Reconfigurable Architecture Requirements for Co-Designed Virtual Machines

Kenneth B. Kent
University of New Brunswick
Faculty of Computer Science
Fredericton, New Brunswick, Canada
ken@unb.ca

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

Abstract

This paper addresses the physical hardware requirements necessary for a co-design hardware/software virtual machine to not only exist, but to also provide comparable performance with other implementation techniques for virtual machines. The discussion will center on requirements of the reconfigurable device and it's peripheral connections to main memory and the general-purpose processor.

1 Introduction

To provide context for the requirements, the idea of a co-designed virtual machine must first be given. A virtual machine for the purposes of this paper is considered to be a general-purpose computing platform that does not physically exist. For example, the Java virtual machine or the Common Language Runtime (CLR). The virtual machine consists of more or less a high-level operating system and a low-level computer architecture. Depending on the virtual machine implementation, the division between the two can be clear or very imprecise. Virtual machines have found a niche in general purpose computing as it provides flexibility in updating and revising the hardware architecture that is not easily available with a physical processor. Homogeneous computing also utilizes virtual machines because of their tremendous level of portability while maintaining a common platform interface.

A co-designed virtual machine looks to exploit the natural split in the virtual machine of hardware and software. While keeping the same common platform interface and portability, the virtual machine is provided through not just a software implementation, but a software and hardware one. The high-level operating system constructs and functionality remain in software, executing within a general-purpose processor, but the underlying hardware architecture is provided through the use of a reconfigurable device. Utilizing the parallel nature of hardware, the performance of the virtual machine can be increased by simply executing the fetch-decode-execute cycle in hardware [6]. An abstract view of the new system can be seen in Figure 1. The potential for a performance increase can be immediately seen in this new paradigm. The use of a dedicated hardware implementation in general can provide a performance increase over a software implementation executing in a shared multiprocessing environment. There now exist two processing units, the reconfigurable device and the general-purpose processor, making parallel execution possible.

This approach is not desired to rely upon a specialized system for its success, other than the reconfigurable device itself. All of the required resources that are utilized in the abstract view of the platform are readily available. Cheap workstations are currently being used to execute the software implementation. A desirable solution is to simply integrate the reconfigurable device into the workstation. What is unclear however are the requirements of the reconfigurable device and its communication interconnects. For this, there are several questions that must be examined:

- Device size. What is the size of the device required in order to provide a performance increase?
- Device speed. What is the device speed needed in order to compete with the speed of the general-purpose processor?
- Memory size. How much memory is required for the reconfigurable device to perform its computations? Does it need access to all of the virtual machine data?
- Memory speed. What is the required communication rate between the reconfigurable device and the memory system?
- Communication rate. What is the bandwidth needed to support communication between the general-purpose processor and the reconfigurable device?

The remainder of this paper will address each of these questions and try to provide an answer.

Within this environment, the reconfigurable device is not investigated for its capabilities in reconfiguration. For the purposes of this research, the reconfigurable device is used in a static context. While it may be suitable for the underly-

ing hardware architecture of some virtual machines to change, it may not be in others. For this reason a virtual machine specific static configuration will be used on the programmable device.

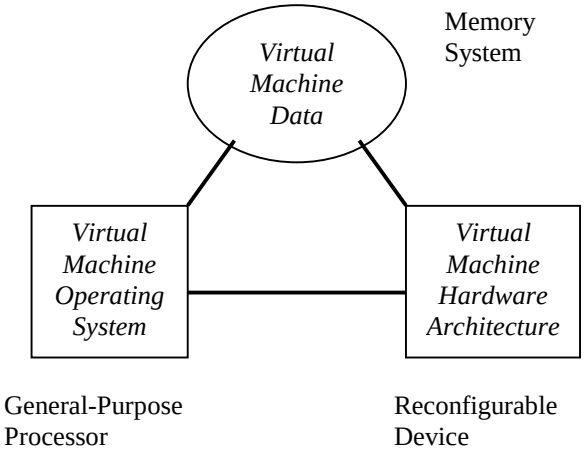

Figure 1 An abstract view of a co-designed virtual machine.

2 Java Virtual Machine

It can be quickly seen that the answers to each of these questions are significantly dependent upon the specific virtual machine. In some virtual machines, the relationship between the operating system and hardware architecture is loose and does not require significant communication between them for the majority of the execution time. While other virtual machines may rely upon a much tighter integration between the two partitions. While it is possible to extract some requirements information from an abstract examination, it is more fruitful to use a test case virtual machine. It was decided to look at these questions specifically in respect to the Java virtual machine.

The Java virtual machine was chosen for several reasons. First, that it is reasonably representative of virtual machines in general. It contains both operating system and hardware architecture attributes, and these are both clearly divisible. Second, the Java virtual machine is a mature test case. Much research has been conducted on the Java virtual machine and its performance issues [1,3,4,9]. Using this test case can also show the potential performance increases in relation to other acceleration techniques. Finally, there is also readily available source code for the virtual machine to assist in the investigation and standard benchmarks exist for measuring performance and testing the execution correctness. To test the co-designed Java virtual machine and generate performance data that can be analyzed, the SpecJVM test suite was used [10]. Unfortunately, not all of the tests from the test suite could be used due to the limited capabilities of the simulation environment for handling multithreading. Figure 2 shows the performance of each of the benchmarks used in relation to the original execution times and the performance increases obtainable under an ideal development environment[1].

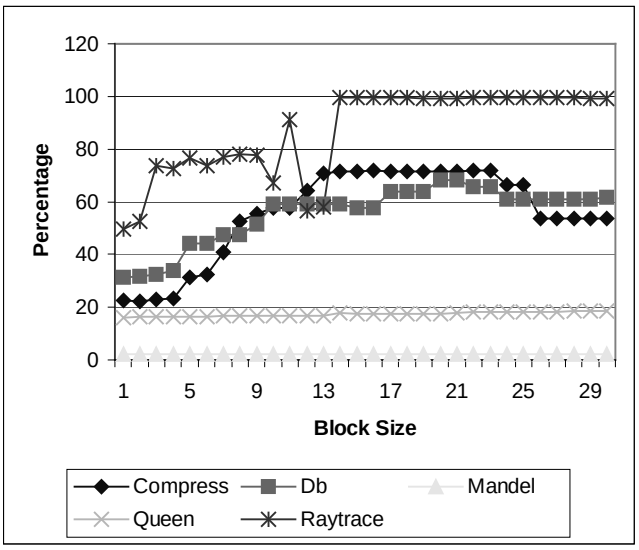

Figure 2 Overall performance increases attainable from a co-design Java virtual machine running under ideal conditions.

The co-designed Java virtual machine that will be used for this inspection of necessary requirements has several interesting features. Details of the hardware design can be found in [8]. The hardware and software partitions are overlapping. This is not the case in traditional co-designed systems. This overlap was used to allow the virtual machine to perform selective run-time execution migration between the two processing devices. To make the decision during run-time of when to transfer execution, several algorithms were developed and tested [5]. These algorithms aimed to find an optimal balance between two offsetting goals. First, to maximize the percentage of time spent executing in the faster hardware partition. Second, to minimize the number of times execution is transferred between hardware and software. Transferring execution is an overhead penalty that does not contribute to the actual execution of the application. Therefore, minimizing the penalty is advantageous.

[1] Block size is the variable used for run-time scheduling. It represents the number of sequential instructions in the method that can be executed in the hardware component for execution to migrate from the software to hardware.

3 Simulator

For this co-designed virtual machine, an initial platform had to be chosen for the research to be conducted. It was decided to have it based on a Field Programmable Gate Array (FPGA) connected to a workstation via the systems PCI bus [2]. The reasoning for this decision was based on both the wide availability of PCI based development environments for reconfigurable computing and that the PCI bus is the most common and inexpensive means to make the connection in a typical workstation [11]. While this is a reasonable basis upon which to conduct the research it does not imply it is an ideal platform.

For this reason it was decided to use a custom simulation environment to provide the hardware design. This will allow us to overcome some of the potential constraints of the platform and provide flexibility to precisely monitor the overall environment for suitability in all its aspects. For instance, examining the effects of increasing hardware support through a larger FPGA while keeping all other factors constant. These factors include memory configuration, size of the FPGA, overall speed of the hardware design, and communication rates between the FPGA and both the memory and software processor. Each of these factors can be examined for their effects when changed individually or together.

The simulator for this research is a custom software simulator. The reason for using a custom built simulator, instead of an already existing simulator, was to allow for easy integration with the software components. While there are other simulators readily available, they do not allow the required integration with the software components. This is especially important since the software components also rely upon the low level operating system for task scheduling and other support mechanisms.

3.1 Simulator Implementation

This section discusses various techniques used to implement the software simulator for the hardware design. Each of these techniques are steps towards not only achieving a correct simulation timing at the clock level, but as well to help facilitate an eventual implementation.

The simulator is based on the VHDL behavioral model, but written in C to allow for direct compilation and execution in software. The C language was chosen due to its ability to provide fast execution and its low-level support. It is important to note that only the constructs in the C language that are directly supported by the VHDL language were used.

To promote a proper hardware design, the design is implemented using a distinct function to encapsulate the description of each hardware component. Signals between hardware components are implemented using two 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 have executed for the equivalent of one clock cycle. Thus signal assignments are delayed and propagated at the appropriate time.

Various components that are modeled in the simulator are described using characteristics of available components. The PCI interface simulated contains the same timing characteristics of the provided Xilinx PCI interface in the Hot-II development environment [11]. This is also true for the controller to access the memory provided on the FPGA card, and any internal caches modeled after the Xilinx LogiBlox memories [11,12]. These components are also restricted to provide legitimate services. For example, the caches are constrained to provide only sizes that fit within those attainable with the Xilinx Foundation tools [12].

While the simulator is targeted specifically for the example Java virtual machine, the design is aware of its possible use for other virtual machines. The simulator maintains a design that reflects the overall design of the architecture with the fetch-decode-execute paradigm. Provided that another virtual machine's hardware architecture is based on the same principle, then the simulator can be successfully changed and reused.

4 Reconfigurable Device Requirements

For the reconfigurable device itself, two criteria are of importance. The speed of the device when configured with the hardware design and the size of the device needed to hold the hardware design.

4.1 Speed Requirements

It is obvious that the faster the FPGA the better. What is unclear is the threshold for how fast the FPGA must be in order to provide a performance increase. For discussion of the required speed for the FPGA, the speed relationship between the FPGA and host CPU will be used. Traditionally, FPGA speeds have been three to five times slower than that of processor speeds.

To simplify the investigation, each of the benchmark results is examined without the communication costs. In these cases, a performance increase was seen for all of the benchmarks, but not for all FPGA speed ratios that fall within the typical ratio between processor and FPGA. Table 1 shows for each benchmark and partitioning scheme the threshold FPGA speed ratio. These ratios indicate the maximum number of cycles the software host processor can execute for each single cycle the hardware design can execute. If the

cute. If the software processor can execute a higher number of cycles for each cycle the FPGA can execute, then a performance decrease will be seen. If the software processor executes less cycles than the threshold for each cycle executed by the FPGA then a performance increase will occur.

With an FPGA that is up to a factor of five times slower than the host processor a performance increase is possible for almost all of the results shown in the table. It can be clearly seen that the lowest threshold value is that of the Mandelbrot application under the compact and host partitioning schemes. Even in this case, the ratio is within the traditional bounds of the speed offerings of FPGAs. These results show that current available speeds of FPGAs are potentially capable of being used in this capacity.

Partitioning	Compress	Db	Mandelbrot	Queen	Raytrace
Compact	5.75	5.47	4.12	4.70	6.29
Host	5.78	6.24	4.12	5.47	6.34
Full	7.19	7.43	53.30	8.32	6.37

Table 1 Threshold speed ratios between the software processor and the hardware design.

4.2 Space Requirements

Though the simulation environment is incapable of providing a quantitative assessment of the design space requirements, the results obtained can be used to project some insights into the qualitative size of the FPGA. Specifically, the partitioning scheme it must be capable of supporting. For all tests, under all variations of parameters, the performance of the co-designed virtual machine increases with the greater level of hardware support. From the previous table, it can be seen that for each benchmark the Full partitioning scheme out performs the Host partitioning scheme, which in turn out performs the Compact partition. Thus, in general, the larger the FPGA the greater the performance increase.

This is only true, however, when the communication costs are negligible. When the communication costs rise to a significant level, the driving characteristic behind a partitioning strategies success is its frequency of transferring execution between the partitions. This frequency is indirectly determined by the partitioning scheme, and directly by the density of the instructions supported in hardware. Tests have shown that the larger the hardware partition, typically the more execution transfer between the partitions.

Both the size and speed requirements of the programmable device are linked. Having a slow FPGA requires a larger hardware design space for a performance increase to be attained. Likewise, a fast FPGA does not require as large a design space area. A prime example of this is the Mandelbrot application. Provided that the FPGA can support the design space required of the Full partitioning scheme, a performance increase can be obtained despite as large a difference in clock ratio of 53:1, as shown in Table 1. For the host system that was used in this research, a 750 Mhz Intel Pentium, that translates into a required minimum 15 Mhz hardware design. This is also true for a small FPGA that can operate at a high clock rate.

4.3 Memory Requirements

For the co-designed virtual machine, performance can be affected by the memory space that is available for the hardware and software components. In the development environment used, the memory space utilized is not unified, but rather split between each components local memory region. For the software partition of the virtual machine the available memory space is not an additional concern because it would provide the same memory resources available to a software only virtual machine. However, for the co-designed virtual machine, the distinct local memory available to the FPGA is typically constrained and may present problems. In the event that not enough memory is available to the hardware component then execution would remain in software, thus under utilizing the hardware partition.

In the case study Java virtual machine, the amount of data transferred between the hardware and software components during execution of each of the benchmarks was recorded. For all of these benchmarks, the maximum amount of data used by the hardware partition was 11312 bytes. This includes the method to execute, the local data variables, and the data stack with sufficient room for growth to the maximum stack size. The common amount required for all benchmarks and partitioning is most likely a result of the same underlying Java API method being executed by all benchmarks. Though the specific memory requirements are application dependent, this demonstrates that in general for the Java virtual machine the memory requirements for the hardware component are substantially low considering the available 4 Mb of local memory in this particular development environment.

4.3.1 Host Memory Accessing Requirements

The development environment that simulation originally targeted did not have the ability to access the host memory system. With the vast amount of memory required by most applications for data, having the programmable device capable of accessing the host memory system is desirable.

There are other possible architectures that can provide this capability. To allow for exploration of the methodologies effects, the simulator was built with the capability.

For the purposes of simulation, the protocol for accessing the host memory system was treated identically to accessing the local memory on the PCI card. This protocol has a 3 cycle delay associated with it for enabling the memory and setting the requested address plus the additional delay of the bus. With this specification the co-designed performance results were collected. These results can be used to determine for each of the benchmarks the delay that can be tolerated before the execution crosses the threshold and degrades performance. Figure 3 shows the threshold number of cycles for each of the benchmarks under the full partitioning scheme, along with the different speed ratios, where accessing the host memory system is vital. If the delay in accessing the host memory is above the threshold value, then the application will execute slower in the co-designed virtual machine. This figure does not factor in the on-chip caching of data once it is initially accessed by the hardware component. Included in the figure are the different thresholds for each of the performance ratios between raw computing elements that have a direct effect.

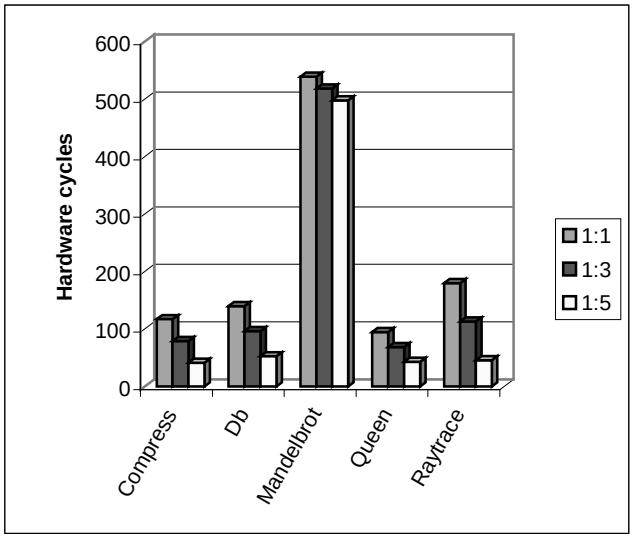

Figure 3 Threshold values for communication delays of accessing memory from the host system.

These results show that accessing a common memory store with a delay is tolerable. However due to how frequently the memory is accessed, the co-designed virtual machine can only tolerate an average delay of up to 50 cycles for each access. Beyond this delay performance begins to degrade in comparison to the software execution. It suffices to say that the development environment used which requires accessing the host memory through the PCI bus is not viable. Tests showed that an average of 8760 cycles were required to retrieve a 32-bit word of data across the PCI bus. It is only through the use of the on-chip data cache that the Mandelbrot application is capable of tolerating the slow PCI bus to provide a performance increase.

5 Communication Requirements

With the tight relationship between the hardware and software components, it is extremely important that the communication link between them be sufficiently fast. In the event that the communication medium is relatively slow, any performance gains achieved by hardware execution over software execution can be overshadowed. The communication speed requirements are directly affected by the application running within the virtual machine, and are thus very instance specific. If the application demands frequent execution migration and high levels of data exchange between hardware and software components, then the more critical the demands on the communication medium.

Different techniques were employed in the overall co-design process to minimize the communication between hardware and software. The overlap in partitions along with run-time scheduling ensures that execution migration only occurs when it is significant and a performance increase is feasible. Communication was further reduced by simply communicating frequently used data and in some cases, only data that was changed. Each of these techniques contributes to significantly less bandwidth, but certain requirements still exist.

Block	Compress	Db	Mandelbrot	Queen	Raytrace
1	2287%	1900%	11.3%	171%	6365%
3	2190%	1668%	11.2%	162%	2624%
5	1652%	1113%	10.8%	137%	2276%
7	1325%	894%	10.8%	136%	2499%
9	879%	741%	10.6%	130%	2381%
11	737%	552%	10.6%	109%	1032%
13	386%	550%	11.0%	127%	4853%
15	389%	678%	11.0%	103%	146%

Table 2 Percentage of original execution time when including the communication penalty over the PCI bus.

Table 2 shows that only one of the benchmarks, Mandelbrot, shows a significant performance increase despite the communication costs of the PCI bus and this is due to

certain characteristics of the application itself. Most other benchmarks show a high performance decrease because of the communication. Examining the Queen benchmark, the execution is close to matching the original execution time. In this case, the average number of hardware cycles per context switch is 8299 with an average of 879.94 instructions per context switch. Though these numbers are dependent on both the type and ordering of instructions, it does provide an estimate of the hardware support density needed to obtain a performance increase. The overall results show that the PCI bus hinders the execution and is the cause for a performance decrease.

There are several underlying reasons for the PCI bus being unsuitable for usage in a co-design virtual machine environment. The most obvious problem is that the bus is shared with other devices. Sharing the bus results in unnecessary delays when waiting for the bus arbitrator to hand over control of the bus. This is especially true when the PCI bus typically holds relatively high bandwidth hardware components such as the audio, video and network devices. In comparison with the relative speeds of the hardware and software computing elements, the PCI bus is exceptionally slow. For the specific development environment used in the case study, the PCI bus operates at 33 Mhz, while the FPGA operates at speeds up to 100 Mhz and the host processor at a phenomenal 750 Mhz. With such a high disparity between the communication and operating speeds the necessary communication between the partitions results in a drastic performance penalty.

6 Conclusions

While the idea of providing the FPGA that implements the hardware design on a bus that has comparable speeds with that of the FPGA itself or the software processor may not be feasible, a better overall architecture is certainly more attainable. Ideally, to have the FPGA device directly attached to the mainboard of the host system on a dedicated bus would provide a considerable improvement. Likewise, to have a fast communication bus between the FPGA and the host's memory is also beneficial. As can be seen from the results in the previous section, the methodology does promise varying performance gains for each of the benchmarks without the communication penalty. This demonstrates that for this approach to succeed in general a more suitable architecture must be present.

From the above analysis, it can be seen that while reconfigurable computing is offering a potentially high performance increase, there still remains much work before it can be used for all applications. Specifically, performance is lacking not in the reconfigurable devices themselves, but in the environments within which they are provided. This bottleneck can be addressed by providing the programmable devices through better interfaces. Specifically by providing reconfigurable devices on the mainboard of a system. By doing so, the communication requirements necessary can be fulfilled.

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] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, "Field-Programmable Gate Arrays", Kluwer Academic Publishers, 1992.

[3] J. M. P. Cardoso, and H. C. Neto, "Macro-Based Hardware Compilation of Java Bytecodes into a Dynamic Reconfigurable Computing System", IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999.

[4] M. W. El-Kharashi, F. ElGuibaly, and K. F. Li, "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.

[5] K. B. Kent, and M. Serra, "Context Switching in a Hardware/Software Co-Design of the Java Virtual Machine", Designer's Forum of Design Automation & Test in Europe (DATE) 2002, pp. 81 - 86, March 2002.

[6] K. B. Kent, and M. Serra, "Hardware Architecture for Java in a Hardware/Software Co-Design of the Virtual Machine", Euromicro Symposium on Digital System Design (DSD) 2002, Dortmund, Germany, September 4-6, 2002.

[7] K. B. Kent, and M. Serra, "Hardware/Software Co-Design of a Java Virtual Machine", International Workshop on Rapid System Prototyping, pp. 66 - 71, June 2000.

[8] K. B. Kent, and M. Serra, "A Co-Design Methodology for Virtual Machines", submitted to IEEE Computer special issue on Hardware/Software Co-Design, April 2003.

[9] H. McGhan, and M. O'Connor, "PicoJava: A Direct Execution Engine For Java Bytecode", IEEE Computer, pp. 22-30, October 1998.

[10] SPEC, http://www.spec.org/osg/jvm98. November 1997.

[11] Virtual Computer Corporation. "Hot-II: Hardware API Guide", Virtual Computer Corp., 1999.

[12] Xilinx Incorporated. Xilinx Foundation Series Software. June 2000.