

An Implementation of the Hardware Partition in A Software/Hardware Co-Designed Java Virtual Machine

Hejun Ma, Ken Kent*, David Luke

*Department of Electrical and Computer Engineering, Faculty of Computer Science**

University of New Brunswick

Fredericton, New Brunswick, Canada

z9xad@unb.ca, ken@unb.ca, luke@unb.ca

Abstract

This paper presents an implementation of the hardware partition of a co-designed Java Virtual Machine. More specifically, it briefly introduces the concept of the co-designed Java Virtual Machine, outlines the partitioning scheme between the software and hardware, illustrates the architecture of the hardware partition, and explains the implementation of the hardware partition itself. Using an FPGA as the hardware target provides great flexibility for different configurations. Several tests of Java bytecodes are applied to evaluate the functionality and performance of various features of the hardware. The experimental results are presented and conclusions are given.

Keywords: *Java Virtual Machine; Co-Design; Hardware Implementation; Field Programmable Gate Array.*

1. INTRODUCTION

The Java language was initially designed to address the problems of building software for heterogeneous network devices [6]. Therefore, system independence is an essential feature for Java to be delivered across the network and run on various platforms [8]. The Java Virtual Machine (*JVM*) provides an intermediate platform and acts as an interpreter between the Java program and the physical computer. Java programs are designed to run on the virtual platform instead of being directly executed on the host processor. This achieves the goal of “write once, execute everywhere”.

There are several approaches to implementing a JVM. Software implementations, such as an interpreter or just-in-time (*JIT*) compiler, are the most common way to implement a JVM. They are relatively easy to implement and cost effective, but offers low performance [7]. Possible hardware implementations include techniques such as a native processor or a hybrid processor [2]. They offer better performance but the design complexity is

increased and they lack flexibility for supporting different platforms. The co-designed JVM takes advantage of the implementation techniques from both software and hardware approaches to achieve a better compromise between performance and cost. The next section discusses the principle of co-design as applied the JVM in detail.

2. CO-DESIGN JVM

A goal of the co-designed JVM is to achieve better performance over a pure software solution. The hardware partition of the co-designed JVM is proposed to work in unison with the host processor of a desktop workstation [4]. The choice of a Field Programmable Gate Array (*FPGA*) provides a development environment for easily shifting the partitioning between hardware and software to arrive at an optimized solution, and the flexibility to fit the design into targets of different sizes.

A co-design system is comprised of two parts, namely hardware and software. Generally it implements the low level support with hardware, and the high level system in software. Yet in this co-design, an overlapping partitioning strategy is applied, i.e. to have the subset of instructions implemented in hardware also supported in software [5]. Such a scheme reduces the data transactions between hardware and software and therefore increases the execution speed. It also provides great potential for parallel processing between the two computing units.

2.1 Software Partition

The software partition includes the high level components of the JVM. It is responsible for providing system support required by the hardware partition. It also implements all the Java bytecodes in this partition.

Due to the complexity and restricted design space, there are some operations that are needed by the hardware partition that cannot be implemented in hardware. Therefore the software partition provides this function and others such as transferring data during

context switches between the hardware and software partitions, performing class loading, and verification [3].

The software partition also includes all the Java instructions that are implemented in the hardware partition [4]. Since this co-design JVM is targeting a desktop workstation environment, the additional space utilized to provide the support is not critical.

2.2 Hardware Partition

It is desirable to have the hardware partition include any instructions that can be implemented in hardware, provided there is enough space in the FPGA. It includes the conventional instructions such as constant operations, stack manipulations, and arithmetic operations. Other Java specific instructions, such as the *quick* instructions, are also supported in this partition. The *quick* instructions perform some operation knowing that the object or class being used is already verified and loaded.

The instruction set for the hardware partition is divided into 3 levels to provide various configurations to fit FPGAs of different sizes. The *Compact* partition contains only the fundamental instructions for execution and requires minimal system knowledge for execution. The *Host* partition extends the *Compact* with support for accessing the host memory system. The *Full* partition extends the *Host* with support for *quick* instructions [5].

3. DEVELOPMENT ENVIRONMENT

A desktop workstation with an FPGA connected through the PCI bus is chosen as the development environment. More specifically, we are using the APEX PCI-Board/A10C from Altera Corporation, which has the following configurations [1]:

- Apex EP20K FPGA with 38,400 logic elements.
- 32MB on-board SDRAM module.
- Supports 64-bit PCI bus @ 66MHz.
- PCI MegaCore function pci_mt64.

The software development platform is Quartus II 3.0. It is used for HDL design entry, synthesis, and place and route. ModelSim 5.7 is used for simulation.

4. HARDWARE IMPLEMENTATION

The hardware partition can be divided into three parts: the Java Engine, the local memory, and the PCI interface. The Java Engine executes the designated Java bytecode instructions. The local memory hosts the Java bytecodes. The PCI interface communicates with the PCI bus to exchange data and control signals. Figure 1 is the abstract architecture of the overall design.

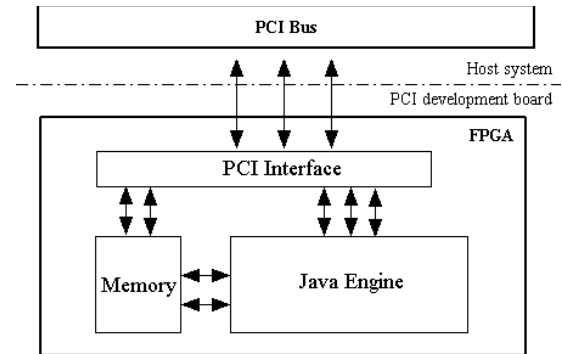


Figure 1 The Hardware Partition

4.1 Java Engine

The Java Engine consists of 4 main units: Host Interface, Instruction Buffer, Execution Engine, and Data Cache Controller. The execution procedure is based upon a 3-stage pipeline architecture, i.e. fetch, decode, and execute. Figure 2 shows the interconnections among these units and the direction of the data flow. The instructions are pipelined through the Host Interface, the Instruction Buffer, and the Execution Engine sequentially, as shown by the thick gray lines. The single lines are the control signals for the handshake between the modules such as requesting an instruction or data. The handshake is necessary in several instances due to the variable amount of execution time for different instructions.

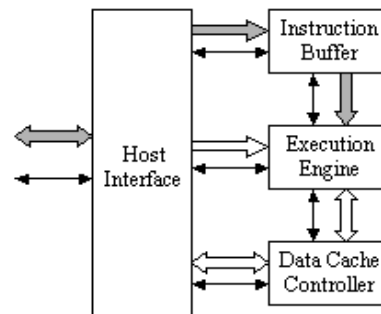


Figure 2 Overview of Java Engine Architecture

4.1.1 Host Interface. The Host Interface is the communication center within the Java Engine. It interacts with both the PCI bus and the on-chip memory to retrieve instructions and data, as well as perform the context switch with the software partition. The other 3 modules have different priorities in requesting the Host Interface to fetch instructions or data for them.

The Execution Engine has several registers that hold the values of program reference pointers, such as the Program Counter (PC). Before any instructions are retrieved, the Host Interface retrieves these parameters

for the Execution Engine. These values are updated during execution and sent back to the software partition when execution is finished.

The Host Interface retrieves the instructions from the memory and pipelines them to the Instruction Buffer. The Data Cache Controller requests data only when the current instruction needs them. In this case, the Host Interface will suspend the instruction fetching for the Instruction Buffer, and retrieve the requested data.

4.1.2 Instruction Buffer. The Instruction Buffer functions as both a cache and a decoder. The cache stores the instructions that are pre-retrieved from the local memory. To provide the flexibility of fitting the design into FPGA devices with different sizes, the size of the cache is variable. Yet it is desirable to have the cache size as large as possible, thus providing a higher probability that upon executing a branch instruction, the next instruction is already in the cache and therefore eliminating the delay of retrieving the instructions from the slower memory.

Instruction decoding is necessary because of the unaligned property of Java instructions. The Java instructions have different lengths, and they are packed together in software to reduce the data block size. The Instruction Buffer performs the decoding on the instructions before sending them to the Execution Engine.

4.1.3 Execution Engine. The Execution Engine executes the instructions that are pipelined from the Instruction Buffer. The JVM is a stack based computing model, therefore as the Java program is executed, it uses a stack to retain the data being manipulated. During the stack overflow and underflow, the stack cache communicates with the local memory that contains the complete stack data via the Host Interface. Such an on-demand loading and storing of the stack protects against data transfers due to context switching between partitions [3]. When the Java Engine completes the execution, it signals the software partition using an interrupt for transferring execution back to the software partition.

The execution time varies for different instructions. The Execution Engine communicates with the Instruction Buffer to stall the feeding of instructions until the current execution process is completed. When a branch event occurs, a request as well as the new address for fetching instructions are sent to the Instruction Buffer.

The Execution Engine sends a request to the Data Cache Controller when the execution needs to write/read data to/from the local variables. In write transactions, the Execution Engine continues with the next instruction as the Data Cache Controller performs the write. The requests for data from the constant pool are sent directly to the Host Interface.

4.1.4 Data Cache Controller. The Data Cache Controller is responsible for loading and storing local variables from and to the local memory under requests. When the data is required for the first time, the Data Cache Controller interacts with the memory to retrieve the data and pass to the Execution Engine. At the same time the data is also stored in the data cache so that no more memory access is needed if the same data is requested again. The Data Cache Controller is also a write-on-demand architecture that writes data to the local memory immediately upon writing to the data cache. This prevents against the cache having to be flushed when the execution returns back to the software partition.

The size of the data cache can also vary depending on the available space of the FPGA. The larger the data cache is, the less access to the memory is required thus the faster execution speed achieves.

4.2 Local Memory

The local memory is responsible for hosting the Java bytecode to be executed in hardware. When the hardware partition is needed for execution, the data block is transferred to the memory via the PCI bus. The data block contains the packed instructions, stack data, and local variables.

In this design, the memory space within the FPGA device rather than the on-board SDRAM is exploited as the local memory. This is due to slow access to the on board SDRAM caused by the burst mode of the SDRAM controller core that Altera provides. Therefore using the on-chip memory can achieve a much faster accessing rate than the on-board memory. The FPGA device used in this research provides sufficient memory to contain the maximum Java program frame required by the test cases.

4.3 PCI Interface

The Altera pci_mt64 MegaCore function is used for the interface with the PCI bus. Therefore the design of the PCI interface focuses on interfacing with this core. This design is comprised of local master transaction and local target transaction control logic modules, as well as the interface with the memory on the FPGA.

The local target control interacts with the pci_mt64 to force it to act as a target. During a target write, the data flows from the PCI bus to the local target. When the context switches to the hardware partition, a target write is performed to load the bytecode from system memory into the memory on the FPGA.

The local master control initiates the pci_mt64 core to have the FPGA work under master mode. The data flows from the FPGA to the PCI bus during a master write. During the execution on the hardware, whenever an instruction requests data, such as the constant pool from

the host system, a master read transaction is triggered to read the data from the host memory via the PCI bus.

Since interacting with the PCI bus is time consuming, the communication between the hardware and software partitions are reduced to a minimum in order to increase the overall execution speed.

5. EVALUATION

The hardware design is tested with 5 different tests that implement different algorithms to evaluate the functionality and performance of various hardware features. These tests are: Loop counter, Fibonacci finder, Ackerman function, Bubble sort and Insertion sort.

With these Java bytecodes, the ability of handling the overflow/underflow of the stack cache, the load/store of data from/to the data cache, and the access to the host system memory are all tested and verified. The tests are performed with the simulation tool, ModelSim, by applying the above testbench. The results are compared to a software simulator that was previously designed [5] and proves that the hardware design achieves a correct result and is capable of handling various Java bytecodes.

5.1 Design Space and Speed

This design architecture provides a flexible solution for FPGA devices of different sizes. By utilizing either one of the partitions discussed in section 2.4, or changing the size of the Instruction Cache or the Data Cache.

For the purpose of balancing between the speed and the design space, the instruction cache is implemented with the logic elements within the FPGA to achieve faster access. The data cache is implemented with the memory bits on the FPGA device.

The design space for different configurations is listed in Table 1. The first three configurations implement the different partition schemes with the same size of cache: both caches of 64 entries. The last one implements the full partition, but with a smaller size of cache setting: both the instruction and data cache are 16 entries.

	# of Inst.	LEs	MHz
<i>Compact</i>	133	26,898	25.18
<i>Host</i>	149	29,595	25.06
<i>Full</i>	162	34,471	23.78
<i>Full (Smaller Cache)</i>	162	30,199	24.24

Table 1 Design Space for Different Configurations

Table 1 shows that design space (logic elements) is dependent on both the hardware support level and the cache size. The greater level of hardware support, the

higher performance the co-design gains when there is little data transfer between partitions [5]. When the requirement for communication increases, it is desirable to have a partition with fewer context switches.

Through the timing analysis, the maximum clock rate achieved is 23.78MHz. The speed is not competitive to the main processor due to some critical routes in the Execution Engine. Some of the instructions are time consuming therefore reducing the overall clock rate. Even though, from the investigation performed before, a better performance of the hardware over the software partition can still be achieved under certain circumstances [5]. Potentially the performance can be further increased with utilization of potential parallel processing of partitions.

6. CONCLUSIONS

In this paper, the hardware partition of a co-designed JVM is implemented. The design is stack based with a pipelined architecture. The functionality of the hardware is tested and validated with several test benches. This design provides a compact solution for a desktop workstation environment by targeting the design to a FPGA device. The flexibility of the Instruction and Data caches allows for fitting the design into FPGAs of various sizes. Future work of this research will focus on improving the performance of the hardware partition and integrating the partitions together for parallel processing.

References

- [1] Altera Corporation, http://www.altera.com/products/devkits/altera/kit-apex_dev_kit.html. Jan. 2004.
- [2] EL-Kharashi, M. W., *The JAFARDD Processor*. Ph.D. Dissertation, University of Victoria, 2002.
- [3] Kent, K. B. and Serra, M., "Hardware Architecture for Java in a Hardware/Software Co-Design of the Virtual Machine," *Euromicro Symposium on Digital System Design 2002*, pp. 20-27, Sept. 2002.
- [4] Kent, K. B. and Serra, M., "Context Switching in a Hardware/Software Co-Design of a Java Virtual Machine," *2002 Design Automation & Test Europe Conference Designer's Forum Proceedings*, pp. 81-86, Mar. 2002.
- [5] Kent, K. B., *The Co-Design of Virtual Machines Using Reconfigurable Hardware*. Ph.D. Dissertation. University of Victoria, 2003.
- [6] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*. Addison Wesley, Sept. 1996.
- [7] Suganuma, T., et als. "Overview of the IBM Java Just-in-Time Compiler," *IBM systems Journal*, vol 39, no 1, pp. 175 - 193, 2000.
- [8] Venners, B., "Java Beyond the Browser," *Java World*, Dec. 1996.