Dynamic Acceleration Management for SystemC Emulation

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 gates arrays (FPGAs) have recently been used to emulate SystemC descriptions. Emulation of SystemC descriptions allows for in-system testing, and has been shown to compare favorably with SystemC simulations on a PC when *acceleration engines* are employed. A limit on the number of acceleration engines that can fit on a SystemC emulation platform creates new dynamic management problems involving decisions as to when and which acceleration engines to load with *SystemC bytecode*. We define an acceleration management problem for SystemC emulation platforms. In contrast to previous works that focus on statically improving SystemC (and the more general event-driven) *simulations*, we utilize dynamic online algorithms to manage the use of a limited number of SystemC acceleration engines in an *emulation* framework, where the kernel must adapt and react to a dynamically changing event queue. We test several online heuristics, and show that we can achieve 14X improvement over software-only emulation and 3.8X over statically preloading SystemC acceleration engines.

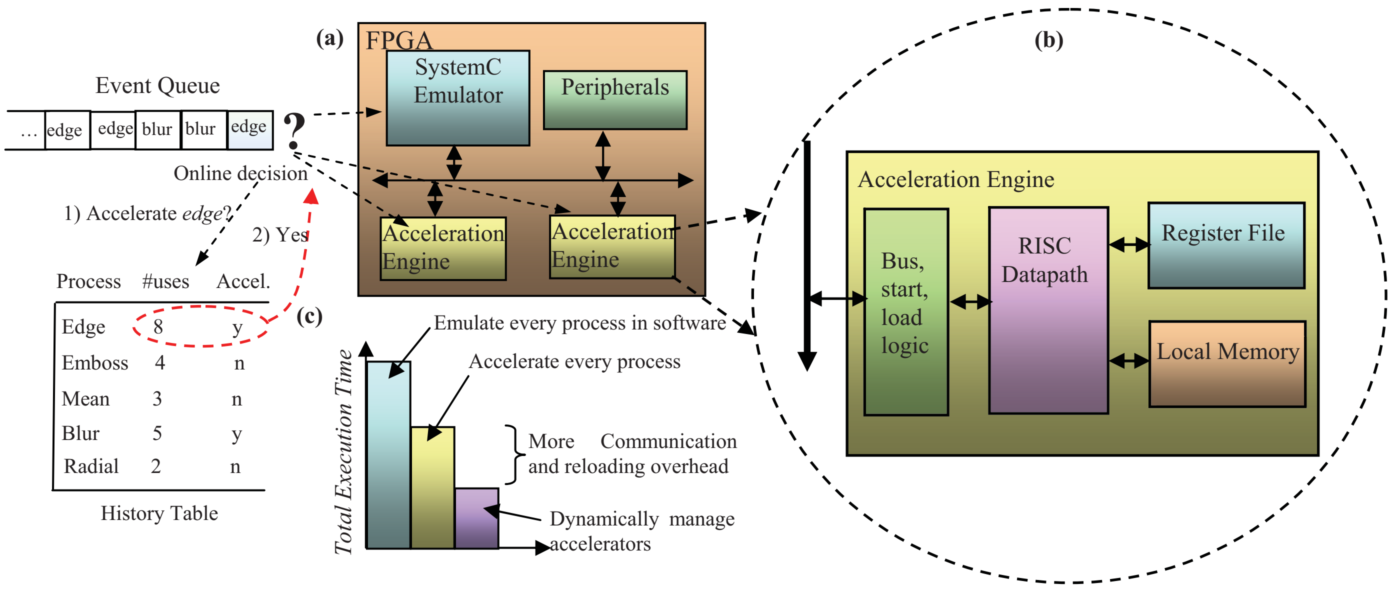
1. INTRODUCTION

SystemC descriptions can be executed in one of several ways. One common way is to *simulate* SystemC descriptions on a PC. Simulation allows for testing SystemC descriptions without costly or unattainable physical equipment. A few key drawbacks are that simulating SystemC models might be slow and inaccurate, and creating fabricated I/O can be difficult and time-consuming, while still not matching the complexity and nuances of real I/O. Some SystemC descriptions can even be *synthesized* to an ASIC, FPGA, or board-level customized implementation. Synthesized SystemC descriptions benefit from interacting with physical inputs and outputs with very high performance. 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. An alternative to SystemC simulation and SystemC synthesis is in-system SystemC *emulation*. Though slower than a custom implementation, emulation enables early prototyping, and benefits from real I/O rather than fabricated I/O in simulation.

For the common situation where the emulation engine is implemented on (or with access to) an FPGA, FPGA-based acceleration engines can substantially increase the emulation speed, enabling SystemC execution speeds comparable to middle-to-high-end PCs. A SystemC *acceleration engine,* shown in Figure 1(b), consists of a MIPs-like data path that executes the same intermediate form of SystemC called *SystemC bytecode* [19] the base emulator executes, albeit orders of magnitude faster.

One potential drawback of in-system SystemC emulation is that the ordering of events on the event queue is not known at runtime, making some existing static acceleration techniques like queue reordering and process splitting less effective. The SystemC emulation framework allows for a dynamic decision to

Figure 1: (a) Emulating an image processing filtering system. (b) An *Acceleration Engine* can execute SystemC bytecode almost two orders of magnitude faster than the base SystemC emulator. (c) Dynamically adapting which processes get accelerated in a SystemC emulation results in better execution times than pure software emulation and accelerating every process because of communication and acceleration loading costs.



occur as to whether to execute the SystemC bytecode on the software emulator, or load and execute that bytecode onto an acceleration engine. But, acceleration engines are limited, and loading acceleration engines involves time overhead, so load decisions should minimize total execution time, as illustrated in Figure 1(c).

Thus, a new problem exists as to how to efficiently utilize the finite number of SystemC acceleration engines to service a dynamically changing, event-driven SystemC emulation event queue such that the emulation time is minimized. We define the *SystemC Acceleration Engine Management* problem, and apply online heuristics to dynamically improve the performance of SystemC emulation.

Section 2 goes over related work. Section 3 defines the problem. Section 4 describes several dynamic heuristics. Section 5 details several experiments, and Section 6 concludes.

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 [6] 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 [5] discusses adaptive protocols for parallel simulations.

Other work has focused on specifically improving SystemC simulations. Naguib [16] automatically splits SystemC processes to prevent unnecessary wake up calls to the SystemC event kernel. Perez [18] creates an optimized implementation of the SystemC kernel that utilizes acyclic scheduling. Wang [20] uses compiled simulation to eliminate unnecessary evaluations, and to improve simulation time. Our work focuses on dynamic SystemC emulations (compared to static SystemC simulations), whose behavior require dynamic scheduling techniques to better improve performance.

Dynamic system optimizations have also been the focus of much research. Balarin [1] presents a survey of real-time embedded system scheduling, which classifies the problem into static scheduling and dynamic scheduling. Huang and Vahid [12][13] develop new online algorithms for managing FPGA coprocessors in a dynamic environment. Noguera [17] proposed dynamic run-time hardware/software scheduling techniques for FPGAs emphasizing dynamic concurrent task scheduling. Our work applies these dynamic techniques improve the performance of SystemC emulation.

3. PROBLEM DEFINITION

3.1 Communication Overhead

The SystemC accelerators communicate with the base emulator through memory mapped registers and signal memories which store the current and next values of each signal in the SystemC description. We use queuing theory to estimate average memory access delay, and model memory contention by the M/M/1 queue. The processes in the base emulator 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: # of cycles for one memory access. According to queuing theory, average delay for one access is $D=\lambda/(\mu(\mu-\lambda))$. System delay: $delay = D\lambda$.

3.2 Problem Definition

We define the SystemC Acceleration Engine Management 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 base emulator w/o communication overhead.
- A set of execution times $Tc=\{tc1, tc2, tc3,...,tcn\}$ for each process *i* when running on a SystemC acceleration engine without 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 the following: *loading time(i)=TR*si*

The SystemC Acceleration Engine Management problem must satisfy the following constraints:

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

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 SystemC Acceleration Engine Management (AEM) problem for time 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 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 AEM 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. HEURISTICS

4.1 Upper and Lower Bounds

An upper bound on total execution time can be determined by running every process on the base emulator. A lower bound can be determined by assuming every process is preloaded onto an infinite set of existing SystemC acceleration engines, and ignoring all communication overhead, referred to as *Infinite Accelerators/No Comm*. Another interesting comparison is running process on an acceleration engine, assuming infinite acceleration engines, but in this case considering communication overhead, referred to as the *Infinite Accelerators*. *Infinite Accelerators* gives a tighter bound

4.2 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 the SystemC acceleration engines are initially loaded with one process's bytecode each, and are not reloaded during runtime. At the beginning of SystemC emulation, the emulator assigns each acceleration engine a process to always execute when and instance arrives on the event queue. The acceleration engines are loaded with the processes that have the largest speedup potential (*Tp-Tc)*. 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 base software emulator could be computationally expensive.

4.3 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 *acceleration engine resident*, the emulation kernel just instructs the SystemC acceleration engine to begin executing. Otherwise, the emulation kernel randomly chooses an idle SystemC acceleration engine to load the process's bytecode instructions. In the case that all the SystemC acceleration engines are busy running, the emulation kernel will wait until the one of the acceleration engines becomes idle. The time complexity of the greedy heuristic is *O(1)*. However, the greedy heuristic may incur lots of loading overhead since it loads a SystemC acceleration engine with bytecode on every execution. Further, the greedy algorithm attempts to use all the available acceleration engines, which increases the amount of communicate overhead on the system bus.

4.4 Aggregate Gain Algorithm

We use the Aggregate Gain (AG) algorithm introduced in [13] to address the SystemC AEM problem. The AG algorithm uses the history of application executions to attempt to predict future executions and hence to predict when reconfiguration overhead is worthwhile. The AG algorithm considers the reconfiguration and communication overhead. The basic idea of AG is that we maintain an aggregate gain table for each process type running in the system. We define the gain as the time saved by running the process instance with the accelerator. The AG table gets updated when new process arrives. The AG table shows which processes make most of the gains by running in the SystemC acceleration engine.

The process instance sequences 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 introduced to refresh the AG table. *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 resident processes.

5. EXPERIMENTS

5.1 Framework

We developed a simulator in C++ to test our heuristics, and applied the simulator to several SystemC descriptions. We implemented two SystemC emulation platforms, one on a Xilinx Virtex4 Ml403 development board, and one on a Xilinx Virtex2Pro development board. We implemented both of the base software emulators on the PowerPC processors running at 100MHz. The base emulators communicate to the acceleration engines and the rest of the peripherals through the PLB bus. The base emulator uses a handshaking protocol over the PLB bus 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 Virtex2Pro development platform could hold three. The base software emulator was written in approximately 2000 lines of C code.

We applied our algorithms to an image filtering system which included a blur filter, an emboss filter, a motion filter, and several implementations of edge detection. We wrote the filters in SystemC and each filter was captured using multiple processes. We modeled several dynamic scenarios in which the image filters be used. We describe one scenario as *Random*, in which image filters are placed on the event queue randomly. Another scenario is the *Biased* case, in which a small number of filters appear on the event queue most of the time. The last is a *Periodic* scenario*,* in which a random subsequence of the filters repeats indefinitely on the event queue.

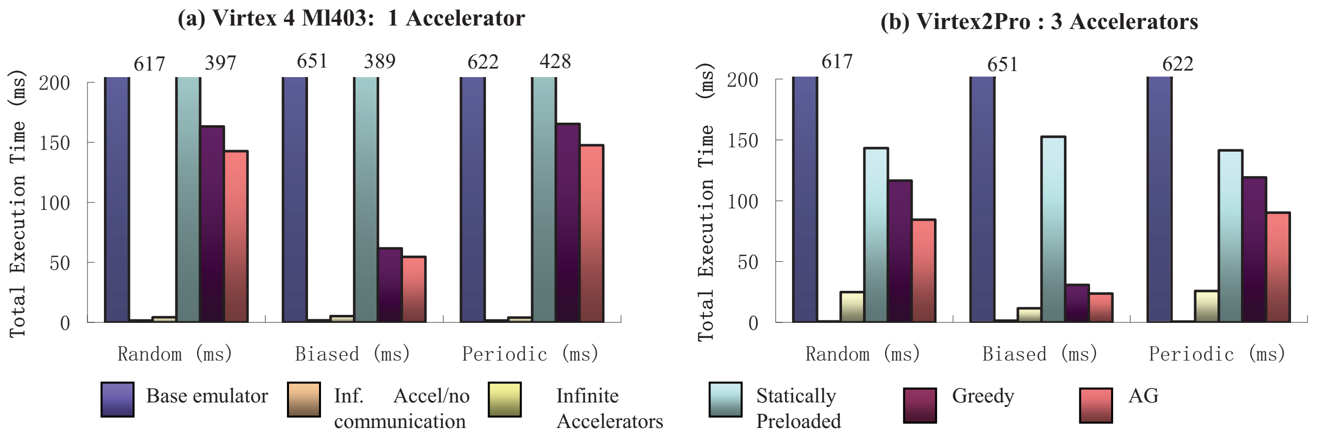
Each sequence's length was 500. For all experiments, because sequences involve some random ordering, we generated 20 sequences, and report the arithmetic average. For this work, execution time data does not include the time to run the heuristics themselves. The heuristic runtimes were negligible, adding only microseconds to each process execution.

5.2 Evaluation

Figure 2(a) shows total execution times of a suite of SystemC image processing descriptions running on a Virtex4 Ml403 implementation of the SystemC emulation framework. The statically preloaded accelerator approach yielded ~1.6X speedup compared to software-only emulation. The dynamic approach (*greedy or AG*) yield more speedup. The execution time obtained by *AG* is 4.3X and 1.2X faster than *statically preloaded* and *greedy solutions*, respectively. AG yields 7X speedup versus a software-only emulation.

Figure 2(b) shows similar results for the same image processing suite running on a Virtex2Pro SystemC emulation framework. The *statically preloaded* accelerator approach yielded 4.3X speedup compared to software-only emulation. The *statically preloaded* speedup compared to the Virtex4 implementation is slightly under the 3X improvement we expected to see from increasing the number of accelerators from one to three. The penalty comes from increased communication costs and reloading costs on the system bus. The execution time

Figure 2: SystemC Acceleration Management Experiments. (a) Image Filtering System running on a base emulator on a Virtex4 Ml403 that can fit one acceleration engine. (b) Same image filtering system running on a base emulator on a Virtex2Pro that can fit three acceleration engines. With three accelerators, emulation runs 14X faster than software-only emulation, and in both cases, the AG algorithm performed 1.2X better than a greedy approach and 3.8X better than statically preloading the accelerators.



obtained by *AG* is 3.2X and 1.3X faster than *statically preloaded* and *greedy* solutions, respectively. *AG* yields 14X speedup versus a microprocessor only solution. The Virtex2Pro emulation framework yielded on average 2X speedup compared to the Virtex4 Ml403 implementation. The greedy algorithms suffered on both platforms because of the high cost to reload the acceleration engines with new bytecode instructions. The *AG* algorithm takes the accelerator reloading cost into account and thus decided not to 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 and without the need to reload the bytecode instructions onto the accelerator) shows that the AG algorithm obtains execution times on average within 33X slower on a platform with one accelerator because of the high loading time, and 3X slower on a platform with three accelerators of this lower bound. The *Infinite Accelerators* suffers from much communication overhead, so *AG* shows less relative slowdown.

Comparing the different application scenarios, both *Greedy* and AG perform better in Biased scenario. Because a small number of applications appear most of the time, the number of reconfiguration is less in Biased scenario compare to random and periodic scenario, result in less total execution time.

6. CONCLUSIONS

SystemC emulation platforms benefits from adapting to a dynamic event queue. We defined the SystemC Acceleration Engine Management problem and applied a several online heuristics to improve SystemC emulation performance by 14X over emulating all of the SystemC on a base software emulation kernel, and 3.8X over statically preloading the acceleration engines. To our knowledge, this is the first work to use dynamic techniques to manage acceleration and improve SystemC emulation.

7. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation (CNS-0614957) and the Office of Naval Research (N00014-07-C-0311).

REFERENCES

[1] BALARIN, F. , LAVAGNO, L., AND MURTHY P. Scheduling for Embedded Real-Time Systems. IEEE Design and Test of Computers, 1998.

[2] BARTAL, Y., BLUM, A., BURCH, C., AND TOMKINS, A. A polylog(n)-competitive algorithm for metrical task systems. ACM Symp. on Theory of Computing, 1997, pp. 711-719.

[3] BENITEZ, D. Performance of remote FPGA-based coprocessors for image-processing applications. Digital System Design, 2002.

[4] BORODIN, A., LINIAL, N., AND SAKS, M.E. An optimal on-line algorithm for metrical task system. Journal of the ACM (JACM), Volume 39, Issue 4 (Oct. 1992), pp. 745 – 763.

[5] DAS, S. R. 1996. Adaptive protocols for parallel discrete event simulation. In *Proceedings of the 28th Conference on Winter Simulation*

[6] FUJIMOTO, R. M. 1989. Parallel discrete event simulation. In *Proceedings of the 21st Conference on Winter Simulation* E. A. MacNair, K. J. Musselman, and P. Heidelberger, Eds. WSC '89.

[7] FUJIWARA, H., AND IWAMA K.. Average-Case Competitive Analyses for Ski-Rental Problems. ISAAC 2002.

[8] GROSS, D., AND HARRIS, C.M. Fundamentals of queueing theory. John Wiley & Sons, Inc. New York, NY, USA. 1985

[9] HAUCK, S. Configuration prefetch for single context reconfigurable coprocessors. Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, 1998.

[10] HAUSER, J.R, AND WAWRZYNEK, J. Garp: A MIPS Processor with a Reconfigurable Coprocessor. IEEE Symposium on FPGAs for Custom Computing Machines, 1997.

[11] HORTA, E.L, LOCKWOOD, J.W, TAYLOR, D.E, AND PARLOUR, D. Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration. Design Automation Conference (DAC), 2002.

[12] HUANG, C., AND VAHID, F. Dynamic Coprocessor Management for FPGA-Enhanced Compute Platforms. IEEE/ACM Int. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems (CASES), Oct 2008.

[13] HUANG, C. AND VAHID, F. Dynamic Transmuting Coprocessors. IEEE/ACM Design Automation Conference. DAC. July 2009.

[14] ISAACS, D., TREXEL, E., AND KARSTEN, B. Accelerate System Performance with hybrid multiprocessing and FPGAs. Embedded Systems Design, 8/15/2007.

[15] JEFFERSON, D. AND REIHER, P. 1991. Supercritical speedup. In *Proceedings of the 24th Annual Symposium on Simulation* Annual Simulation Symposium. IEEE 159-168

[16] NAGUIB, Y. N. AND GUINDI, R. S. 2007. Speeding up SystemC simulation through process splitting. In *Proceedings of the Conference on Design, Automation and Test in Europe*.

[17] NOGUERA, J., BADIA, R.M. Dynamic run-time HW/SW scheduling techniques for reconfigurable architectures. CODES-ISSS, 2002.

[18] PÉREZ, D. G., MOUCHARD, G., AND TEMAM, O. 2004. A New Optimized Implemention of the SystemC Engine Using Acyclic Scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*

[19] SIROWY, S., MILLER, B. AND VAHID, F. Portable SystemC-on-a-Chip. UCR-CSE-TR-052709. Technical Report. April 2009.

[20] WANG, Z. AND MAURER, P. M. 1990. LECSIM: a levelized event driven compiled logic simulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (Orlando, Florida, United States, June 24 - 27, 1990). DAC '90