Efficient Usage of Concurrency Models in an Object-Oriented Co-design Framework

Piyush Garg
Sandeep K. Shukla
Rajesh K. Gupta

ICS
TECHNICAL REPORT

Technical Report # 01-52
(September 17, 2001)

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425

Information and Computer Science

University of California, Irvine

Efficient Usage of Concurrency Models in an Object-Oriented Co-design Framework

Piyush Garg Sandeep K. Shukla Rajesh K. Gupta

Center for Embedded Computer Systems,
Department of Information and Computer Science,
University of California at Irvine,
Irvine, CA 92697
E-mail: {pgarg,skshukla,rgupta}@ics.uci.edu

Contents

List of Figures

List of Tables

Abstract

Increased complexity of micro-electronic systems demands a need for efficient system level models. System level models can provide detailed architectural simulation results to make architectural tradeoffs in the early stages of the design process. For effective system-level design, there is a need for an efficient co-simulation model for precise and speedy system level simulation and design exploration. Recently, several object-oriented language based co-design frameworks have been proposed for hardware modeling at the system-level. In this paper, we focus on modeling concurrency in these frameworks and how it can be used to improve the efficiency of system-level simulation. Specifically, we examine the use of threads to implement process concurrency and compare against non-threaded implementations using function calls. This distinction is important because it determines the choice of the class (e.g., synchronous/asynchronous processes, blocks etc) to use for specification of a given system block or behavior. Although, it is a commonly held belief that usage of threads might slow down the simulation performance due to context switch overheads, our analysis and experiments show, that by judicious choice of the threading model, and by striking a balance between the usage of threads and function calls in a system model, one can improve the simulation speed. Hardware designs being inherently concurrent, it is very important to be able to express concurrency in co-simulation models without compromising simulation efficiency. Hence our methodology is based on analysing the concurrency in the model, before expressing them in a high level language. To demonstrate the simulation effectiveness, we present an architectural model of a system with adaptive memory hierarchy and demonstrate the effect of modeling choices on the overall simulation efficiency.

1. Introduction

System level models can provide detailed architectural simulation results to make architectural tradeoffs in the early stages of design with sufficient precision and speed. A complete system model in VHDL/Verilog would be detailed but too slow for architectural exploration. On the other hand, programming languages such as C/C++ are meant for software design and a complete system model in these languages would not be accurate enough to characterize the timing performance. So there is a need for a efficient co-simulation model for precise and speedy system level simulation.

Various co-simulation tools [2][3][4][12][15] are available commercially. Most of these tools provide a heterogeneous environment for co-simulation of C/C++ and VHDL/Verilog. The communication between software and hardware models are done using remote procedure calls or some form of interprocess communication (sockets). There is an overhead in passing data back and forth between the largely HDL-based hardware world and the largely C/C++ based software world during such a co-simulation.

Recently, several object-oriented language frameworks based on C++[5][13][14][18][19][20] or Java[8][10] and specialized language frameworks[7][9] have been proposed for system design. The advantage of having a system model in object-oriented frameworks is that it preserves synthesizability and allows late binding of the components to HW/SW partition because the models of software components can be readily interchanged with models of hardware components in them. Since these language frameworks are extended C++ or Java, one can compile and generate an executable simulator for the whole system. Such an executable simulation is much faster than interpreted simulation. Simulation paradigms used by these frameworks either express concurrent behavior by sequentializing them using function calls (often facilitated with relaxation mechanisms for propagating signals), or map concurrency directly to user level threads. Since threads incur context switching overhead, one is led to believe that usage of threads might compromise simulation efficiency. However, our experiments show that the simulation efficiency can be enhanced by a methodological tradeoff between threads and function calls. Our further experimentation also shows that this tradeoff methodology is largely dependent on the threading model (*cooperative vs. preemptive*). We attempt to obtain a characterization of the design space by partitioning it in such a way that the system components, independent of other components and computationally intensive, are mapped to threads, while the components that are non-computationally intensive and form a dependency chain among themselves are modeled using functions. This way, we experimentally show,

that by proper partitioning of the design space, one can achieve better simulation speed, as compared to fully sequential function call based model. In our experiments, we also find that the choice of threading model can affect this methodology significantly. The experiments described in this paper are based on a *cooperative* threading model, where threads are scheduled by the application, and not by O/S kernel, and are not *preemptive*. Our recent experiments show that a preemptive threading model does not show the performance characteristics we describe here. In fact, we suspect that unless multiple processors are used, a preemptive threading model will make the performance deteriorate as the degree of concurrency is increased by introducing more threads. However, we don't go into any details of the preemptive threading based experiments in this paper. For experimentation purposes we used SystemC[18] as our modeling framework. The version of SystemC, we use, used a cooperative threading model, by building the system on top of quick threads library. However, for any other system level modeling framework[5][13] which offers thread versus function call mechanism, the similar methodology can be applied.

In section 2, we recapitulate some basic co-simulation concepts. In section 3, we discuss the thread based versus the function call based modeling in SystemC (namely asynchronous and synchronous processes versus asynchronous blocks). Section 4 describes the architecture of AMRM [1] in brief to show the overall design complexity, and to describe various main modules which are natural candidates for our manual partitioning. Also, we describe the experimental results, and compare the variation of partitions and resulting simulation efficiency in this section. Finally, section 5 summarizes the paper.

2. System Modeling and Co-simulation

In a hardware-software co-design framework, the application generating stimuli can be the part of a software while the target system can be a part of the hardware. To carry out execution-driven simulation on the such a system, we need aco-simulation model of it. A co-simulation model can be created by using *Programming Language Interface (PLI)* by modeling the hardware system in a HDL and the software application in a high level programming language. But such a co-simulation model would be slow due to the interfacing overheads between two different languages. Another way to do co-simulation is through system calls like *sockets*. Sockets are used to create an interface between the software and the hardware simulator for communication. Many commercial tools[2][12][15] use sockets for co-simulation. Here as

well, sending data to and fro across the interface slows down the simulation speed. Moreover in the above two approaches, the hardware and software components, being designed in two different languages, are not readily interchangeable across HW/SW partition to do design tradeoffs. The third approach is to use *function calls* for co-simulation. Unlike sockets, function calls do not have communication overheads, so they are fast. In the model of a system in object-oriented modeling paradigm, co-simulation is achieved through function calls.

3. Co-simulation Efficiency

For the fast simulation of a system model, the modeling language should meet two requirements: first, efficient modeling of the hardware and second, efficient way to model the concurrency. SystemC meets the first requirement by providing efficient support for reactivity in form of two constructs : *watching* and *waiting*[11]. To meet the second requirement, SystemC provides two types of processes: *synchronous* and *asynchronous*. A *synchronous* process is a process that communicates with other processes only at specific instances of time determined by the clock edge to which the process is sensitive. On the other hand, *Asynchronous* objects are more general form of the synchronous processes that can be used to model any kind of circuit. SystemC provides two types of asynchronous objects: an *asynchronous process* and an *asynchronous block*.

In the SystemC, synchronous and asynchronous processes are implemented as *threads*. As the threads have context switching and synchronization overheads, in a uniprocessor environment they might cause slow execution. But at the same time, threads bring concurrency in the model which can lead to certain amount of speedup in simulation due to concurrent execution. This is visible when certain aspects of the model, test data, etc., are read through disk I/O.

On the other hand, asynchronous blocks are not thread based. They are evaluated by the *main* SystemC kernel thread. As the result, an asynchronous block simulates significantly faster than a synchronous or an asynchronous process. But, the execution of asynchronous blocks is sequentialized during simulation, and hence results in loss of speed that can be achieved due to concurrent execution.

Thus an analysis of the computational and communication nature of the components of a system can guide one to decide whether a specific component should be modeled as a thread or as a function. We have done this analysis manually to partition the components of system model among processes and

blocks.

4. Implementation and Experimental Results

We now describe the design of a realistic system by taking the example of the AMRM machine which has been modeled as well as implemented in a board level design.

4.1. The AMRM Machine Architecture

The *Adaptive Memory Reconfiguration Management*, or the AMRM[1], project aims to find ways to improve the memory system performance of a computing system. The AMRM machine uses reconfigurable logic blocks integrated with the system core to control policies, interactions, and interconnections of memory.

Figure 1 shows the main components of the AMRM machine. It consists of a general 3-level memory hierarchy plus support for the AMRM ASIC chip implementing architectural assists within the CPU-L1 datapath. The FPGAs contain *controllers for the SRAM, DRAM and L1 cache*. A 1 MB SRAM is used for tag and data store for the L1 cache. A total of 512 MB of DRAM is provided to implement part of the cache hierarchy.

The host processor writes commands to specific addresses in the *PCI* address space and the *Command Processor* reads a command and launches its execution in the AMRM system. In order to perform detailed and accurate simulation of diverse memory hierarchy configurations at any clock speed, a *hardware "virtual" clock* has been implemented as part of the performance monitoring hardware. The *AMRM chip* uses an ASIC implementation of the AMRM cache assist mechanisms. It may contain a write buffer or an application-specific prefetching unit to access the L2 cache.

4.2. The AMRM Co-simulation Model

The AMRM machine implementation is divided into two phases. First phase consists of implementation of a board-level prototype which only has L1-cache management i.e. FPGA0, PCI Interface, SRAMs and DRAMs; followed by a second phase single-chip implementation of the complete cache memory system. The phase I of the project has already been implemented. The block diagram of the design inside FPGA0 is shown in figure 2. The design is partitioned into following modules:

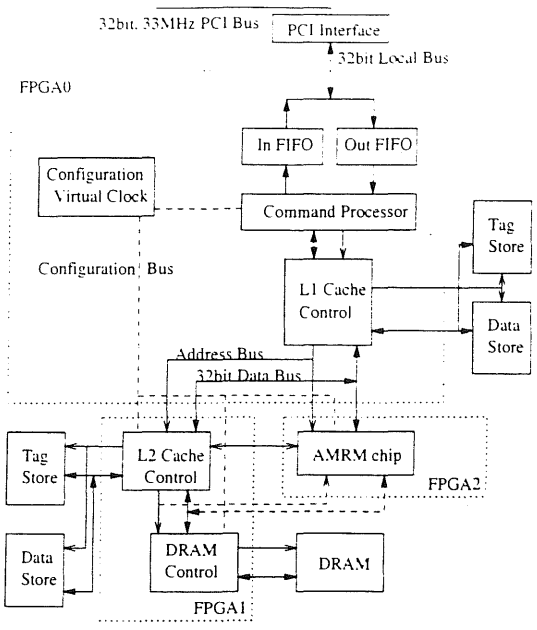

Figure 1. Block diagram of AMRM machine

SRAM Controller: Interfaces the synchronous SRAMs on the board.

DRAM Controller: Interfaces the DRAM modules on the board.

PLX Interface: Interfaces with the PLX chip local bus. It contains input and output register files.

Command Processor: Reads commands from the PCI interface, decodes and invokes the correct module to perform the operation as describe in previous section.

L1 Control: Implements the L1 cache controller. The cache controller design is configurable over a range of cache sizes and cache-line sizes.

AMRM Register File: Contains control, status and performance monitoring registers.

Virtual Clock: Implements the virtual clock functionality as described earlier.

The complete phase I implementation has been re-modeled in SystemC to study the effect on simulation speed. Also the SRAMs and DRAMs on the board are modeled in SystemC. In the VHDL implementation, these memory models were generated using memory modeling tools from Denali[6]. The generated models were then interfaced with VHDL implementation through CLI interface, provided in Synopsys VHDL simulator (VSS)[17].

In the VHDL implementation of AMRM prototype, there are 20 varying sizes of *Entities*, each of

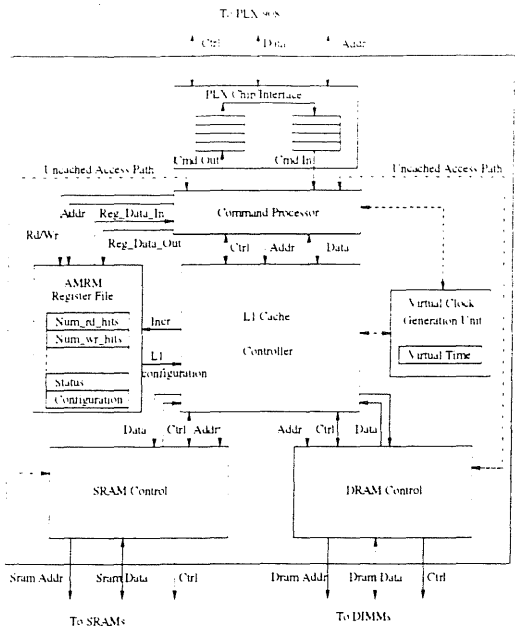

Figure 2. Block diagram of Phase I prototype

which have one or two *processes*. All the 20 entities are at RT-level and are synthesizable using Synopsys Design Compiler [16]. For simulation purpose, a test-bench consisting of seven processes have been created.

The SystemC implementation of AMRM prototype also contains 20 entities. All of the processes inside these entities are modeled as *asynchronous blocks* and integrated using *sc_module* construct of SystemC. All these processes are modeled at RT-level and hence synthesizable. The processes of the test-bench are modeled as *asynchronous processes*. The test-benches are modeled as processes because a process execution can be suspended and resumed from the same point without explicitly saving the state by creating state machines. But at the same time processes are not synthesizable using Synopsys Design Compiler. Overall, there are 83 processes in SystemC implementation, of which six are asynchronous processes and rest of them are asynchronous blocks.

4.3. Simulation Results

The VHDL and the SystemC model of AMRM prototype board has been simulated to see the difference in simulation speed. Note that the VHDL and the SystemC implementations are exactly the same in structure. All the simulations are run on Sun Ultra 5 Sparc station in a single processor environment. The

VHDL simulator used for simulation is Synopsys VSS. SystemC simulation is only 2.5x faster than VSS. The test-bench contributes six asynchronous processes in SystemC implementation. The total number of context switching among them is 8,948,235. As discussed in the previous section, asynchronous processes are implemented as threads in SystemC and threads have context switching overheads. To see the effect of context switching on the simulation speed, we have converted all asynchronous processes to asynchronous blocks. When all the processes are switched to asynchronous blocks, the simulation time has decreased by almost 50 seconds (i.e. almost by 7%). This is because these six processes form a dependency chain where first process waits for the second to finish, the second wait for the third and so on. As the result of the dependence chain, they execute more or less in a sequential fashion. Thus the scope of concurrency among them is very less and, when modeled as asynchronous processes, there was an overhead due to context switching.

Next, we moved one-by-one all the 83 processes modeled as asynchronous blocks to asynchronous processes. The graph in figure 3 shows the effect on simulation time as the asynchronous blocks are made into asynchronous processes. The graph plots the simulation time in seconds versus the number of context switching in millions. The points on the curve mark the main components of AMRM prototype as they moved from asynchronous blocks to asynchronous processes. The different curves corresponds to the different order in which the blocks are moved to processes. When all the asynchronous blocks are moved to asynchronous processes, the total number of context switching are 57,309,966. This has degraded the simulation speed by almost 13%.

We can infer from the graph that the simulation time changes differently depending on the order in which the asynchronous blocks are moved to asynchronous processes. On all the plotted curves, simulation time decreases when *SRAM Controller, DRAM Controller* and *Virtual clock* are modeled as asynchronous processes. This is because these components are computationally intensive and do not form any dependency chain among themselves, hence the scope of concurrency among them is very high. As the result, speed gains from concurrency are higher than speed losses due to context switching. This explains why on curve 3, when these three components are moved to asynchronous processes we achieved better simulation time than when all the processes are modeled as asynchronous blocks. In the case of other components, simulation time increases as the asynchronous blocks are moved to asynchronous processes. The degradation in simulation speed is due to the lack of independence among

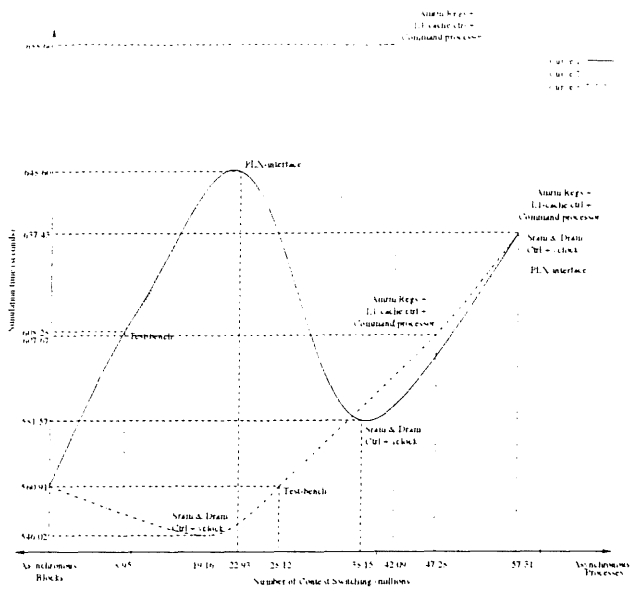

Figure 3. Effect on simulation speed as asynchronous blocks converted to asynchronous processes

them and the increment in number of context switching. Thus modeling SRAM controller, DRAM controller and Virtual Clock as separate threads and rest of the system as asynchronous blocks gives us the best partition, while modeling them as asynchronous blocks and the rest as threads leads to the worst partition. There is almost a 20% simulation speed improvement in first partition over the second partition.

When we replaced the cooperative threading package in SystemC, with a preemptive package (POSIX threads), the results were markedly different, because in the uniprocessor simulation, with computationally intensive tasks, preemptive model imposed too many interruptions. However, we need to do further experiments to explore, if the preemptive threading model will perform better in a multiprocessor based simulation environment.

5. Summary

The advantage of our co-simulation model is that the design is synthesizable and the components can readily be switched between hardware and software partition without breaking the overall integrity of system. We have illustrated the effect of modeling thread based concurrency on the AMRM system

model in SystemC framework. Our experiments show that cooperative threads result in the degradation of overall simulation speed if the components in a system model form a dependency chain, in which one component waits for the result from the other component before starting its computation. But in a computationally intensive system, where components are independent, modeling using threads results in speedup. The future work will include a more elaborate characterization of the design space to help the system designer to decide whether a specific component in the system should be modeled using a thread or a function, and more experiments on the effect of threading model, and the uniprocessor vs. multi-processor machines.

References

[1] AMRM Website, http://www.cecs.uci.edu/~ amrm.

[2] Arexsys, http://www.arexsys.com. *Cosimate.*

[3] Co-design Automation Inc., http://www.co-design.com. *Superlog : Evolving Verilog and C for System-on-Chip Design.*

[4] CoWare N2C Website, http://www.coware.com.

[5] CynApps Inc., http://www.cynlib.com. *Cynlib Reference Manual.*

[6] Denali Software Inc., http://www.denalisoft.com. *Memory Modeler's User Guide.*

[7] D. D. Gajski. *SpecC: Specification Language and Methodology.* First Edition, Kluwer Academic Publisher, 2000.

[8] R. Helaihel and K. Olukotun. JAVA as a Specification Language for Hardware/Software Systems. In *Proceedings of the ICCAD-97*, pages 690–697, San Jose, CA, November 1997.

[9] A. Jerraya and K. O'Brien. SOLOR: An Intermediate Format for System-Level Modeling and Synthesis. In *Computer Aided Software/Hardware Engineering, J. Rozenblit, K. Buchenrieder, eds, IEEE Press*, 1994.

[10] T. Kuhn, W. Rosenstiel, and U. Kebschull. Description and simulation of Hardware/Software Systems with JAVA. In *Proceedings of the 36th Design Automation Conference*, pages 790–793, New Orleans, CA, June 1999.

[11] S. Liao, S. Tjiang, and R. Gupta. An Efficient Implementation of Reactivity for Modeling Hardware in Scenic Design Environment. In *Proceedings of the 34th Design Automation Conference*, pages 70–75, Anaheim, CA, June 1997.

[12] MentorGraphics, http://www.mentorgraphics.com. *Seamless.*

[13] OCAPI Website, http://www.imec.be/ocapi.

[14] P. Schaumont, S. Vernalde, L. Rijinders, M. Engels, and I. Bolsens. A Programming Environment for the Design of Complex High Speed ASICs. In *Proceedings of the 35th Design Automation Conference*, pages 315–320, San Francisco, CA, June 1998.

[15] Synopsys, http://www.synopsys.com. *Eagle*.

[16] Synopsys Inc., http://www.synopsys.com. *Synopsys DC User Manual*.

[17] Synopsys Inc., http://www.synopsys.com. *Synopsys VSS User Manual*.

[18] Synopsys Inc., http://www.systemc.org. *System C Reference Manual, Release 0.9*.

[19] D. Verkest, J. Cockx, F. Potargent, G. de Jong, and H. D. Man. On the use of C++ for System-on-Chip Design. In *Proceedings of the IEEE CS workshop on VLSI-99*, pages 42–47, Orlando, FL, April 1999.

[20] S. Vernalde, P. Schaumont, and I. Bolsens. An Object Oriented Programming Approach for Hardware Design. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, pages 68–73, Orlando, FL, April 1999.