

Rapid Prototyping of a Co-Designed Java Virtual Machine

Kenneth B. Kent

Faculty of Computer Science

University of New Brunswick
Fredericton, NB, Canada

ken@unb.ca

Hejun Ma

Dept. of Electrical and Computer
Engineering

University of New Brunswick
Fredericton, NB, Canada

z9xad@unb.ca

Micaela Serra

Dept. of Computer Science

University of Victoria
Victoria, BC, Canada

mserra@cs.uvic.ca

Abstract

Virtual machines as an execution paradigm suffer in performance due to the extra layer of abstraction that is introduced. A possible new solution to relieve the performance penalty is to utilize a co-designed virtual machine that leverages the combined benefits of hardware and software. The feasibility of such an approach is an interesting challenge due to the complex nature of a virtual machine and its resource requirements to achieve performance gains. This paper discusses the process undertaken to prototype a co-designed virtual machine so as to rapidly determine its suitability. From this prototype a complete implementation of the hardware partition is developed. Several resource requirements determined from the process are highlighted in addition to an assessment of the process used.

1 Introduction

The Java virtual machine (JVM) provides a homogeneous platform for the execution of Java applications despite the underlying hardware architecture [14]. This layer of abstraction results in degraded performance in comparison to applications executed on the native hardware architecture. Many solutions have been proposed to address the issue of performance. Software solutions exist that perform Just-In-Time (JIT) compilation [7,17]. Several hardware solutions provide a dedicated Java based processor or a hybrid processor that directly supports Java [5,6,16].

An alternative solution to addressing the performance problem is a co-designed approach. This approach provides a fully functional JVM comprised of both hardware and software support in a desktop workstation environment¹ [9]. The dedicated hardware is realized through the use of a Field Programmable Gate Array (FPGA) tightly coupled

with the workstation's general purpose processor through a PCI bus. Our research maintains a clear vision of potentially providing the hardware support directly on the workstation mainboard. In this environment, the virtual machine can be implemented by both custom hardware and software components. The choice of where each specific functionality within the virtual machine exists must be determined during the co-design process.

The overall co-design process for a virtual machine is rigorous and contains many decisions which affect the resulting implementation. In this context the primary concern of the co-design is to obtain the best performance within the constraints of the environment. These constraints are:

- Size of the FPGA to hold the hardware implementation.
- Speed of the PCI bus connecting the hardware and software components.
- Two distinct memory regions; one for each partition within the virtual machine.
- Size and speed of the local memory necessary to support the hardware partition execution.

These constraints are in addition to keeping the overall design within reason in regards to cost and development time. To further complicate the process, it is unclear as to if this approach will result in a performance increase. It is questionable as to if the performance gains of executing in the faster hardware partition can outweigh the penalties incurred for switching execution between the partitions. For this reason, it is clearly desirable to rapidly prototype the co-designed virtual machine before investing the efforts necessary to fulfill an implementation. Quickly prototyping the co-designed virtual machine will also assist in choosing a suitable environment for the hardware partition.

¹ The co-design JVM is based on the JSDK version 1.3.0 JVM.

2 Co-Designed Java Virtual Machine

As with all co-designed systems, the partitioning of the design between hardware and software is a vital step in the process. For the context of this work, a general purpose workstation, there is no requirement to maintain a small software partition. Due to this missing constraint, it is possible to overlap partitions between hardware and software instead of maintaining disjoint partitions as is normal in co-designed systems. This design decision relaxes the conditions to switch execution between partitions, instead execution in the hardware partition is by choice rather than necessity.

The selection process of bytecodes to support in the hardware partition must consider the environment in which execution will take place. Due to the disjoint memory regions, it is necessary to maintain support for instructions that change the execution context (execution frame, runtime stack, ...) only in the software partition. This decision simplifies the execution model as it ensures that any operation that performs the synchronization of threads must be performed in software (i.e. *monitorenter* and *monitorexit*). Overall, this results in only compute intensive methods receiving execution in hardware. The impact of these two decisions on context switching between partitions is discussed in Section 2.4.

2.1 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 operators, 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 [11]. A detailed justification and list of the bytecodes supported in the hardware partition can be found in [9].

2.2 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*, *checkcast*, and *instanceof*.

2.3 Hardware/Software Interface

The interface between the hardware and software has a direct effect on the design of both partitions. In this case, it was decided to maintain a minimal amount of communication between the two partitions when either is computing. Thus, the design is handed addresses to locations where the necessary data exists within local memory on the FPGA card. Once the hardware partition is finished execution, it signals the software using an interrupt, 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.

2.4 Selective Context Switching

The result of duplication in functionality between partitions is a two level partitioning scheme. Because the functionality of bytecodes supported in the hardware partition is also supported in software, there is a run-time decision as to where execution will take place. Depending on choices made, this partitioning schema can result in an application being executed exclusively in software or alternatively dispersed between the two partitions.

There are various factors that contribute to the decision process. One of these factors is the computation ratio between the hardware and software processing elements. A second major factor is the communication speed between the computation elements. Clearly, these two factors are in contrast to each other and a desirable solution must find the correct balance between them.

To perform this run-time decision, different algorithms can be used to control the flow of execution between the two processing units (CPU and FPGA) [8,10]. The goal of the algorithm is to i) maximize the execution performed in the faster hardware partition while ii) minimize the instances where execution is transferred between the hardware and software components. This is based on the reasonable assumptions that the custom hardware circuit is faster than the software processor, and that the communication link between them is relatively slow. Maximizing the execution in the hardware partition is desirable since it provides more instruction throughput than the software partition. Minimizing the transfer of execution between partitions is desirable since this transition presents an overhead penalty for moving the necessary data for execution.

3 Co-Design JVM Concerns

With a co-design system of such magnitude there are several unknowns that can affect the success. This makes prototyping a worthwhile task in the overall process. With the availability of a software JVM, the primary concern is prototyping the hardware partition. For this project it was decided to simulate the hardware architecture using a custom software simulator. Several reasons for this include:

- The ability to have a flexible interface between hardware and software. This flexibility will allow greater analysis of the interface between the partitions.
- The same is true for analyzing the requirements of the communication rate between the partitions. A simulated hardware design allows for greater investigation into different communication rates between the hardware and software partitions.
- Several questions are raised concerning the FPGA and its suitability for this purpose. Is the FPGA sufficiently large? Is the FPGA fast enough? When targeting a physical environment, the capabilities are fixed. In a simulation environment there are no physical constraints, but we can analyze for relative thresholds.
- It is necessary to analyze the integration between hardware and software. This requires the ability to easily integrate the two partitions. This integration is more easily realized using a software simulation due to its inherent flexibility.

- Targeting a specific platform environment can result in encountering technical difficulties with the environment. Technical issues such as these are not the focus of the work and in a physical environment can result in loss of time or project failure.

Additionally, there are the normal benefits of simulating over implementing that include the following:

- Lower costs, as simulating requires no special hardware.
- Better software support, as support in software is more dynamic and extensive than in hardware.
- Fewer environment quirks. Software allows a generic environment, where a hardware implementation requires the design to involve its quirks.
- Faster development time, as typically software development is faster than hardware implementation.

Overall, this flexibility allows for design space exploration which is crucial to this process.

Various simulation environments already exist for simulating hardware designs. Unfortunately, there are two major factors that suggested using a custom simulator. The first is the complexity of the software component. The software component in this co-designed system is very intricate and relies upon certain functionalities available through the host operating system, namely scheduling and memory management. Running the co-designed software in an encapsulated simulator would not provide realistic results. Secondly, the tight integration between the software and hardware components requires the intricate integration of the software components with the hardware simulator. With the low level dependency between the hardware and software partitions, it is unclear if the available simulators would support and allow investigation of this communication. For these reasons, it was decided to build a custom simulator in software.

4 The Co-Design JVM Simulator

This section discusses various techniques used to implement the software simulator of this hardware design. Each of these techniques is a step towards not only achieving a correct simulation timing at the clock level, but as well to help the later implementation become an easier task. Specification of the simulation at this stage is expected to assist during specification of the subsequent implementation.

4.1 Simulator Goals

Overall, the simulator's purpose is to give an indication of the potential performance of the co-design JVM. To ac-

compish this there are several smaller goals that the simulator must strive to achieve to provide an accurate indication. For this simulation these goals are:

- To model the pipeline stages of fetching, decoding, and executing Java bytecodes in parallel.
- To model the various data caches that exist in the design and provide flexibility for investigation into the effects of varying sizes.
- To model the communication interface between the hardware design (FPGA) and the software partition (host processor) through the PCI interface.
- To model the memory available to the hardware design (FPGA) through a realistic interface [18].
- To model the interface between the hardware design and the host workstation memory subsystem.
- To model the different execution stages of each instruction that is supported by the hardware design.
- To provide a reasonably fast simulation of the hardware design.
- To provide an accurate simulation of the hardware design.

To best achieve these goals, it is suitable for the simulator to leverage known characteristics of existing hardware components. Likewise, it is desirable for the simulator to be based upon a specification language that is synthesizable into a hardware implementation.

4.2 Simulator Design Overview

It was decided to base the simulation on the VHDL behavioral model [4]. Limiting the usage of C in the implementation to only the subset of constructs that are supported by VHDL can contribute towards a later effort of converting the specification to VHDL if deemed desirable. Some additional effort is necessary to provide support for VHDL constructs that are not directly available in C.

The simulator performs a time-driven simulation of the hardware design for the Java virtual machine. In this simulation, each of the different components in the design executes for one clock cycle and then interchanges signals that relay information between the components. Each of the different components in the hardware design is either implemented as a custom defined component, or modeled using some other existing components.

4.3 Signal Propagation

Using the VHDL behavioral model, it is possible to specify each of the different components in the hardware design as its own process. To provide an accurate simulation of the process concept within the VHDL language, the

simulator is implemented using a distinct function to encapsulate the description of each hardware process (or component). To support the VHDL specification model further, signals between hardware components are implemented using two global shared variables. One variable possesses the state of the signal at the current time t , and the second variable holds the value of the signal at time $t+1$. Using this technique, the setting of signals can be delayed until each of the components has executed for the equivalent of one hardware cycle. Thus signal assignments are delayed and propagated at the appropriate time.

4.4 PCI Interface Model

To ensure a correct and realistic simulation, the interface to the hardware design is wrapped by the interface definition of a commercial IP component [18]. This interface in turn wraps the Xilinx PCI interface as provided from Xilinx [19]. In simulation, a wrapper is used to provide the same interface to all external resources. This wrapper provides and ensures not just the same signals, but also the same properties. External RAM that is located on the FPGA card is accessed through this interface. So too is the interrupt signal to software to indicate the hardware has completed some assigned task. To complete the simulation, the interface wrapper also incorporates all of the appropriate delays associated with the signals.

Initially, the interface contained only a host to FPGA communication direction flow, with communication in the other direction being performed by writing to the on-board RAM and signaling the software. Through a simulation communication in the reverse direction can be tested. The interface for supporting interactions with the host systems memory is extended to contain the same interface characteristics used to access the external RAM located on the FPGA card. This interface consists of an address bus (32 bits wide), a data bus to deliver the data to and from the memory (32 bits wide), and three bit signals to indicate the desired operation, read or write. The three signals replicate the same control signals used by the interface to manipulate the external memory on the FPGA card. The use of the same three signal specification is for consistency.

For the delay in accessing the memory, it was decided to allow the simulation to be configured for a fixed delay by the user. A variable delay is also possible, by introducing a random function into the macro definition of delay in the implementation. This delay can be used to simulate the effects of operating over different communication connections as presented later in this paper. This is used to determine threshold communication rates between hardware and software, and the benefits of using a faster connection.

4.5 Modeling Memory Caches

Within the hardware design there are several data caches. To implement these caches it was decided to model existing support for memories found in the Xilinx Foundation environment. Using this feature of the development environment required modeling the memories interface and timing characteristics according to the specification provided by Xilinx. This interface works with a 1-cycle read, a 3-cycle write and can support both reading and writing simultaneously as specified by Xilinx [19].

4.6 Primitives Enforcement

The simulator specification uses only basic operations and data manipulations that are supported by the VHDL model. All of the constructs used in the C implementation are directly transferable into constructs of the VHDL language. There is no formal checking to ensure that this is upheld in the simulator's implementation. This is avoided however, by using sound software engineering practices, code review, and using interfaces to other components correctly. Proper use of the interfaces for the PCI interface and memory often exposed timing idiosyncrasies between components and violations of primitive operations.

What is uncertain is the number of clock cycles required to perform some of the operations involved in the instructions themselves. Depending on the implementation techniques used by the designer, these instructions can require a different number of clock cycles. For example, a designer could choose to have a double precision multiplication instruction occur in as little as 1 cycle, or as many as 10 for instance. In these instances a delay can be incorporated into the simulator to acquire the correct timing requirements.

For the simulation results presented in Section 5, it was decided that no delay would be added and operations would take a base of 1 clock cycle to complete beyond the number of clock cycles necessary to interact with the caches in fetching operands and storing results. This decision is justifiable since it makes no assumptions about the technology or other components used (such as a floating-point unit). From the base time acquired through simulation, the analysis can factor in additional time required for completion of operations once the full target technology and components are known. This is also suitable since different clock rates need to be considered for other factors.

4.7 Simulator Validation

To validate the simulator for correct execution, the result of execution through simulation was compared against the expected result of execution gathered from software execution. This is made possible since an already validated software implementation exists and is available. With any

given execution in the hardware partition, the results are stored into the stack, local variables, and constant pool. Through duplicating these memory regions, it is possible to perform duplicate execution of a given block of bytecode. Comparisons can then be performed with the different memory regions of execution to confirm that both regions produce identical results. This technique was used with all of the benchmark tests at each transition from hardware to software execution. It greatly improved the process as the specification was thoroughly tested and errors in the specification became rather trivial to locate and correct.

Other precautionary measures can also contribute to the validation process. One example is to clear all interconnection signals between components to contain default values. This can be used to ensure communication between components is timed correctly and only happens through the proper supported interface. Likewise, the execution order of the hardware processes being simulated can be interchanged. Doing so ensures that no illegal interconnects or assumed ordering is being used and that signal propagation is executing correctly.

5 Simulator Experimental Results

The hardware design, as provided through the custom simulator, is tested through two sets of benchmarks. The first set of benchmarks implement different algorithms to evaluate the functionality and performance of various hardware features in an isolated environment. These tests include: 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 data from/to the data cache, and the access to the host system memory are all tested and verified.

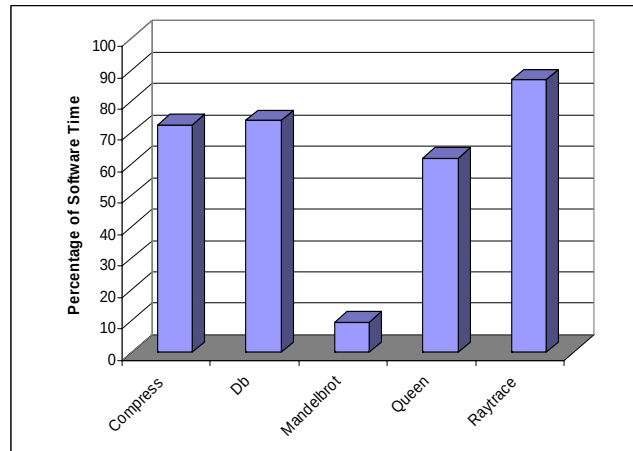


Figure 1: Co-Design vs Software JVM performance

The second set of benchmarks, some of which chosen from the SpecJVM 98 benchmark set, utilize the simulator in a full co-design JVM. These tests include: Raytrace; Db, Compress; n-Queens problem; and Mandelbrot calculation. These tests are used to examine the full co-design JVM with interactions between the hardware and software partitions.

Figure 1 shows some of the overall performance gains that are observed when using the co-design JVM in comparison to a full software implementation of the JVM. The simulator confirms that the overall co-design JVM approach is sound, but more importantly it allows investigation into the conditions necessary for success. Some of these conditions include [13]:

- The hardware design typically must operate at a clock rate no less than 5 times slower than the host processor.
- The hardware design must support the full partition to achieve performance gains for most applications.
- Local memory requirements of the hardware partition are minimal, typically around 12Kb.
- Accessing the host system's memory from the hardware design can tolerate up to 50 cycles while still improving performance.
- To offset the communication penalty of the PCI bus when switching execution, the execution in the hardware partition should be approximately greater than 8300 instructions.

These results overall provide sufficient insight to suggest that following through with an implementation is worthwhile, but also indicate some of the issues that require consideration during implementation. Mainly these are to: realize the full partitioning scheme; attempt to maximize the clock rate; and to minimize the number of execution transfers between hardware and software. These considerations not only affect the implementation of the hardware partition, but also the environment that is to be used.

6 The Co-Design JVM Implementation

Using the specification of the hardware partition as described for the custom simulator, the process of deriving a synthesizable implementation for use on a FPGA is rather direct. By structuring the simulator as described above and adhering to the VHDL model, the process was completed rather quickly and involved little design effort.

From the results gathered through simulation, there were two changes decided upon for the implementation. These changes reflect the target environment in which the

hardware implementation is realized [1]. Both changes are discussed in detail in the following subsections.

6.1 Local Memory

The target environment used provides SDRAM as the local memory to the FPGA with a SDRAM controller for interfacing. In the implementation, the memory space within the FPGA device is used rather than the on-board SDRAM. This is due to the slow access to the on board SDRAM caused by the burst mode SDRAM controller IP that Altera provides [3]. Therefore using on-chip memory achieves better performance. This design decision is possible from the results that were gathered through simulation. With the amount of local memory provided by the FPGA device, sufficient memory is available to contain the maximum Java program frame as indicated by the benchmarks.

6.2 PCI Interface

Originally, the hardware partition was designed targeting the [18] environment. With the results of simulation it was clear that a much larger FPGA device would be necessary and hence a different target environment is chosen [2]. The Altera `pci_mt64` MegaCore function is used as the interface with the PCI bus. This requires altering the hardware partition interface to interact with this core. The PCI interface is composed of the local master transaction control and local target transaction control logic, as well as the interface with the memory on the FPGA.

The local target control logic 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 local memory on the FPGA.

The local master control logic triggers the `pci_mt64` function to work under master mode and the data flows from the local master 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 initialized 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 [8,10].

7 Implementation Experimental Results

Testing of the implementation is performed utilizing the same set of benchmarks as was used by the custom simulator. The tests are performed with the simulation tool, Mod-

elSim, and the results are compared to that of the simulator that was previously implemented [15]. The implementation provided the same results as the simulator with respect to correct execution and the number of clock cycles to perform the actual computation in hardware. Only one difference existed and this was a result of changing the PCI interface as mentioned in Section 6.2. With the implementation, it is now possible to gather information with respect to the design space and speed that can be obtained.

7.1 Design Space

This design architecture provides a flexible solution by utilizing one of several different partitions or by changing the size of the Instruction or Data Cache. The design space, both Logic Elements and Memory bits used for different configurations, is listed in Table 1. The first three configurations implements the three different partition schemes with the same size of cache: both the instruction cache and the data cache are 64 entries. The last configuration implements the full partition, but with a smaller cache setting: both the instruction cache and the data cache are 16 entries.

	Number of Instructions	Logic Elements	Memory Bits
<i>Compact</i>	132	30,743	133,120
<i>Host</i>	148	33,110	133,120
<i>Full</i>	161	37,756	133,120
<i>Full (Smaller Cache)</i>	161	33,490	131,328

Table 1: Design Space for Different Configurations

7.2 Design Speed

Through timing analysis, the maximum clock rate achieved is 24 MHz. The speed is not competitive to the host processor as was desired due to some critical routes in the Execution Engine. Some of the instructions implemented are time consuming therefore reducing the overall clock rate. Despite the low performance, from the investigation performed before several benchmarks are capable still achieving a performance increase under certain circumstances [9]. This performance may be increased further as potential parallel processing between the hardware and software partitions can now be explored.

8 Future Work

The work completed to date has provided many promising results. This work is being continued in two directions.

First, the implementation of the simulator in hardware is undergoing several optimizations. Most notably is the use of available logic elements on the FPGA instead of memory bits for local caches. This modification will reduce data access times while still fitting within the target FPGA. The impact of this, and other optimizations, on the overall clock rate is not known. Second, this research has shown that it is possible to fit multiple Java hardware engines into a large FPGA. Our simulator has already been modified to support multiple Java hardware engines and the simulated results are promising. The continuation of this work will involve the design and development of an arbiter module that will share resources between both processing elements.

9 Conclusions

The methodology described in this paper of developing a custom simulation of the hardware partition within a described framework proved to be very effective. This allowed the rapid determination of feasibility of the co-design JVM and the constraints for its success while avoiding technical issues encountered in an implementation. More importantly, the simulator provided a quick design process and a smooth transition to a synthesizable implementation. The implementation supports the results obtained through the simulator reinforcing its usefulness. With a now complete implementation, further research is now possible into the effects of utilizing both the hardware and software partitions in parallel.

References

- [1] Altera Corp.; *APEX PCI Development Board Data Sheet*. PCI Development Kit Version 2.1.0. CDROM, April 2002.
- [2] Altera Corp.; *PCI MegaCore Function User Guide*. PCI Development Kit Version 2.1.0. CDROM, August 2001.
- [3] Altera Corp.; *SDR SDRAM Controller White Paper*. PCI Development Kit Version 2.1.0. CDROM, May 2000.
- [4] Ashenden, P. J.; *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 1996.
- [5] Cardoso, J. M. P.; Neto, H. C.; *Macro-Based Hardware Compilation of Java Bytecodes into a Dynamic Reconfigurable Computing System*. IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999.
- [6] El-Kharashi, M. W.; ElGuibaly, F.; Li, K. F.; *A New Methodology for Stack Operations Folding for Java Microprocessors*. High Performance Computing Systems and Applications, chapter 11, pp. 149-160, Kluwer Academic Publishers, 2000.
- [7] Gu, W.; Burns, N. A.; Collins, M. T.; Wong, W. Y. P.; *The Evolution of a High-Performing Java Virtual Machine*. IBM Systems Journal, vol. 39, no. 1, pp. 135-150, 2000.

- [8] Kent, K. B.: *Branch Sensitive Context Switching between Partitions in a Hardware/Software Co-Design of the Java Virtual Machine*. IEEE Pacific Rim Conference on Computers, Communications and Signal Processing (PACRIM) 2003, pp. 642-645, August 2003.
- [9] Kent, K. B.: *The Co-Design of Virtual Machines Using Reconfigurable Hardware*. Ph.D. Dissertation, University of Victoria, 2003.
- [10] Kent, K. B.; Serra, M.: *Context Switching in a Hardware/Software Co-Design of the Java Virtual Machine*. Designer's Forum of Design Automation and Test Europe (DATE) 2002, pp. 81-86, March 2002.
- [11] Kent, K. B.; Serra, M.: *Hardware Architecture for Java in a Hardware/Software Co-Design of the Virtual Machine*. Euromicro Symposium on Digital System Design (DSD) 2002, pp. 20-27, September 2002.
- [12] Kent, K. B.; Serra, M.: *Reconfigurable Architecture Requirements for Co-Designing Virtual Machines*. Reconfigurable Architectures Workshop (RAW) part of the International Parallel and Distributed Processing Symposium (IPDPS) 2003, April 2003.
- [13] Lindholm, T.; Yellin, F.: *The Java Virtual Machine Specification (2nd edition)*. Sun Microsystems Inc., 1997.
- [14] Ma, H.; Kent, K. B.; Luke, D.: *An Implementation of the Hardware Partition in a Software/Hardware Co-Design of the Java Virtual Machine*. to appear in the Canadian Conference on Electrical and Computer Engineering (CCECE) 2004, May 2004.
- [15] O'Connor, M. J.; Tremblay, M.: *picoJava-I: The Java Virtual Machine in Hardware*. IEEE Micro, pp. 45-53, April 1997.
- [16] Suganuma, T.; Ogasawara, T.; Takeuchi, M.; Yasue, T.; Kawahito, M.; Ishizaki, K.; Komatsu, H.; Nakatani, T.: *Overview of the IBM Java Just-in-Time Compiler*. IBM Systems Journal, vol. 39, no. 1, pp. 175-193, 2000.
- [17] Virtual Computer Corporation: *Hot-II: Hardware API Guide*. Virtual Computer Corporation, 1999.
- [18] Xilinx Corporation: www.xilinx.com. 2003.