Online SystemC Emulation Acceleration

Scott Sirowy, Chen Huang, and Frank Vahid *

Dept. of Computer Science and Engineering

University of California, Riverside

{ssirowy,chuang,vahid}@cs.ucr.edu

*Also with the Center for Embedded Computer Systems, University of California, Irvine

ABSTRACT

Field-programmable gate arrays (FPGAs) have recently been used as platforms to emulate SystemC descriptions. Emulation supports in-system testing using real input and output. We previously showed emulation speed to be competitive with SystemC simulations on a PC when the emulator uses acceleration engines. A limit on the number of acceleration engines that can fit on an emulation platform creates new online problems involving runtime decisions as to when to load a SystemC process into an acceleration engine. We define the online SystemC emulation acceleration problem. In contrast to previous works that focus on statically improving SystemC (and the more general event-driven) simulations, we utilize online heuristics to manage the use of a limited number of SystemC acceleration engines in an emulation framework, where the kernel must adapt and react to dynamically changing process and event queues. We test several online heuristics and show 9x improvement over microprocessor-only emulation and 5x over statically preloaded acceleration engines. We further improve emulation performance by 10-20% by adding kernel bypass connections between acceleration engines and by adapting the online heuristics to make use of those connections.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: – RTL, Optimization, Simulation

C.0 [**Computer Systems Organization**] – HW/SW Interfaces

General Terms

Algorithms, Design, Languages, Performance

Keywords

SystemC, Emulation, Simulation, Virtual Machines, Bytecode, Online Algorithms

1. INTRODUCTION

A SystemC description can be executed in various ways. One common way is to *simulate* the description on a PC. Simulation allows for testing of the description without expensive or unavailable physical hardware and without extensive time spent mapping to a physical platform. Drawbacks are that simulating a SystemC description might be slow or inaccurate, and that constructing input stimuli can be difficult and time-consuming while still not matching the complexity and nuances of real inputs and outputs (I/O). Another way is to *synthesize* a SystemC description to an ASIC, FPGA, or board-level customized implementation. A synthesized SystemC description benefits from interacting with physical I/O at high speed. However, SystemC synthesis tools can be expensive (compared to compilers), may only run on limited PC platforms and be challenging to install (especially on lower-end PCs), may be unpredictable with respect to circuit size/speed or tool runtime, often require long runtimes (such as hours or days), may not support particular target devices or platforms, and can only synthesize the parts of the code written for synthesis. A recent alternative to SystemC simulation or synthesis is SystemC in-system *emulation*, wherein SystemC *bytecode* executes on a processor interacting with real I/O [21]. The core of the SystemC emulation platform is a software-based *SystemC kernel* that consists of a SystemC bytecode virtual machine, an event-driven kernel, and access to on-board peripherals. Though slower than a synthesized implementation, emulation of an application (e.g., of an image processing system) enables early prototyping that benefits from real I/O rather than constructed I/O in simulation, as shown in Figure 1(a) and (b).

For the common situation where the emulation platform is implemented on (or with access to) an FPGA, FPGA-based acceleration engines can increase emulation speed, enabling SystemC execution speed comparable to simulation on middle-to-high-end PCs. Our SystemC *acceleration engine* has a MIPS-like datapath that executes the same bytecode that the SystemC kernel executes, but executes that bytecode orders of magnitude faster.

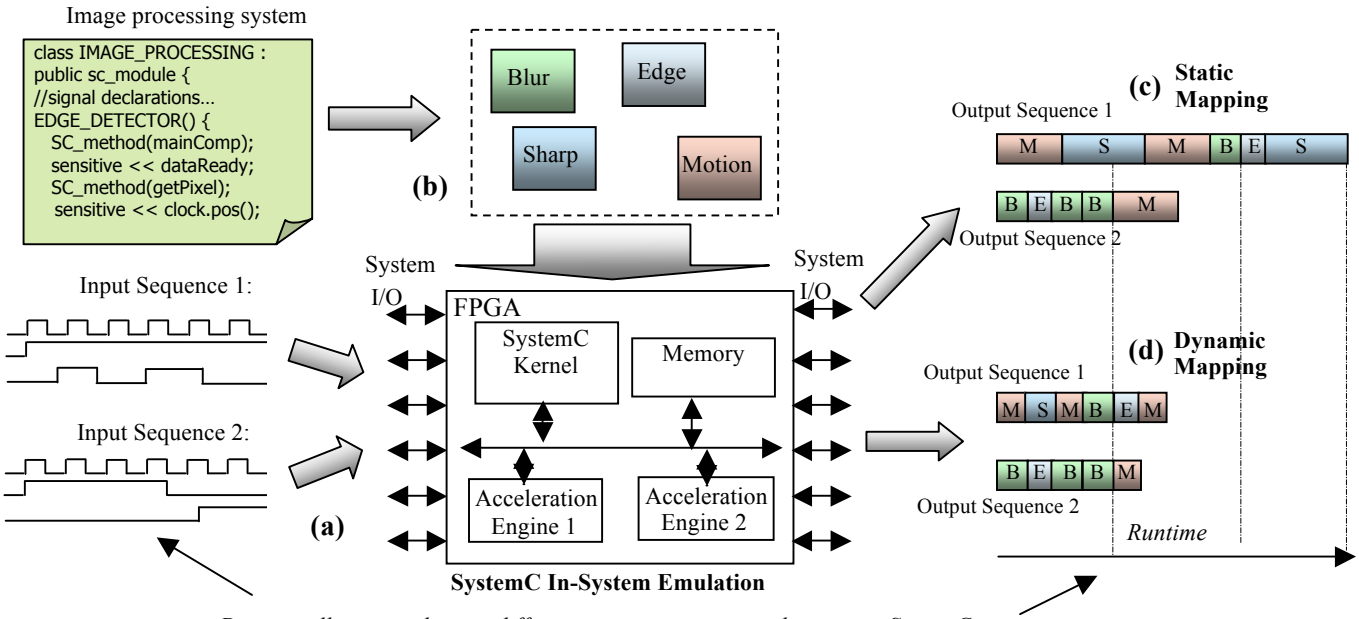
One drawback of SystemC in-system emulation is that the ordering of events on the event queue is not known before runtime, making some existing static acceleration techniques like queue reordering [15] and process splitting [17] less effective. Figure 1(c) and (d) show how two different input sequences into a SystemC emulation image processing system can generate two different output sequences, for which an adaptive mapping of processes to acceleration engines can gain faster emulation performance. Our SystemC emulation framework allows for dynamic decisions of whether to execute a process' bytecode on the microprocessor SystemC kernel or to load and execute that bytecode on an acceleration engine. However, acceleration engines are limited, and loading acceleration engines involves time overhead, so load decisions should be made so as to minimize total execution time.

Thus, a problem exists as to how to efficiently utilize the limited number of SystemC acceleration engines to execute a changing event-driven SystemC emulation event queue to minimize total emulation time. We define the *online SystemC emulation acceleration* problem, and apply online heuristics to dynamically improve the performance of SystemC emulation.

The paper is organized as follows. Section 0 discusses related work. Section 3 summarizes the SystemC emulation architecture and SystemC bytecode accelerators. Section 4 defines the online SystemC emulation acceleration problem. Section 5 describes several dynamic heuristics. Section 6 details several experiments, and Section 7 concludes.

DAC'10, June 13-18, 2010, Anaheim, California, USA

Figure 1: SystemC in-system emulation: **(a)** in-system emulation of a description allows testing with real I/O, **(b)** sample image processing system that invokes several different filters depending on the input, **(c)** statically mapping each process to either software or an acceleration engine results in widely-varied runtimes for different input sequences, **(d)** dynamically mapping SystemC processes in response to the input sequence results in faster emulation for all the input sequences.



Dynamically responding to different input sequences and mapping SystemC processes to available SystemC acceleration engines result in faster emulation.

2. RELATED WORK

Improving the performance of event-driven simulations has been extensively researched. Much research has concentrated on developing parallel frameworks for general event-driven simulation. Fujimoto [8] presents a comprehensive survey of several parallel simulation techniques. Jefferson [15] analyzes the critical paths of event-driven simulations, and discusses techniques to achieve supercritical speedups in simulation. Das [7] discusses adaptive protocols for parallel simulations.

Other work has focused specifically on improving SystemC simulations. Naguib [17] automatically splits SystemC processes to prevent unnecessary wake up calls to the SystemC event kernel. Perez [20] creates an optimized implementation of the SystemC kernel that utilizes acyclic scheduling. Wang [23] uses compiled simulation to eliminate unnecessary evaluations, and to improve simulation time. Our work focuses on dynamic SystemC emulation (rather than static SystemC simulation) whose behavior requires dynamic scheduling techniques to improve performance.

Another area of research combines both of the above approaches to parallelize the SystemC *simulation* kernel. Chopard [4] and Combes [5] show how relaxing a number of constraints on the event queue makes feasible a parallel SystemC event-driven kernel. Chandran [3] identifies methods to execute the SystemC kernel on simultaneous multiprocessor machines for faster performance. Our work utilizes FPGA resources to accelerate the execution of SystemC processes for faster emulation.

Dynamic load balancing has been studied extensively in previous works [11][13][16]. The idea of dynamic load balancing is that migrating processes across a network from high load hosts to lower load hosts can minimize application execution time despite overhead in migrating processes between processors. Our online SystemC emulation acceleration problem can be considered a special case of dynamic load balancing with heterogeneous processing units and high migration overheads.

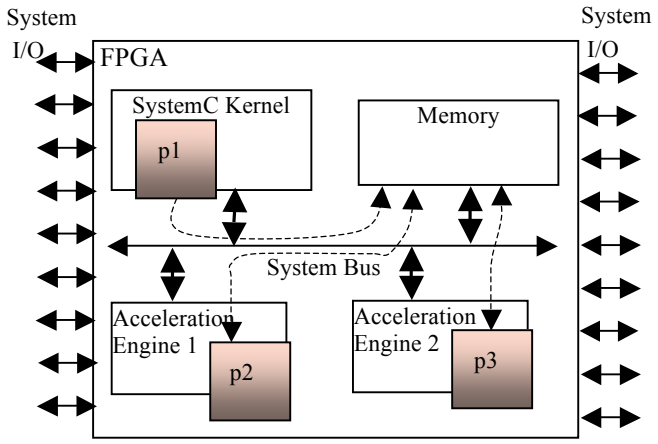
Dynamic system optimizations have also been the focus of much research. Balarin [2] presents a survey of real-time embedded system scheduling, which classifies the problem into static scheduling and dynamic scheduling. Danne [6] introduced real-time scheduling algorithms for periodic applications in an FPGA. Ghiasi [9] uses the task graph model to reorder task execution offline to minimize reconfiguration overhead. Huang and Vahid [12][13] developed new online heuristics for managing FPGA coprocessors in a dynamic environment. Noguera [18] proposed dynamic run-time hardware/software scheduling techniques for FPGAs emphasizing dynamic concurrent task scheduling. Steiger [22] proposed the use of a reconfigurable operating system to manage dynamically incoming tasks and the online scheduling problem. Our work applies some of these dynamic techniques to improve the performance of SystemC emulation.

3. SYSTEMC IN-SYSTEM EMULATION ARCHITECTURE

3.1 Base Architecture with Acceleration Engines

A SystemC emulation architecture enables the execution of SystemC descriptions on real platforms without the need to synthesize/map for the particular platform, by executing an intermediate form of SystemC called *SystemC bytecode* [21]. Figure 2 shows a basic SystemC emulation platform. The platform consists of a main processor that executes the SystemC kernel, which is a combination of a virtual machine and event-driven kernel. The SystemC kernel connects to the platform's peripherals (memories, lights, buttons, timers, general I/O) through a shared

Figure 2: SystemC emulation platform, with SystemC bytecode acceleration engines that speed up the SystemC kernel, communicating through a shared memory via a system bus.



bus, allowing a SystemC description full access to a variety of peripherals.

For the common situation where the emulation engine is implemented on (or with access to) an FPGA, the SystemC kernel can offload process emulation to a SystemC *acceleration engine.* An acceleration engine, shown in Figure 3(a), consists of a MIPS-like datapath, communicates with the SystemC kernel via memory-mapped registers, and executes SystemC bytecode faster than the SystemC kernel.

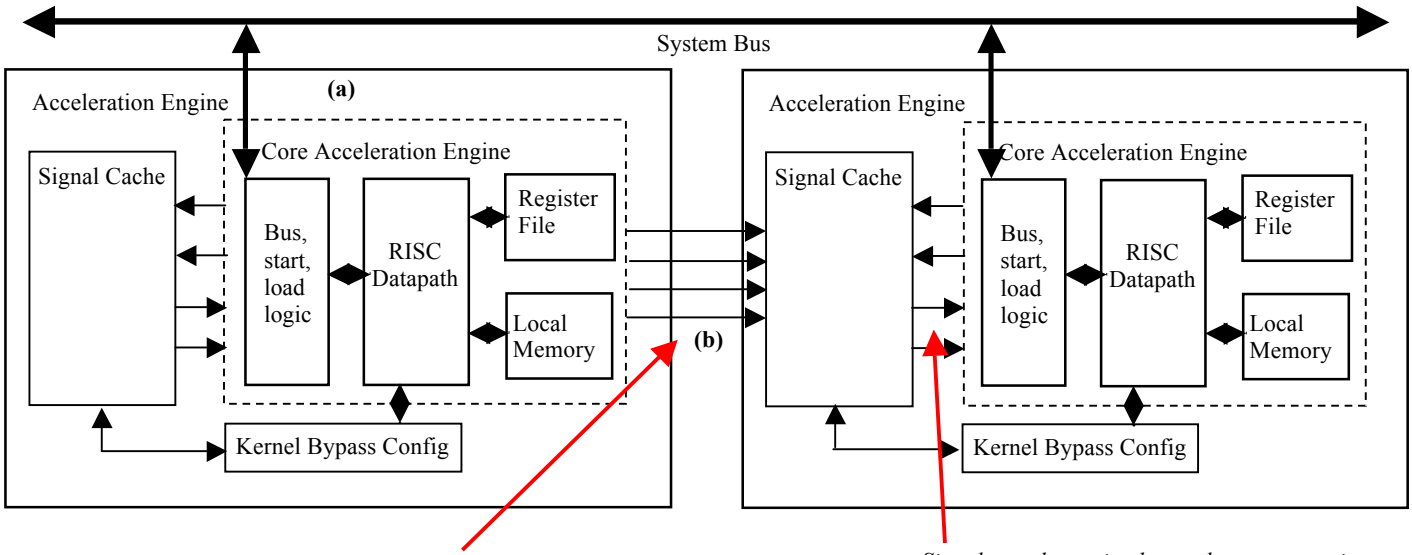
3.2 Kernel Bypass

We observed that the SystemC emulation platform possesses a memory bottleneck when both the main emulation kernel and the SystemC acceleration engines attempt to read and write the shared signal memories. To mitigate the memory bottleneck, we introduce *kernel bypass connections,* which are direct one-way connections between neighboring accelerators that allow the SystemC accelerators to communicate without having to read and write their values to shared memories via the system bus. Figure 3(b) shows the kernel bypass architecture for two SystemC accelerators. Another advantage of kernel bypass connections is that the emulation kernel also reduces some overhead of maintaining the event queue since the writing accelerator can directly flag the reading accelerator to start execution once the writing accelerator is done.

To facilitate direct communication between two neighboring accelerators, we add a SystemC kernel-controlled configuration register and small signal cache. A *signal cache* is a small memory data structure that holds a signal identifier, the signal's value, and a valid bit. If an accelerator is configured to be a kernel bypass reader, the acceleration engine will instead first look for a signal value in signal cache prior to fetching the value from the signal memory on the bus. Similarly, if a SystemC accelerator is configured as a kernel bypass writer, the SystemC accelerator will write to the connected accelerator's signal cache by sending the signal's ID and its current value. In contrast to the system bus that can take tens of cycles, the signal cache allows single-cycle signal writing and retrieval. For each simulated time step, a utilized kernel bypass connection can save between tens and hundreds of cycles, depending on the number of signals written to and read from.

The signal cache size is currently limited to ten signals. If two processes communicate with more than ten signals, the two processes must communicate through the bus-connected signal memories. Processes that communicate with more than ten signals can still see some speedup because ten read and writes to the system bus are eliminated every simulated time step.

Figure 3: SystemC acceleration engines: **(a)** internal structure, **(b)** direct connection of two SystemC acceleration engines using a kernel bypass connection. In some situations, bypassing the bus and SystemC kernel can lead to significant speedup for a given SystemC description.



The direct connections between the core acceleration engine and the adjacent signal cache allow the two acceleration engines to communicate without using the system bus or shared memory.

Signals to the main datapath to communicate with the signal cache rather than the system bus when configured properly

4. ONLINE ACCELERATION ASSIGNMENT

4.1 Problem Definition

We define the *Online SystemC emulation acceleration* problem as follows. Given are:

- A process set *P = {p1, p2, p3, ..pn}* containing the *n* processes that comprise a given SystemC description.
- A set of execution times *Tp = {tp1, tp2, tp3,... tpn}* containing the execution time of each process *i* running on the SystemC kernel without communication overhead.
- A set of execution times *Tc = {tc1, tc2, tc3...,tcn}* for each process *i* when running on a SystemC acceleration engine; the times do not include communication overhead.
- A set of sizes *S = {s1 ,s2, s3,..., sn}* giving the size of each process *i* in terms of number of bytecode instructions..
- The total number of acceleration engines *AE* in the SystemC emulation framework.
- The time to load one instruction into a SystemC acceleration engine *TR.* The total time to load an acceleration engine with process *i* can be thus be written as: *loading time(i) = TR*si.*

The online SystemC emulation acceleration problem must satisfy the following constraints:

- Processes running on the SystemC kernel and on the acceleration engines may run in parallel, unless that process is the same process *i.* For instance, in the queue *<p2, p1, p1, p1, p3>,* the three instances of *p1* must execute sequentially, but *p2* and the first instance of *p1* can run in parallel.
- The SystemC kernel cannot be interrupted to run a process when the SystemC kernel is loading a process onto an acceleration engine or when the SystemC kernel is itself running a process.

We define two additional constraints to the online SystemC emulation acceleration problem, which take advantage of the kernel bypass connections within the SystemC emulation framework:

- A set *O* of process pairs (*Oi, Oj*) that satisfy the condition that all inputs into *Oj* are outputs from *Oi*. These process pairs can be determined statically and sent to the SystemC kernel at download time. The process pairs are treated as big processes that take 2 acceleration engines in the simulator.
- The number of kernel bypass connections in the SystemC emulation platform.
- The number of signal connections between each process pair (*Oi, Oj*).

The dynamic input to the problem is an event queue *Q*, such as *<p2, p1, p4, p2, p1, p1....>,* that lists and orders the process instances that run on the platform for a given time step.

The Online SystemC Emulation Acceleration problem is defined as an online problem: For each process in the event queue, using only knowledge of *prior* and *current* processes in the queue, determine whether to load that process into a SystemC acceleration engine, such that the time for the *entire* event queue (including future instances of the process in the queue) is minimized. When a process is already loaded into a SystemC acceleration engine, we refer to the process as being *acceleration-engine resident*. The *current process* is the process that at a given time is to be executed next and for which the acceleration engine load determination must be made. Thus, the solution to the online SystemC emulation acceleration problem consists of an acceleration engine management decision for each process instance in the event queue. Each decision is either: load, don't load, or already loaded. For a decision to load, the decision also lists a process that must be unloaded to make room for the new process being loaded.

4.2 Communication Overhead

The SystemC accelerators communicate with the SystemC kernel through memory-mapped registers and *signal memories,* which store the current and next values of each signal in the SystemC description. Our emulation architecture may have multiple acceleration engines running in parallel. The acceleration engines read/write to the system bus randomly which cause bus contention. We use queuing theory [9] to estimate average memory access delay, and model memory contention by the M/M/1 queue. The processes in the SystemC kernel and in the SystemC acceleration engines generate memory access requests through *READ* and *WRITE* bytecode instructions. We define the following:

- Random memory access rate: The random memory access rate is the number of times a process *i* reads from memory, where λi is the memory access rate of running process *i*.
- Bus service rate: μ. The bus service rate is the number of requests the system bus can process in a second. E.g. Assuming a 100Mhz memory bus, one access takes 20 cycles, so μ=5M/s.
- Average delay: The average delay is the number of cycles for one memory access. According to queuing theory, average delay for one access is $D=\lambda/(\mu(\mu-\lambda))$.
- System delay: $delay = D\lambda$.

5. HEURISTICS

5.1 Upper and Lower Bounds

An upper bound on total execution time can be determined by running every process on the SystemC kernel. A lower bound can be determined by assuming every process is preloaded onto an infinite set of existing SystemC acceleration engines, while considering communication overhead, referred to as the *Infinite Accelerators* bound.

5.2 Accelerator Static Assignment

To see the advantage of dynamically loading bytecode to the SystemC acceleration engines for higher performance emulation, we compare to a *statically preloaded* approach, which assumes each SystemC acceleration engine is initially loaded with one process' bytecode each, and is not reloaded during runtime. At the beginning of SystemC emulation, the SystemC kernel assigns each acceleration engine a process to always execute when an instance arrives on the event queue. The acceleration engines are loaded with the processes that have the largest speedup potential (*tpi-tci)*. Compared to dynamic techniques, the benefits of static accelerator assignment are one-time acceleration engine loading, and a simpler emulation event kernel. The drawbacks are that there might only be a few acceleration engines, and running the rest of the SystemC processes on the software SystemC kernel could be computationally expensive. An alternative method for static assignment would have been to utilize profile information to predict which processes execute most frequently. However, due to simulation complexity, profiling information may not be available.

5.3 Greedy Heuristic

A greedy heuristic can be defined that always loads the current process into a SystemC acceleration engine before executing. If the

process is already *acceleration-engine resident*, the SystemC kernel just instructs the SystemC acceleration engine to begin executing. Otherwise, the SystemC kernel randomly chooses an idle SystemC acceleration engine into which to load the process' bytecode instructions. In case all SystemC acceleration engines are busy running, the emulation kernel waits until one of the acceleration engines becomes idle. The time complexity of the greedy heuristic is *O(1)*. However, the greedy heuristic may incur much loading overhead since it loads a SystemC acceleration engine with bytecode on every execution. Furthermore, the greedy heuristic attempts to use all available acceleration engines, which increases communication overhead on the system bus.

5.4 Aggregate Gain

We use the aggregate gain (AG) heuristic introduced in [13] to address the online SystemC emulation acceleration problem. The AG heuristic uses the history of application executions to attempt to predict future executions and hence to predict when reconfiguration overhead is worthwhile. The AG heuristic considers reconfiguration and communication overhead. The basic idea of AG is that the heuristic maintains an aggregate gain table for each process type running in the system. The gain is the time saved by running the process instance with the accelerator. The AG table gets updated when a new process arrives. The AG table shows which processes would have gained the most speedup by running in a SystemC acceleration engine.

Sequences of processes on the event queue often exhibit temporal locality—recently-executed processes are more likely to execute in the near future than are processes from long ago. A fading factor *f* is thus introduced to refresh the AG table. The fading *f* is adaptive to the average loading time. The intuition of the loading, replacement, and wait decision is to make the total gain of the acceleration engine resident processes high. Thus, the load, replace, and wait decisions will be made only if the decision would not decrease the total gain of resident processes.

We can alter the AG heuristic to support the additional kernel bypass feature. The modified AG heuristic treats tightly-coupled processes as one large process. The large process requires multiple acceleration engines and the heuristic assumes the acceleration engines of the large process must be loaded together. The load, replacement, and wait policies of the large process are similar to the definitions in original AG heuristic.

6. EXPERIMENTS

6.1 Framework

We developed a simulator in C++ to test our heuristics, and applied the simulator to several SystemC descriptions. We also fully implemented two SystemC emulation platforms, one on a Xilinx Virtex4 Ml403 development platform, and one on a Xilinx Virtex5 vlx110t development platform. The SystemC kernels ran on a PowerPC and Microblaze processor respectively, both operating at 100 MHz. The SystemC kernels communicate to the acceleration engines and the rest of the peripherals through Xilinx's PLB (processor local bus). The average memory access time is 40 cycles. The SystemC kernel uses a handshaking protocol over the PLB to communicate and load instructions into each of the acceleration engines. The total time to load one instruction (*TR)* onto an acceleration engine is approximately three microseconds. The Virtex4 Ml403 development platform could hold one acceleration engine, and the Virtex5 vlx110t development platform could hold three. For two of the accelerators in the Virtex5 vlx110t, we connected them for kernel-bypassed enabled execution. One accelerator was configured as a reader, and one was configured as a writer. We chose this configuration because reader/writer configuration is very common for streaming applications such as image processing, encryption, etc. Since streaming applications are widely used, we first studied a reader/writer configuration. We may consider other configurations in future works. The kernel bypass circuitry required 5% more of the Virtex5 vlx110t FPGA logic than the core acceleration engine. The SystemC emulation kernel was written in approximately 2500 lines of C code. The online heuristics consisted of only a few hundred lines of code.

We applied our heuristics to an image filtering system (including a blur filter, an emboss filter, a sharpen filter, and several implementations of edge detection), a digital lung model [19], and a reconfigurable radiosity design [1]. We wrote the image filters, lung model, and reconfigurable radiosity designs in SystemC, capturing each design using multiple processes. We modeled several dynamic scenarios in which the image filters, lung model, and radiosity design might be used.

For all experiments, because sequences involve some random ordering, we generated 20 sequences, and report the arithmetic average. The heuristics' runtimes themselves were negligible.

6.2 Evaluation

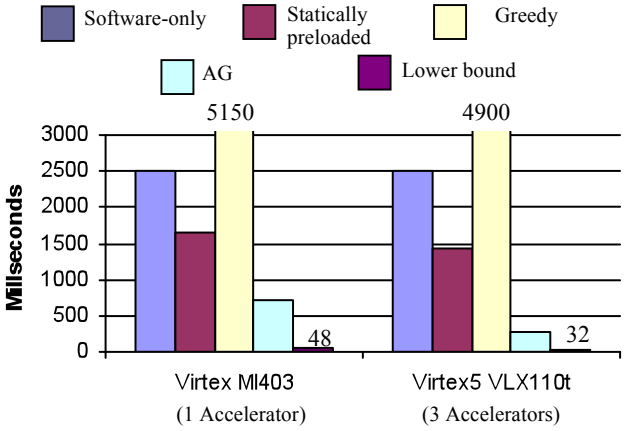
Figure 4 shows total execution times of a suite of SystemC image processing, lung, and radiosity descriptions running on Virtex4 Ml403 and Virtex5 vlx110t implementations of the SystemC emulation framework without the kernel bypass mechanism enabled.

For the Virtex4 Ml403 implementation, the *statically preloaded* accelerator approach yielded about 1.5x speedup compared to software-only emulation (i.e., only running on the SystemC kernel with no acceleration engines). The greedy heuristic results in a slowdown of 50% compared to software-only emulation. This is because the heuristic reconfigures the accelerators without consideration of the high reconfiguration cost of downloading new bytecode instructions. The dynamic *AG* approach yields more speedup over software-only emulation and a statically preloaded approach: 3.5x and 2.3x respectively.

For the Virtex5 vlx110t implementation, the *statically preloaded* accelerator approach yielded about 1.75x speedup compared to software-only emulation. Compared to the Virtex4 Ml403 implementation that only had one accelerator, the nominal speedup achieved with the Virtex5's three accelerators was unexpected, and was resulted due to the execution time of processes running without an acceleration engine. The penalty also has been due to increased communication overheads on the system bus. The *greedy* heuristic was again about 50% slower than software-only emulation because of the high cost to reload the acceleration engines with new bytecode instructions. The *AG* heuristic performed 9x and 5x faster than *software-only emulation* and *statically preloaded* solutions, respectively. The *AG* heuristic takes the accelerator reloading cost into account and thus did not always reload the accelerators every time there was a new process on the event queue.

Comparing with the *Infinite Accelerators* lower bound (i.e., all processes are accelerated without any loading overhead) shows that the AG heuristic obtains execution times on average within 15x slower on a platform with one accelerator because of the high loading time, and 8x slower on a platform with three accelerators of this lower bound. The lower bound solution does not need to contend with the high reconfiguration time the other heuristics do.

Figure 4: Emulation runtime results of image filtering, lung, and radiosity examples emulated on two different emulation platforms. *AG* performs up to 9x faster than software-only emulation, and 5x faster than a *statically preloaded* approach.



Future work could look into modifying the architecture for decreased reconfiguration times.

Figure 5 shows the effect of enabling a kernel bypass connection between two accelerators on the Virtex5 vlx110t emulation platform (the Virtex4 MI403 could only hold one acceleration engine, so kernel bypass was not applicable). On average, the SystemC examples improved their speedup by 11%. *Blur* and *Sobel2* achieved 20% speedup with kernel bypass because they contained a few processes that had much communication. Other examples like *Lung* and *Radiosity* only improved by a few percent because of communication between processes. More kernel bypass connections could increase performance further.

7. CONCLUSIONS

SystemC emulation platforms benefit from adapting accelerator usage to a dynamic event queue. We defined the Online SystemC Emulation Acceleration problem and applied several online heuristics to improve emulation performance by 9x over emulating all of the SystemC on the SystemC emulation kernel, and 5x over statically preloading the acceleration engines. Online heuristics could further speedup emulation by up to 20% using kernel bypass.

8. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation (CNS-0614957 and DUE-0836905).

9. REFERENCES

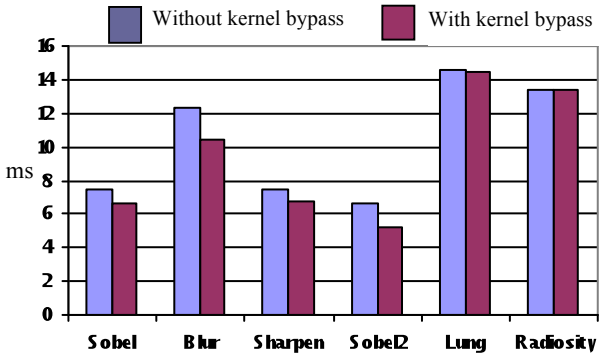
[1] Baker, P., Todman, T., Styles, H., and Luk, W. Reconfigurable Designs for Radiosity. FCCM 2005.

[2] Balarin, F. , Lavagno, L., and Murthy P. Scheduling for Embedded Real-Time Systems. IEEE Design and Test of Computers, 1998.

[3] Chandran, P., Chandra, J., Simon, B. P., and Ravi, D. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines. Workshop on Parallel and Distributed Simulation, 2009.

[4] Chopard, B., Combes, P., and Zory, J. A Conservative Approach to SystemC Parallelization. Lecture Notes in Computer Science. Volume 3994. 2006.

Figure 5: Emulation runtimes without and with kernel bypass using the AG heuristic on the image processing examples. Kernel-bypass-enabled emulations performed on average 11% better than without kernel bypass, and up to 20% in some examples.



[5] Combes, P., Caron, E., Desprez, F., Chopard, B., and Zory, J. Relaxing Synchronization in a Parallel SystemC Kernel. IEEE international Symposium on Parallel and Distributed Processing with Applications, 2008.

[6] Danne, K., Platzner, M. Periodic Real-Time Scheduling for FPGA Computers. Intelligent Solutions in Embedded Systems, 2005

[7] Das, S. R. Adaptive protocols for parallel discrete event simulation. In 28th Conference on Winter Simulation, 1996.

[8] Fujimoto, R. M. 1989. Parallel discrete event simulation. WSC 1989.

[9] Ghiasi, S. and Sarrafzadeh, M. Optimal reconfiguration sequence management. ASP-DAC 2003.

[10] Gross, D., and Harris, C.M. Fundamentals of queueing theory. John Wiley & Sons, Inc. New York, NY, USA. 1985.

[11] Harchol-Balter, M. and Downey, A. B. 1997. Exploiting process lifetime distributions for dynamic load balancing. ACM Trans. Comput. Syst. 15, 3 (Aug. 1997), 253-285

[12] Huang, C., and Vahid, F. Dynamic Coprocessor Management for FPGA-Enhanced Compute Platforms. CASES 2008.

[13] Huang, C., and Vahid, F. Transmuting coprocessors: dynamic loading of FPGA coprocessors. DAC 2009.

[14] Ishfaq Ahmad , Arif Ghafoor , Kishan Mehrotra, Performance prediction of distributed load balancing on multicomputer systems, ACM/IEEE conference on Supercomputing, 1991.

[15] Jefferson, D. and Reiher, P. Supercritical speedup. Annual Simulation Symposium. 1991.

[16] Min-You Wu. On runtime parallel scheduling for processor load balancing, IEEE TPDS 1997.

[17] Naguib, Y. N. and Guindi, R. S. Speeding Up SystemC Simulation through Process Splitting. DATE 2007.

[18] Noguera, J., Badia, R.M. Dynamic run-time HW/SW scheduling techniques for reconfigurable architectures. CODES-ISSS 2002.

[19] NSR Physiome Project. http://nsr.bioeng.washington.edu/.

[20] Pérez, D. G., Mouchard, G., and Temam, O. A New Optimized Implemention of the SystemC Engine Using Acyclic Scheduling. DATE 2004.

[21] Sirowy, S., Miller, B. and Vahid, F. Portable SystemC-on-a-Chip. CODES/ISSS 2009.

[22] Steiger, C., Walder, H., Platzner, M., AND THIELE, L. 2003. Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices. RTSS 2003.

[23] Wang, Z. and Maurer, P. M. LECSIM: a Levelized event driven compiled logic simulation.). DAC 1990.