performed to align the instructions before passing to the Execution Engine. The Java virtual machine has the property of different sized instructions, and the instructions come from software packed together to reduce memory usage. In the current environment the memory available on-board is rather low (4Mb). Therefore, it is better that our design perform the decoding and padding in hardware rather than software. This allows for utilization of the local memory of the FPGA to the fullest and reduces communication between the hardware and software partitions.

The Instruction Buffer decodes the next instruction for the Execution Engine to execute and pipelines the instruction. The Instruction Buffer predicts no branching and feeds the next sequential instruction. In the event that a branch occurs, the Execution Engine ignores the incorrect pipelined instruction, and the Instruction Buffer pipelines the correct instruction. This may take a variable amount of time depending on cache hits and misses. In the event of a cache miss, the instruction cache is cleared and it starts re-filling at the new branch address.

7.3 Execution Engine

The Execution Engine receives instructions from the Instruction Buffer to execute. To assist in the execution, the Execution Engine has a hardware stack cache of 64-entries that contains the top elements on the stack. As the stack under/over flows, stack elements are communicated with the Host Interface that manages the complete stack in RAM on the FPGA card. This on-demand loading and storing of the stack prevents against unnecessary loading of the stack when context switching to hardware and execution may return back to software quickly. It is not seen as a performance penalty when the stack elements are required and execution is stalled, since execution will have to be delayed in either situation. This approach protects against situations where execution may return to software before the stack elements are required.

The data cache controller is used to fetch/store data from/to the execution frames local variables. Data that is required in the execution controller from the constant pool are received directly through the Host Interface. This is done since the constant pool will never be updated in the hardware design. Thus, instructions that require accessing the constant pool or local variables take a performance hit. This is unavoidable due to the potential size of both the constant pool and local variables. The Java virtual machine avoids taking this hit, by loading data onto the stack, performing its manipulations there, and only pushing the data back out to memory when completed.

7.4 Data Cache Controller

The Data Cache Controller is responsible for interacting with the memory both for loading and storing data from the local variables when required. Ideally it contains a cache that buffers data to reduce the number of transactions with the on-board memory. This cache is a write-on-demand architecture that writes cache data to the RAM immediately upon writing to the cache. This prevents against the cache having to be flushed when execution returns to software. In the event that no design space is available to house the cache, the size of the cache can be zero, which results in a memory transaction every time the Execution Engine makes a request. The effects of this on performance are dependent upon the difference in penalties for accessing RAM instead of cache.

8. Architectural Features

The architecture possesses various features that allow it to be a successful design in the resource restricted target environment. The design is based on a native stack and a pipelined architecture. It also attempts to be active and assist the software in the transfer of data between software and hardware. This is achieved by the loading of the data stack in hardware on demand. This avoids the situations

where the data stack is loaded, only to have execution switch back to software. This is an expensive penalty since the stack needs to be stored back to software. The Data Cache Controller is designed to push all of the cached data back to software on writing to the data cache to avoid dumping the cache to RAM when context switching occurs.

The design is also compact and flexible as required. The Instruction Cache and Data Cache are both configurable and can be resized to accommodate a larger or smaller memory cache. As expected, the larger the cache, the better, but this is a trade-off against area resources. The core pipeline can be altered to remove the data cache from the design completely. This is possible; however removing the data cache forces all local variable accesses to communicate with the external RAM memory. This drastically affects performance as expected and should only be performed when the FPGA resources are at a minimum.

9. Comparison to picoJava

The picoJava core is the original design of a Java processor based on the virtual machine for the goal of creating a native Java computer [7-11]. Thus, a comparison between the design of picoJava and our design is appropriate. Through the comparison it is clear that both designs share various basic characteristics, however they also differ significantly.

A major difference between the picoJava core and this design is the environment for which they are targeted. The picoJava core is designed for the purpose of being the sole processing unit, while our design is intended to complement an already existing general microprocessor. Thus, the proposed architecture does not require operating system instructions for support in accessing hardware components such as RAM and various busses, as well as additional instructions for supporting different languages and paradigms.

With a greater restriction in the design space available, it was deemed suitable to remove the process of folding instructions from hardware. This process can be relocated to software. The restriction in design space results in less area and less space to implement special instructions that can perform multiple Java virtual machine instructions as a single instruction. Any special techniques that can be utilized to increase performance through folding, re-ordering, or re-structuring of instructions are left to be implemented in software. This is beneficial not only for saving precious design space, but also for performing the operation during class loading where it need only take place once. This is as opposed to in hardware where it is done at every encounter of the instructions.

The picoJava core provides flexibility in its design with the caches it uses and allows for various cache sizes. Our design also contains caches for the same purposes, but it emphasizes flexibility with much smaller sizes. The picoJava specification outlines the caches as ranging from 16Kb to 0Kb in size. Our design pushes the smaller caches further, by using caches that are 1Kb to no cache.

Overall, the emphasis on the differences between the picoJava core and this design is in the simplicity and reduction of the design. This is a necessary step for the targeted environment. This can be clearly seen in the simplified pipeline architecture in comparison with the complex parallel architecture of picoJava with instruction folding.

The differences between our design and picoJava can also be seen in the users and applications targeted by each. Both designs can be used for embedded systems, but our design is better for this application due to its smaller size and tight coupling with a microprocessor and other factors. The picoJava design has extra functionality that is undesirable in a co-designed system where hardware resources are more limited.

10. Initial Evaluation

Before integration of the hardware architecture with the software partition, the design is tested for both correctness and performance against the software virtual machine. This fine-grain testing allows for insight into the potential gains a co-designed virtual machine can provide. For this, the tests may only use the subset of Java bytecode that is available currently in the hardware partition as the full co-designed machine is under full development. The following tests were implemented and executed:

- loop counter

- Fibonacci finder
- Ackerman function
- bubble sort
- insertion sort

Each of these tests were designed to evaluate various design features of the hardware architecture. The first two tests allow for a long duration and constant test with no effects of stack cache, data cache or instruction buffer interference. The Ackerman function tests the architecture for handling of overflow/underflow of the stack cache. The bubble sort test sorts local variables and rigorously tests both the instruction cache and the data cache. The insertion sort test utilizes the ability of the hardware architecture to access the host systems memory.

For gathering test results, the software execution was performed on a Windows 2000 workstation, with no other user applications executing. Thus, it is not guaranteed that the software timings are not affected by contention with other system processes over the processor. However, this contention is at a minimum for a typical workstation environment. Results of execution timings are in clock cycles, thus the processor speed is not a factor in the comparison.

10.1 Linear Execution Tests

These tests are designed to show the potential improvement of the hardware execution over software interpretation. Figure 2 shows the ratio of increase of the simulated hardware over the timed software execution. It can be seen that the software performance improves as the problem becomes larger. This is attributed to the effects of software executing in a multi-tasking environment. As the performance is extended over a larger period, these effects are minimized to give clearer results. At the largest problem size computed, the hardware provides performance gains at a factor of 11.7 for the simple loop counter and 8.7 for the Fibonacci problem.

10.2 Stack Testing

To test the performance of the hardware architecture with over/under flow of the stack cache, a Java method to compute the Ackerman function was used. This function is recursive and is ideal for stack use. The largest Ackerman function calculated was for Ackerman (3,5). This function provided an 11.5 factor of improvement over the software only interpretation, while over/underflowing a total of 12787 times during execution. Thus, the hardware architecture maintains better performance than software despite the stack maintenance.

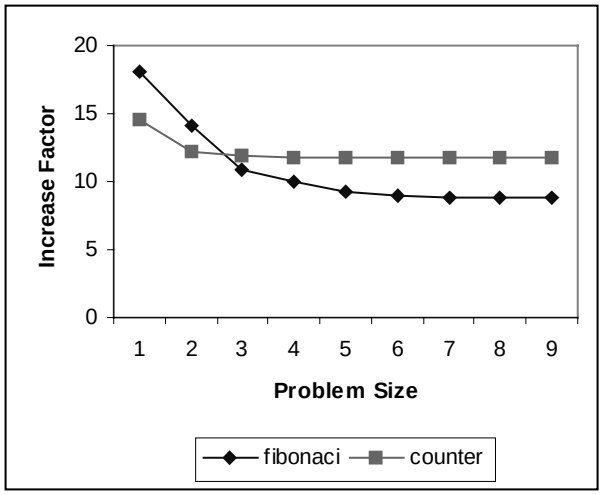

Figure 2: Improvement Factor for linear tests.

10.3 Instruction Buffer Testing

The bubble sort test sorts the local variables of the Java method into ascending order. Using local variables, it is not possible to index all of the variables at run-time. Instead each variable is resolved at compile-time. As a result, there is code replication of the bubble sorting algorithm for each pair of local variables compared. For this test, 64 local variables of descending order were sorted into ascending order, thus providing a worst case scenario. The data cache was set to a fixed size of 64, equal to the number of local variables used. The size of the instruction cache was manipulated to view the effects on performance. Figure 3 shows the fluctuation of performance in hardware. The instruction cache ranges from 64 bytes to 1088, just large enough to hold all the method's bytecode.

From the graph it can be seen that the hardware execution time decreases by roughly 1600 clock cycles, or 1%. This shows that the instruction caching mechanism is sufficient in providing

enough instruction throughput for the Execution Engine to compute.

10.4 Data Cache Testing

The bubble sort test is also used to test the effects of resizing the data cache. This may be necessary in order to take advantage of the trade-off between speed and space. The bigger the data cache, the bigger the FPGA to hold the design is required. It may become necessary to reduce the size of the data cache to fit the architecture within the available resources. Figure 4 shows the change in execution performance versus the increase over executing the same bytecode in software. As the data cache is reduced, from 64 entries down to 0, 2 entries at a time, the performance consistently drops. The initial performance increase factor of 7.7, drops to 6.2. While still providing a considerable performance improvement, the effects of reducing the data cache are significant. Clearly the effect of reducing the data cache is more severe than reducing the instruction cache. This indicates that if area is at a premium, then the instruction cache should be reduced prior to reducing the data cache in an attempt to use less area.

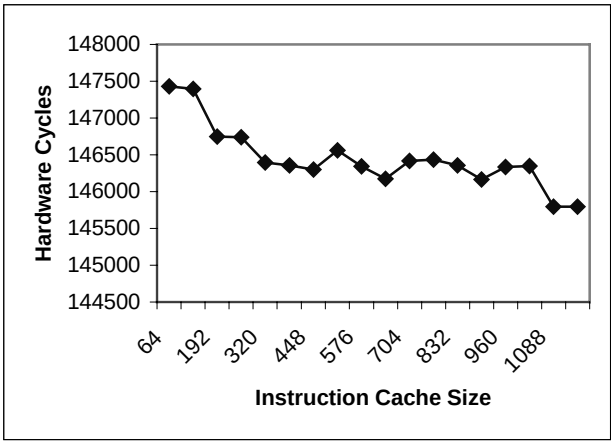

Figure 3: Execution times for varying size of Instruction Cache.

10.5 Remote Memory Testing

The insertion sort test provides interesting information on the effects of executing in the hardware architecture when it requires communication with the host system to access its object store. The insertion sort execution provides a constant performance increase of a factor of 6.6. This constant performance gain is achieved by the constant ratio of host system memory transaction to instructions executed. What is interesting about this ratio is that the threshold latency of host memory transactions can be calculated to determine when software performance will be better than hardware.

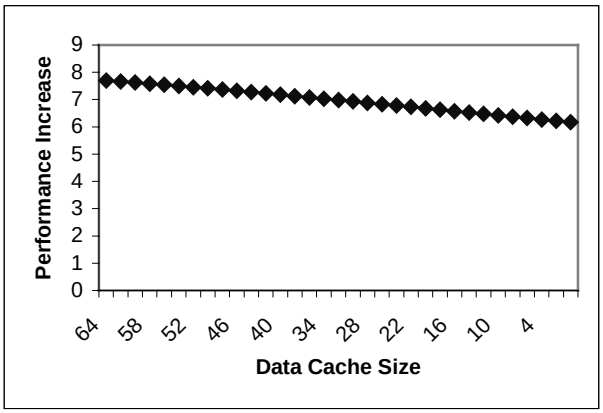

Figure 4: Performance degradation for reduced data cache sizes.

Calculations show that the threshold latency for the insertion sort problem is in the range of 238 – 243 cycles. This is dependent on the 20% ratio, 1 memory transaction instruction for every 5 hardware instructions. Research has shown that the frequency of instructions that require data from the object store is only 17.61% [1,2]. This shows that for the typical application, hardware support for these instructions can provide a performance increase. These results can also be used to gain an insight into the communication requirements between the hardware and software components.

11. Conclusions and Summary

The picoJava core is not suitable for all hardware implementations of the Java virtual machine. For example, it is too large and complex when providing hardware support for the virtual machine on a desktop workstation. This is emphasized by the lack of resources available in a typical reconfigurable environment. Our design takes into consideration the target environment and is flexible enough to be modified as needed, and so is a more suitable solution. These considerations include reduced area

for design of the processor and memory areas, and the fact that our design is dedicated solely to the purpose of increasing Java performance. The results thus far show performance gains of up to a factor of 11.7, and we have demonstrated that if area for the design is at a premium, reductions to the instruction cache size do not severely affect the performance. We also provide some indication of the required communication rate between our design and the host system.

12. References

[1] El-Kharashi, M. W., Elguibaly, F., and Li, K. F., *A Quantitative Study for Java Microprocessor Architectural Requirements. Part I: Instruction Set Design*, Microprocessors and Microsystems, vol. 24, no. 5, pp. 225-236, Sept. 1, 2000.

[2] El-Kharashi, M. W., Elguibaly, F., and Li, K. F., *A Quantitative Study for Java Microprocessor Architectural Requirements. Part II: High-level Language Support*, Microprocessors and Microsystems, vol. 24, no. 5, pp. 237-250, Sept. 1, 2000.

[3] http://www.vcc.com/Hotii.html. July 2001.

[4] Kent, Kenneth B. and Serra, Micaela, *Context Switching in a Hardware/Software Co-Design of the Java Virtual Machine*, 2002 Design Automation and Test Europe conference Designer's Forum proceedings, March 4-8, 2002.

[5] 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.

[6] Lindholm, Tim and Yellin, Frank, "The Java Virtual Machine Specification", Addison Wesley, September 1996.

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

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

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

[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.