A SIMULATION PLATFORM FOR RECONFIGURABLE COMPUTING RESEARCH

Wenyin Fu, Katherine Compton

Department of Electrical and Computer Engineering
University of Wisconsin at Madison
wenyinf@cae.wisc.edu, kati@engr.wisc.edu

ABSTRACT

In this paper, we present a full-system reconfigurable computing simulation platform intended to promote innovative new research in reconfigurable computing. Currently, reconfigurable computing researchers frequently use tools such as simple trace-based simulators to provide performance estimates for their designs and methods. However, because the scope of these simulations is generally high-level and limited to the reconfigurable hardware and possibly a host processor, the results presented may unfortunately be less than completely accurate, particularly with respect to memory and processor-hardware communication timing. As reconfigurable computing systems become less theoretical and move closer to widespread realization, more accurate evaluations are necessary to compare new system designs, configuration methods, hardware scheduling algorithms, and hardware structures. We therefore discuss the issues involved in implementing a full-system simulator for reconfigurable computing, presenting specific implementation techniques, and demonstrate the value of reconfigurable computing full-system simulation on an example application.

1. INTRODUCTION

A reconfigurable computing (RC) simulation platform would provide a flexible environment to examine new ideas and compare different designs before choosing the best one or combination to implement, either in an FPGA or in one or more custom chips. Modular design would allow simulation of different reconfigurable hardware (RH) designs without changing the core functionality. The use of a modifiable simulation platform would permit designers to alter the coupling of the RH with other parts of the system, such as the processor and memory. A full-system simulator would also allow a degree of realism not available with simpler estimation tools. Currently, researchers tend to use simulators and estimation tools that only model user-space code, and ignore system-level activities. These tools may also lack realistic memory models, and may model non-mainstream ISAs such as Alpha or PISA [2]. In these cases, it can be difficult to contextualize the results in terms of current processor technology, and for odd ISAs, compiling new useful benchmarks may be a difficult and painful process.

In this paper, we present a full-system RC simulator built on top of the Simics platform [1], using the GEMS memory system extension [4]. We discuss the issue of simulation vs. system implementation in section 2. A quick introduction to Simics and GEMS is given in section 3. Section 4 then discusses the modifications we have made to Simics/GEMS to support RC system simulations. We then present the results of executing an AES application in our simulation environment in section 5, and demonstrate how a full-system simulator provides important timing information not available using less sophisticated estimation tools.

Finally, we present a list of future additions we plan to make to our simulation environment. Among them is using one or more real FPGAs to accelerate hardware simulation. This differs from the approach described previously, as the FPGAs would not be a direct representation of the simulated RH, but instead a tool used by the simulation framework to provide functional accuracy more quickly. This and other future plans are discussed in more depth in section 6, and our conclusions given in section 7.

2. WHY SIMULATION?

This simulation platform is not intended to replace the value of implementing an actual system in hardware. However, the design of a RC system is a complex process. Depending on the needs and goals of the designers, the system may be built from commercial off-the-shelf components, including commercially-available FPGAs. In this case, commercial tools provide critical infrastructure. In particular, if the system uses a complete commercial board, the system is essentially already implemented, and may not require simulation beyond the capabilities of the tools that come with it. In other cases, a designer may want to create a fundamentally new system or reconfigurable architecture. Although one can sometimes successfully model a new design on an existing FPGA platform, the difficulty of the task can increase at least proportionally to the difference between the new design and the available FPGA platform.

Differing logic designs may be accommodated by modeling the new design in an HDL and synthesizing it to an FPGA. However, vastly different configuration architectures (such as modeling a partially reconfigurable FPGA on a serially-programmed FPGA) or routing architectures (modeling an FPGA with non-uniform channel widths on one with uniform widths) are much more difficult, or at best, much more space-inefficient to model. The difficulty translates to time-investment, which may in turn bound the number and range of different designs or ideas a researcher or a group of researchers can implement and compare to find the best one. Space-inefficiency may be to such a degree that a significant financial investment in a large and expensive multi-FPGA platform is required in order to emulate a design of any appreciable size, limiting those able to perform this type of investigation, or the scope of investigations. In these cases a simulator allows new architectures to be tested and evaluated in the context of a full system much more easily.

Furthermore, for designs coupling reconfigurable logic with one or more microprocessors, there are a limited number of microprocessor options easily available for commercial FPGA boards and SOPC/Platform FPGAs, restricting the designer's choices. Although several different soft-core processors are offered for use in FPGAs, these tend to have similar capabilities to one another, and generally fall into the simple single-threaded embedded processor category, which may not meet the designer's needs. Options for interfacing reconfigurable logic, processor(s), and memory in commercial systems are similarly limited.

A flexible, modifiable simulation framework permits modeling a wider variety of designs with a smaller financial investment. Virtutech currently provides free licensing of the Simics full-system simulator—the basis of our RC simulation platform—to individual academic students and faculty. The system can be modeled at various degrees of detail, allowing an investigator to perform a quick but less-detailed simulation early in the development of a new idea, and a more sophisticated implementation as the idea becomes more concrete. The simulation platform, because it is full-system, provides a level of accuracy much closer to a real system than a simpler estimation tool, but with significantly less effort than implementing an actual system.

3. SIMICS/GEMS OVERVIEW

Simics [1] is a full system functional simulator developed by Virtutech AB [5] that is capable of simulating the range of hardware in a typical computer system (CPU, memory, disks, network card, etc). It offers sufficient functional accuracy to run unmodified binaries on the the simulated machine, including operating systems. However, Simics by itself is only a functional simulator, and does not provide cycle-accurate timing. Fortunately, Simics provides a rich set of APIs to allows users to extend Simics and provide this support through customized modules.

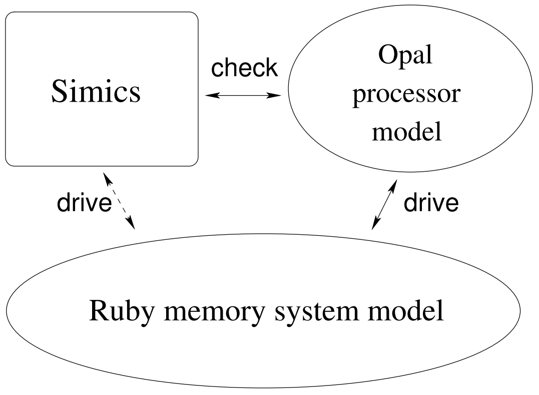

Fig. 1. Simics/GEMS overview.

The Wisconsin Multifacet General Execution-driven Multiprocessor Simulator (GEMS) is a collection of Simics modules that provide timing simulation capability for a SPARC processor and a multiprocessor memory system [4]. The processor timing model (Opal) models a dynamically scheduled SPARC V9[3] processor. The memory system module (Ruby) models caches, cache controllers, interconnect, memory controllers and banks of main memory in a multiprocessor memory system. Ruby can be driven by Opal to provide memory access timing information while Opal itself simulates the timing of the processor as shown in Figure 1. In our work, we extended Opal/Simics to simulate a processor augmented with RH.

4. ADDING RC SUPPORT

We have modified Simics/GEMS to support RC. However, before discussing simulator details, we first define our view of RC. In this type of system, applications are composed of both hardware and software components. The hardware components are swapped in and out of RH as needed, and a host microprocessor executes the software portion of the application. We refer to the hardware-mapped compute-intensive parts of RC applications as application *kernels*. In a multi-tasking or multi-threaded system the need for hardware resources may exceed the availability. Binding kernels dynamically at runtime to hardware or software allows kernels that do not fit in hardware to proceed in software to avoid stalling [8], [9], [10], [7].

Based on our view of RC, we have several goals for our simulation platform. One is to support the dynamic binding of kernels to hardware or software as discussed above. Another is to provide a realistic implementation of RC, modeling it in a way that it could actually work in real hardware. We also need to provide cycle-accurate processor and memory timing information, which is facilitated by the use of

```
  ......
  bkernel kid, L_RC_end; // branch on kernel
  RC_func();             // kernel's SW code
L_RC_end:
  ......
```

Fig. 2. Original concept of kernel invocation [7]

Simics and GEMS. The simulation should be functionally correct, and ideally, the software binaries run in the simulator should be identical to the software binaries that would run on an actual implementation of the simulated system. The following sub-sections discuss implementation issues as they relate to meeting our simulator design goals.

4.1. Special RC Instructions

In our original work on dynamic scheduling techniques for reconfigurable computing [7], we proposed a special branch-on-kernel instruction (`bkernel`) to invoke the kernel execution as shown in Fig. 2. When the requested kernel, indicated by the kernel id (`kid`), is not available in RH, this branch will fall through to execute the conventional software binary. Otherwise, the `bkernel` instruction will trigger the kernel computation in hardware. The mapping between the `kid` and the actual hardware configuration information can be established either at application installation or application execution time. Since different applications may have been compiled to use the same `kid` values, the processor can augment the `kid` with the process id to avoid ambiguity issues. The second operand, `L_RC_end`, is the return address for after a hardware kernel execution, used to skip the alternative software implementation code.

Due to common hardware design techniques such as pipelining, a single call to a kernel's hardware implementation may be the equivalent of several executions of that kernel's software equivalent. In these cases, the application designer (or preferably the compiler) would simply unroll that number of software iterations in the section of the binary for the kernel's software code in order to maintain equivalency. Accommodating more complex differences between the software and hardware kernel implementation interfaces will be part of future work.

Although this method was appropriate for our previous small scale trace-based simulator, we have refined and modified the RC invocation to integrate it into our new Simics/GEMS-based RC simulation platform. To add RC-related instructions to the SPARC ISA, we overload an existing instruction currently interpreted as a `NOP`. This opcode is actually the `seti imm, %g0` instruction, but in the SPARC architecture register `%g0` is hardwired to zero, and any attempt to write to it is treated as a `NOP`. This instruction provides 22 bits for encoding the immediate operand, allowing us to overload it to create a number of different special instructions with space left for data used in those instructions.

```
  ......
  RCK.nop Rs, kid;     // Rs<-1 if kid on HW
  bz Rs, L_RC_sw_kid; // 0->SW, 1->HW
  RC_HW_func.nop;      // launch kernel HW
  ba  L_RC_end;
L_RC_sw_kid:
  RC_func();           // kernel's SW code
L_RC_end:
  ......
```

Fig. 3. Kernel invocation split into two separate instructions

To minimize instruction set modification, we implement the special branch using a sequence of two instructions as shown in Fig. 3. The first instruction is a special register set instruction (`RCK.nop`), that sets register R_s to 1 if a hardware implementation for the requested kernel is currently available on the RH, and 0 otherwise. Then, an existing branch-on-zero (`bz`) instruction can be used to branch to the software equivalent if the hardware version is not currently loaded and ready on the RH. After the hardware kernel execution completes, we branch to the end of the software implementation, rejoining the normal program flow.

This two-instruction sequence has two advantages. First, adding a new special "set register" instruction to the Simics/GEMS environments is significantly simpler than adding a new branch instruction. Second, if the application code ensures that R_s is set to zero at the beginning and is not modified apart from the `NOP`, modified application binaries will run correctly on existing (unmodified) SPARC machines. The simulator, on the other hand, captures the special `RCK.nop` when encountered, and uses Simics APIs to directly manipulate the R_s register. If the kernel is currently loaded on the simulated system's hardware, the simulator sets R_s to 1, and the program flow will fall through to the hardware support code at the following `bz` instruction. In the hardware branch, we add another special instruction: `RC_HW_func`, which is intended to actually trigger hardware kernel execution in a real RC system, and is therefore captured by our simulator for this purpose. For functional correctness, the kernel must somehow be actually computed at this point, as discussed in the next section.

4.2. Simulation Accuracy

Although binary compatibility and software execution issues are relatively easy to solve, augmenting Simics/GEMS to accurately simulate RC hardware execution is more difficult. In particular, we need to consider simulation speed, functional accuracy, and timing accuracy. Actual Verilog simulation of the hardware components is more complex than we currently support (or need) in the simulator, but is a possibility for future versions of the simulation environment. This approach would aid in functional and tim-

```
  RCK.nop Rs, kid;      // Rs<-1 if kid on HW
  bz Rs, L_RC_sw_kid;  // 0->SW, 1->HW
  RC_HW_func.nop;       // launch kernel HW
  RC_func();            // functionally correct
  ba  L_RC_end;
L_RC_sw_kid:
  RC_func();            // kernel's SW code
L_RC_end:
```

Fig. 4. The simulated CPU can execute kernel software for functional correctness, but this does not preserve application binary compatibility between real and simulated systems.

ing accuracy, but could negatively impact simulation speed. Another possibility could be to use actual RH as an integral part of the simulation platform, which would provide similar benefits to Verilog simulation, but without as negative an impact on simulation time. This option is discussed in more depth later in Future Work. Yet another approach would be to simply account for the time spent in hardware and not worry about actually simulating the functionality of the kernel, which would allow fast simulation, and timing accuracy (provided hardware timing is constant for all possible input data), but would not be functionally accurate.

At first, we considered ensuring functional accuracy by simply requiring the processor to always take the software branch, or to duplicate the kernel software routine inside the hardware part of the branch as shown in Fig. 4. The selfish benefit of this approach is that it would be the easiest to implement, as it requires no extra modification to Simics/GEMS. Simulation speed is the equivalent of simulating the software, though one might intuitively object to simulating the hardware operation on a simulated processor.

A significant problem of this approach is the difficulty of achieving timing accuracy. Presumably, the timing of the software version of a kernel will be greater than the hardware version, or it would not have been converted to hardware. We considered "freezing" simulation time while the simulated CPU executed the kernel software, then advancing simulation time based on the kernel's hardware execution time. Unfortunately, this is a complex issue in a full-system simulator capable of simulating multi-processor systems, as the CPUs may become "out of phase" with respect to each other and the rest of the system. Also, the machine state could change at different times in a simulated vs. real system. For example, exceptions or interrupts could occur while the simulated CPU was executing the kernel's software equivalent, which could change the program flow. Also, until runtime, we don't know exactly what the memory or cache state is, and therefore, what our memory latencies will be. If we execute the software version of the kernel without advancing simulation time, we could not accurately model memory access times (particularly in shared-memory multiprocessor environments). We could instead use the opposite approach of advancing simulation time while executing the software equivalent, then subtracting simulation time according to the difference in hardware and software execution times, but this would still be inaccurate if memory accesses were performed after the hardware would have completed, we would still have synchronization issues, and roll-back of simulation time (but not system state) is not supported in the Simics API. Another important drawback of using the simulated CPU to model hardware execution is that application binaries used in simulation would differ from those used on real RC systems. A real RC system would use actual hardware to implement hardware kernels, not a software-based simulation.

Therefore, we chose instead to use the *simulator itself* to natively execute software equivalents of the kernels. A software equivalent of each kernel is registered, along with its kernel ID, in advance with the simulator. The simulator then executes a pre-compiled version of the software equivalent to generate the actual kernel outputs when `RC_HW_func` is called. For situations where a software-only version is not appropriate (too slow, too much power) for an actual deployment of the simulated system, the simulator can be forced to always choose the hardware branch. Currently, this requires stalling until the configuration is loaded, but an upcoming version will allow the stalled thread to sleep and one or more other threads to execute in the meantime. However, a software equivalent must still be registered with the simulator for functional correctness, even if it is not part of the actual application software binary.

This approach also allows us to achieve timing accuracy, as we can advance simulation time based on the kernel's hardware execution time, as determined by synthesis for the target reconfigurable architecture. The simulator models the configuration time of the RH as either a constant value (serially-programmed single-context or serially-programmed multi-context design) or as a user-defined function of the size of the kernel configuration (partially-reconfigurable design). We will discuss the issue of how we preserve memory timing accuracy in section 4.3. Simulation time is actually improved compared to the previous method, since the simulator executes a pre-compiled version of the kernel native to the machine running the simulation. Finally, we also can use the same binary in the simulator as an actual RC system, with the same code as in Fig. 3.

The simulator must also update the simulated machine's state based on the executed kernel, just as it would be in a physical RC system. Because we are aiming for interchangeable hardware and software versions for application kernels, the memory values should be updated the same way whether the hardware or software version were used. The software code has instructions to update memory locations. For the hardware version, we need to match these memory updates. Initially one might think that the simulator could simply write these values to the appropriate memory and register locations using the standard Simics API (which allows users to directly access and modify physical memory locations). However, applications use virtual memory addresses, which could be mapped to different physical locations each time the applications are run. This problem is exacerbated by the fact that the kernel `RC_HW_func` call may need to operate on a very large memory space (such as for multimedia applications). In this case, some or all of the virtual memory locations may not yet be loaded into physical memory when the kernel is called, preventing a direct virtual→physical address translation.

However, a memory residence guarantee *can* be provided at the application level by modifying the application source. For example, applications can touch the required pages in advance, or lock a page in memory through system calls. Unfortunately, these techniques by themselves are insufficient given that we are performing a full system simulation of a multi-tasking environment. Pre-touching a memory page is not an actual guarantee if a processor context switch occurs between the pre-touch and the subsequent call to `RC_HW_func`. Locking a page in memory is possible provided the lock can occur atomically with the pre-touch, but repeated system calls like this can adversely affect system performance. The solution we have implemented is tied directly to the current processor/RC/memory interface model implemented in our simulator, and is therefore discussed in the next section. Other possible solutions are discussed in the Future Work section.

4.3. Modeling the CPU/RH/Memory Interface

In order to provide a standard interface between hardware kernels in the RH and the rest of the system, we currently use a local buffer within each kernel to hold the input and output data. Before the `RC_HW_func` call, the input data to the hardware must be loaded from the CPU's memory hierarchy into the kernel's local buffer (the "pre-load" phase). After the call, the output results can be read from the buffer to store into the CPU's memory hierarchy (the "post-store" phase). Another ISA extension would allow applications to directly read from and write to kernel buffers. The CPU is therefore responsible for all memory accesses, simplifying the problem of virtual addressing. The special loads and stores must be added to the application code within the hardware part of the kernel branch, as shown in Fig. 5. The use of the local buffer also simplifies the interface from a hardware designer's perspective, as they do not need to use a specialized interface, or have to worry about variable memory latency. Memory timing and bandwidth limitations are implicitly addressed by Simics and GEMS for these CPU-initiated memory accesses. Advanced memory interfaces discussed in the Future Work section that do not use the CPU as an intermediary will have to interface more explicitly with GEMS to maintain accuracy.

```
  RCK.nop Rs, kid;    // Rs<-1 if kid on HW
  bz Rs, L_RC_sw_kid; // 0->SW, 1->HW
  ld;                 // input pre-loads
  ......
  RC_HW_func.nop;     // launch kernel HW
  st;                 // output post-stores
  ......
  ba  L_RC_end;
L_RC_sw_kernel:
  RC_func();          // kernel's SW code
L_RC_end:
```

Fig. 5. Complete kernel invocation, including pre- and post-HW execution memory operations

This approach has a number of downsides. First, the CPU is acting as a memory controller when it could either be performing other tasks to increase system performance, or enter a sleep state to conserve power. Second, a local buffer essentially duplicates storage requirements for kernel inputs and outputs. Third, currently the kernel cannot begin execution until the complete input data is ready. The interface between RH and system memory is still a topic of open research, and in the Future Work section we discuss our future plans for improving kernel memory interfacing. For now, we use the described local buffer method.

Despite its drawbacks, the pre-load and post-store operations do provide the opportunity to address the memory's functional correctness problem described earlier in section 4.2. The reason a direct memory manipulation is not always possible is that the target virtual memory location might not yet be located in physical memory. However, the pre-load/post-store instructions ensure that the entire input/output space is accessed, guaranteeing that these memory locations reside in physical memory at least once in close temporal proximity to the related hardware kernel call. The pre-load phase models loading the input values into a kernel's local data buffer, while the post-store phase models loading the kernel outputs from the buffer into the CPU's memory. The CPU performs the required virtual→physical memory translations, and loads required memory pages.

The remaining problem is to actually put the correct kernel output values into the required memory locations. Remember that when choosing the hardware implementation of a kernel, for functional accuracy the kernel's software equivalent is executed within the simulator itself, not on the simulated platform. Therefore, the results of that execution are in the memory of the computer *running* the simulation, not the simulated system itself. Since the post-stores are executed on the simulated platform, but the actual results are not known within that scope, the application binary writes dummy values to all output memory locations. Inside the simulator, the Simics API allows us to intercept these store operations as they *complete* execution. Although the sim-

Table 1. Machine Configuration.

Processor	2GHz, 4-way OOO, 14 stages
ROB size	128 entries
L1 cache	split I-D, each 4-way 32K, 1 cycle latency
L2 cache	unified, 4-way 1M, 18 cycle latency
Memory	256M, 160 cycle latency
RH	Xilinx Virtex-4

ulator knows the virtual memory addresses for the result data, it is not aware of the corresponding physical address until the simulated CPU performs address translation during the dummy store. The CPU loads the memory page if required and performs address translation, generating the correct physical addresses for the result data. The simulator then replaces the dummy values at the stored locations with the actual values it had computed. This ensures that the correct results will be stored at the correct addresses with correct memory timing.

5. SIMULATION RESULTS

In this section, we demonstrate the use of our full system simulation platform to investigate a particular problem in RC research—performance results that can depend on the simulation approach. We model a RC system as previously described, including the local buffer to store kernel inputs and outputs. We compare performance numbers of three different possible ways to simulate this type of system:

1. Ignore the local buffer memory copy overhead, and only consider the RC HW execution time
2. Charge a fixed number of cycles for local buffer copy operations, which assumes an L1 cache hit on all data accesses
3. Fully simulate the memory overhead (which may include cache misses) using our pre-load/post-store augmented binary

The simulated machine has a 2GHz 4-way out-of-order SPARC-9 processor with 256MB memory and 8GB SCSI disk running Solaris 9, augmented with RH based on a Xilinx Virtex-4. Configuration details are shown in Table 1.

We implemented a 128-bit AES encryption application as a benchmark for this comparison. This particular application spends most of its time in a single routine `aes_encrypt` that encrypts one 16-byte block per function call. We implemented a hardware version of this kernel in a Xilinx Virtex 4 FPGA to obtain timing information. Since this is the only kernel implemented, we assume for all cases that the kernel fits in hardware, and is pre-loaded. We also assume that since we are only using one key, that the subkeys are precomputed. The hardware implementation is pipelined and encrypts *two* blocks for every hardware call. The subkeys are initially loaded in 11 cycles, and it takes 22 hardware cycles to execute the kernel with a cycle time of 4.35ns, for a total latency of 96ns per two blocks encrypted. This translates to 192 CPU cycles, at a 2GHz CPU clock rate. We assume a 64-bit bus width for the data connection between the CPU and RC HW, an easily achievable configuration. The buffer copy therefore requires 4 hardware cycles to get the input, and another 4 for writing the output.

Table 2. Simulated Performance Results

	Kernel Inst Count	Kernel Cycle Count
SW	56,950,784	26,879,489
HW Case 1	294,912	6,643,382
HW Case 2	294,912	9,005,565
HW Case 3	2,752,512	16,187,770

If we assume memory accesses are free, we do not add any overhead to the kernel timing. If we assume that all data accesses can be served in L1 cache, this will add 35ns memory latency overhead to the RC kernel execution. We simulated a short run of this AES benchmark with an input file of 1MB size, which requires approximately 30 million instructions. Table 2 shows the number of cycles spent in the `aes_encrypt` kernel for a software-only implementation, and the three different simulation techniques. As we can see, ignoring the local buffer copy overhead significantly underestimates the actual cycle count. While adding a fixed number of cycles to account for the buffer copy process (assuming that all data will be in L1 data cache when needed) improves timing estimate accuracy, the results still differ significantly from the full-system simulation results with the pre-load and post-store operations. This data indicates that semi-reasonable yet at least partially inaccurate assumptions and estimates used in evaluating a RC application or system may adversely affect the validity of the evaluation results. In the case of the extremely simple model, this lead to a 2.5x error in performance results.

6. FUTURE WORK

This simulation platform is still relatively new, and we have several plans for extending it and improving it. A natural next step will be to incorporate our previous work on RH scheduling [7] into the operating system we use in the simulation platform, using multi-processor support in Ruby [4] to model a CMP system augmented with RH.

In future simulator versions, users will be able to choose between several different memory interfaces (and corresponding processor interfaces), or even add their own with the help of the simulator's code base. The simulator will be able to model a variety of RC platforms, from a processor with a reconfigurable functional unit, to a system with a separate reconfigurable co-processor, simply by selecting the appropriate interfaces, and adjusting the parameters controlling simulated communication delays.

Currently, the simulator is limited to a primitive buffer-based interface between the processor and RH, and the main processor is the RH's memory controller. One of our next steps is to add a dedicated RH memory controller, allowing the CPU to attend to other tasks while the RH receives and stores information and performs its computation. Furthermore, we will implement a streaming memory model to allow computation to begin before all input data has been loaded into the buffer. We will add a special "stall" input to the kernel interface to allow the memory controller (or the buffer) to stall kernel hardware when the input data is not ready. However, more complex memory interfaces reintroduce issues that previously could be ignored, such as the fact that hardware implementations may access memory in a different order and at a different rate than software. In these cases, memory access pattern information must be provided to the simulator to ensure accuracy.

Another possible extension is to accelerate the simulation platform itself using an FPGA board. This board would *not* be used as a representation of the simulated RH, because of the difficulties discussed in section 2. Instead, the hardware would fill the same role as the simulator itself when executing the pre-compiled binary of the application kernel — a tool to provide functional correctness. Within the simulation, there would be no difference between our current technique and using the FPGA-augmented simulation platform (apart from simulation speed). Timing would still be based on user parameters, determined from the structure of the RH modeled in the simulator. The simulator will manage the real FPGA resources as it farms out native kernel implementations to it for execution, but will also maintain a view of the simulated RH resources. The simulated design may be a completely different design, with a different configuration and communication architecture, and with different kernels residing in simulated hardware than actual hardware. Essentially, the simulator would act as a virtual machine layer on top of the real FPGA resources.

7. CONCLUSION

A full-system reconfigurable computing simulation infrastructure offers researchers a powerful tool to investigate new reconfigurable hardware designs, RH/processor/memory interfaces, and systems software techniques. Our simulator provides a realistic environment for RC evaluation, able to evaluate performance at a level of detail and accuracy not previously possible using simple user-space code simulators. A simple comparison of common memory interface assumptions showed that the simplest (and least accurate) assumption resulted in performance estimate errors of approximately 2.5x, while the slightly more accurate (but still not a full-system simulation) approach was an improvement, but still off by 1.8x. A simulation platform also allows for a wider variety of designs to be evaluated than if researches had to take the time to implement each possible one in a real hardware system to obtain realistic performance evaluations. A full-system RC simulator therefore both facilitates research in existing and emerging areas of RC, and improves the accuracy of that research.

8. ACKNOWLEDGEMENTS

This work was supported in part by a grant from the Wisconsin Alumni Research Fund.

9. REFERENCES

[1] Peter S. Magnusson et al, "Simics: A Full System Simulation Platform", *IEEE Computer*, 35(2):50-58, February 2002.

[2] D. Burger, and T. M. Austin, "SimpleScalar Tutorial", presented at *30th International Symposium on Microarchitecture*, December, 1997.

[3] "The SPARC Architecture Manual. Version 9", SPARC International, Inc. San Jose.

[4] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset", *Computer Architecture News (CAN)*, September 2005.

[5] http://www.virtutech.com

[6] Carl J. Mauer, Mark D. Hill, and David A. Wood, "Full System Timing-First Simulation", *Proceedings of the 2002 ACM Sigmetrics Confernece on Measurement and Modeling of Computer Systems*, June 2002.

[7] Wenyin Fu, Katherine Compton, "An Execution Environment for Reconfigurable Computing", *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005.

[8] H. Quinn, L. S. King, M. Leeser, W. Meleis, "Runtime Assignment of Reconfigurable Hardware Components for Image Processing Pipelines", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[9] V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins, "Designing an Operating System for a Heterogeneous Reconfigurable SoC", *Reconfigurable Architecture Workshop*, 2003.

[10] R. Lysecky, F. Valid, "A configurable logic architecture for dynamic hardware/software partitioning", *Design, Automation, and Test in Europe Conference and Exhibition*, pp. 480-485, 2004.