Reduction of Co-Simulation runtime through parallel processing

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Jason Coutu

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

During the design phase of modern digital and mixed signal devices, simulations are run to determine the fitness of the proposed design. Some of these simulations can take large amounts of time, thus slowing down the time to manufacture of the system prototype. One of the typical simulations that is done is an integration simulation that simulates the hardware and software at the same time. Most simulators used in this task are monolithic simulators. Some simulators do have the ability to have external libraries and simulators interface with it, but the setup can be a tedious task. This thesis proposes, implements and evaluates a distributed simulator called PDQScS, that allows for speed up of the simulation to reduce this bottleneck in the design cycle without the tedious separation and linking by the user. Using multiple processes and SMP machines a simulation run time reduction was found.

Acknowledgements

Dwight, David, Doug, Andrew, my wife and kiddies... They have all had much more patience with this than I deserve, Thank you!

This thesis is dedicated to my wife. When this thesis started, we had one wonderful toddler in the house. During this thesis we had two more wonderful, busy children. My wife has been blessed with patience as she has raised these kids and stayed sane while I worked away on my thesis. Thank you dear.

Contents

List of Tables

List of Figures

List of Abbreviations and Glossary

SMP	Symmetric shared Memory multi-Processing
DMP	Distributed Memory multi-Processing
API	Application Programming Interface
MPI	Message Passing Interface
IPC	Inter-Process Communication
HDL	Hardware Description Language
PLI	Programming Language Interface
IDE	Integrated Development Environment
IP	Intellectual Property
DUS	Design Under Simulation, used to differentiate the testbench from the design in the simulation
Co-Designed systems	The end result of the Co-Design process
Time quantum	the smallest unit of time to be simulated by a discrete event simulator
Process	a single running instance of a program; a program may spawn multiple processes, but each process has its own memory space
Thread	a single line of execution of a program, programs may have many threads which all share the same memory

Chapter 1

Introduction

Modern electronic systems are composed of both hardware and software components. These systems range from network routers and sensors to control systems in cars and complete microprocessor environments that include operating systems. These new systems have more functionality and complexity than the previous generations of systems. Hardware die sizes are continually decreasing, and transistor gate counts are increasing with every new generation of devices. The range of functionality and complexity of these new systems has increased continually. In 2004 when this thesis began, the International Technology Road-map predicted that over the next 15 years, memory in devices continue to double every 3 years, and chip functionality will continue to increase by 50% every three years[13]. In their 2008 report [14], these trends have held up and are predicted to continue to hold true for the next 15 years.

The designers of these systems must ensure that the Design Under Simulation(DUS) meets various functional and performance requirements. Simulations are done at various levels of detail to predict how the systems will perform if manufactured as designed. As these systems become more complex, the simulations that verify them have to become more complex, and thus, consume more resources. There are two resources related to the run time of a simulation: 1) Time - how long the simulation takes to run, and 2) Processor - how much computational effort is available to the simulation. Additionally the level of detail has an effect on the run time. A simulation at the same level of detail for a more complex simulation will take longer to run. The resource that takes no conceptual effort to consume more of is time. If the designer simply uses the same computer, with the same simulator at the same

level of detail for a more complex design, it will take longer to simulate. This can create a costly delay in the development of new systems.

The simulations that are used in the current design are numerous and time consuming [16]. Some simulations can take days or weeks to complete. This delay of waiting for the results of a simulation can be costly, as more refined design cannot take place until the simulation results have been analyzed. Additionally, there can be large numbers of simulations that need to be run to evaluate possible design alternatives. Thus, reducing simulation time will help reduce design time for new systems.

There are several ways to reduce the simulation run time. Two possible solutions are mixed level simulation, and distributed simulation. Mixed level simulators only use complex models for sections of the simulation when necessary, the remainder of the simulation is run at a higher (more abstract) level that completes faster [1]. Distributed simulations take advantage of parallelism in the system and use multiple CPUs to run the simulation.

Mixed level simulations allow portions of the simulations to run at differing levels of detail, allowing some speed up when some sections of the DUS are run at a more abstract level. Some sections/modules that have already been proven to function as necessary, or represent IP of other companies, can be run as a high level simulation. This allows for reduced simulation run time for these elements of the design, while new sections can be simulated in a more detailed fashion. This can be taken to an extreme where all modules are simulated at a high level and only the intercommunication and connections are simulated in detail.

The strategy for reducing simulation time that this thesis examines is to have portions of the simulation run separately in parallel with a unified simulation clock. Distributed simulations are not new to computer science, or to hardware design. There has been previous conjecture that distributed simulations can reduce run time in complex computer system design simulations [23, 34, 51].

Partitioning of modules into groups for evaluation is nessasary for a distributed simulation. This partitioning can automated or manual. Automatic partitioning

can create modules more quickly than manual partitioning, but this partitioning often creates overhead and the simulation still needs to be explicitly created as a distributed simulation. The simulation can also slow down due to increased overhead and poor partitioning. Manual partitioning results in faster simulations, but makes the designer choose where the simulation is split up and partitioning may take longer. The solution proposed in this thesis looks at enabling the designer to have the maximum number of distributed partitions in a single simulator without having to explicitly create the partitions. Distributing the computing load will allow for faster run times.

Distributing a simulation does not guarantee a faster run time. Communications cost is often a limiting factor in distributed simulations. The communications could take longer than the time to run the simulation on a single processor. Balancing communications cost and distribution is central to an effective solution. The largest component of the communications cost is transfer of the data between the distributed parts. Some solutions will not need the information to be transfered as it is used in place, while other solutions will require the data to be replicated to every location that it is needed. A distributed grid will require the data to be synchronized between machines, while a SMP machine could use shared or local memory and thus not need any data transfer.

Using current simulation techniques, the units that the simulations are broken into for distributed simulation tend to very coarse. This is because the partitioning of the system design for simulation purposes needs to be done by hand, if the benefits of reduced simulation time are to be realized [19]. Often, the partitioning is done solely to use two different simulators because of simulator limitations. One simulator may not be able to deal with a type of design, so the design is partitioned and one portion is run on a specialized simulator. This can speed up the simulation, but the manual partitioning reduces the number of partitions due to the amount of designer work involved.

Several areas need to be examined in any project that wishes to address the run time speed issues in Co-Simulation with a distributed solution. These include

how much speed up is possible due to the structure of the system being simulated, and how the software interacts with the hardware in the Co-Designed system. The main problem is how much can be done in parallel due to the way the hardware and software of the DUS interact.

1.1 Definitions

For the purposes of this thesis, the following terms are used as defined below:

- Co-Design: The design of a system in which the hardware and software are codependent, and some modules may be either software or hardware in the final design.
- Co-Simulation: Simulation of a system that requires the simulation of both the hardware and the software running on the hardware and/or the simulation of the hardware and external systems that communicate with the hardware system.
- Distributed simulation: simulations that use multiple processing units.
- Simulation: A step of verifying the functionality and validating the correctness of a design in a controlled environment. Often used before a devices is created to validate the design.
- Monolithic simulator: A simulator that uses one large process to do all of the work for a simulation.
- Partitioning: The division of a device into modules for separation into hardware/software components or for distributed simulation.
- Concurrency: The ability of a computer program to perform multiple operations simultaneously. One instance of concurrency is a pipeline where each stage is always executing on a piece of data, but every stage operates continuously. After completion of processing, an individual piece of data is handed to the next concurrent process, while new data is received.

- Parallelism: The ability of a set of related tasks to be completed at the same time without affecting each other.

- Design Under Simulation (DUS): The portion of the simulation that is specification for the new system, device or component.

- Testbench: The portion of the simulation that provides the framework for setup, testing and running the simulation to test the DUS

1.2 Motivation and Thesis Statement

As systems get more complex, the partitioning of systems into software and hardware components, especially co-designed systems, becomes a large and tedious task. There are two types of partitioning that are done. The first form of partitioning determines what will be done in software and what will be done in hardware with respect to the target system. The second type of partitioning is an artifact of distributing the simulation. In this case the partitions determine on which computational node each portion of the simulation will run.

Partitioning for the hardware/software division is done repeatedly during the design cycle of a new system. The partitioning can determine the cost, speed, extensibility and whether the device meets its timing constraints. Each time a module is chosen to be placed in either hardware or software, each of these aspects of partitioning is affected. Balancing the partitions so that the system configuration is optimal is a repetitive task that requires multiple simulations. This makes the simulation time an important factor in the design cycle.

When a distributed simulator is used, partitioning the system into modules to run on each computational node is also done. There is a need to reduce the communication between the modules so that communication does not dominate the simulation. Careful partitioning, often done by hand, can affect the simulation speed.

An improper or poor choice in either or both of these types of partitions can create performance issues in either the simulation or the final device. The problem

of partitioning circuits into sets with minimal links across each cut has been found to be NP-Complete [4], while the partitioning of hardware/software without respect to the communication issue has been found to be NP-Hard [30]. A lot of work has been done to try to approximate a viable solution or solve a constrained version of the problem [33, 39, 40].

The number of simulations required to find a system configuration that is in the optimal solution space can be larger than it is possible to explore. Simulating to find the correct hardware/software partition is central to the design cycle for new devices. There are too many simulations that take too long for an exhaustive search of the solution space. This thesis *demonstrates a methodology and its implementation to reduce simulation time through the use of distributed simulations. Experiments will be conducted to verify that a distributed SystemC simulator, called Parallel Distributed Quick SystemC Simulator (PDQScS), is capable of reducing the time required to perform Co-simulation.*

1.3 Thesis Outline

Chapter 2 covers background material and previous research in Co-Design and Co-Simulation. This review covers papers on the design process of integrated circuits and larger systems, the software for these systems, partitioning problems in hardware/software systems, distributed simulations, and parallel computing. This material includes reviews of current and past Co-Design systems and Co-Simulations. Chapter 3 details the design of the experimental system. It provides an overview of the tasks involved in the experimental system, and how each task was completed. It also examines the research methodology that was used. Chapter 4 discusses the three results of the research for this thesis. The results are described in stages: 1) the original motivating experiments that were done to determine if the solution proposed by this thesis could work. 2) the development of a proof of concept simulator for the thesis. 3) performance results from the proof of concept prototype. Chapter 5 contains conclusions and a discussion of possible future work.

Chapter 2

Background and Related Work

The Co-Simulation of hardware and software systems involves many related topics. The context and previous work of each of these must be explored to enable the reader to see the inherent difficulties in Co-Simulation and explain the motivation for possible improvements. The related topic areas are the following: design path, simulation of new systems, division of new systems into hardware and software, communication and techniques for parallel processing, and techniques for distributing simulations. The domain specific interpretation of terms are explained in this chapter. Further definitions can be found in the glossary.

The design path for new systems has multiple stages and has always been time consuming. One of the stages of the design path is simulation. As processors are able to do more work with advances in CPU chip technology[1] [2], the simulation time should fall. However, systems have become more complex, and simulation has taken more of the design time and not less [42]. Part of this complexity is the interaction between software and hardware components. These new more complex systems require that the software and hardware components both be simulated at the same time. The design of these systems is called Co-Design, and the simulation of these systems is called Co-Simulation.

New techniques and languages to describe co-designed systems were developed along with new simulation tools to work with them. In some cases, hardware description languages (HDLs) were extended to include the ability to attach external

[1] Moore's law states that the number of transistors on a complex integrated circuit doubles every two years. It has been shown to hold over the past decades, and is predicted to continue to hold for the next 15 years.

[2] The ITRS reports are available at http://public.itrs.net/

software module simulators [37]. In other cases, entirely new languages and methodologies were developed to try to deal with the new complexities in Co-Design and Co-Simulation [31]. In contrast to these, some programming languages have been extended to become HDLs [7].

The general partitioning problem describes the complexity of finding optimal groupings of N objects into M groups. This problem requires that every combination be tested, or N^M combinations, to find the optimal configuration. Optimal is defined by the version of the partitioning problem under investigation. In *Co-Design*, the grouping of modules across the hardware/software boundary is part of the partitioning problem. Optimal in this case is defined by the designer of the co-designed system. The partitioning problem in distributed simulations also deals with allocating portions of the simulation to separate computational nodes. Optimal in *distributed simulation* is the configuration that will provide the best possible speed up for the available resources. These partitioning problems in systems design and distributed simulation are constrained versions of the general problem, but they still have a high degree of computational complexity.

Modern systems have large amounts of parallelism inherent in the design [46]. Parallel processing techniques should be able to use this target system parallelism to reduce simulation time. Parallel processing can be achieved in many ways. Multi-processor machines, cluster computing, and grid networks all allow applications to run more then one process at a time, thus achieving parallel computation. Symmetric shared-memory multi-processor (SMP) machines offer the ability to use one memory bank and several processors to complete a job that has tasks that can be done in parallel. Cluster computers require dedicated hardware and communications. An alternative to clusters is grid computing. Grid computing often requires the use of special communications methods and software to control the system.

In the past, attempts at distributed simulations of hardware have been tried. TyVis [47] and Ptolemy [31] are examples of major previous attempts. They were unable to do both the high level and low level simulations within their frameworks. As systems required more and more simulations to be done outside of the framework,

distributed simulations were dropped. Non-distributed simulation frameworks were created that were capable of simulating at all levels.

The rest of the chapter covers five topics of background information and related work undertaken with respect to that topic. The first section covers the design and Co-Design process for new systems and the differences. The second section details how and why new systems are simulated, and the problems that Co-Simulation adds over simple simulation. The third section covers the two partitioning problems: hardware/software partitioning, and module distribution partitioning during simulation. The fourth section examines parallel processing systems. The final section looks at techniques for and previous work regarding distributed simulation.

2.1 System Design

The traditional design path for new HW-SW systems had several steps [45], all of which still exist in some form in Co-Design [35]. The traditional design path is described below and in figure 2.1:

1. Determine the statement of work. A description of the task to be completed, its inputs and its outputs.

2. Requirements specification produces a document that lists all of the features and performance requirements for the system.

3. Functional models are built and tested to prove that the algorithms chosen will in fact meet the functional requirements.

4. Logical models are created that implement the algorithms in terms of physically implementable units.

5. The results of testing of this model are compared to both the requirements and the results from the functional model. This is done to ensure that no side effects or artifacts have slipped into the design in the translation to the logical level. If problems or mistakes exist, the process goes back to step 3.

6. The system is simulated at the register transfer level. This simulation includes timings of the signals as they pass through the components and their connections. If the simulation indicates that the module/device will not perform as designed, return to step 4.

7. A prototype system is produced and tested against the test bench that was used in the simulations.

8. If at this point, the prototype is deemed to not perform satisfactorily, two options exist. If there are only a few changes (i.e. small errors) that can be fixed by changing some of the connections on the physical board, this is often done. Otherwise, the system needs to be re-synthesized with the problems fixed. (i.e. go to step 3 or 4.)

9. Once the prototype has been produced, a copy is turned over to the software team for testing of the software components.

In this older design path, there were some significant problems [54]. Software development is delayed until near the end of the cycle, and the division of hardware and software is determined before design is completed and thus before detailed simulations can be run. Changes in the interface between the hardware and software modules may cause redesign and re-implementation of software modules. This separation of the hardware and software teams requires clear and unambiguous documentation, including recording the changes as they happen so that the changes can be identified and tracked.

The major limitation of the approach was that the division of what components are implemented in software and hardware respectively is determined only once, during the requirements elicitation. This decision is not looked at again in the traditional design path unless significant problems[3] are found. When significant problems are found, the system is redesigned and the partitioning is reexamined.

[3]A device that performs too slowly to be useful in the market or a device that consumes too much power are examples of significant problems.

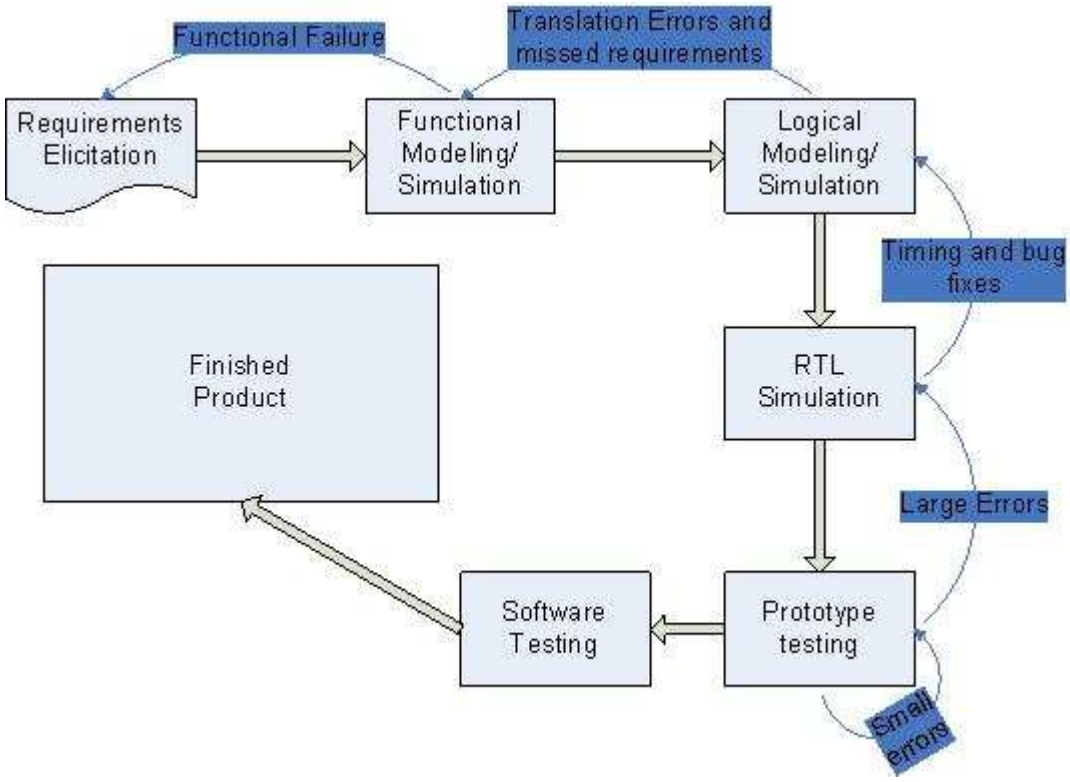


Figure 2.1: The traditional design path

This singular look at the partitioning can create a system that does not have a good cost/performance ratio.

As these design path limitations were recognized, the design path for new systems underwent changes. These systems have become more complex and the design problems more common [54]. Systems now need more simulations, contain more complex designs and have tight timing requirements. With this complexity has come the necessity to simulate hardware, software and the inter-communication at the same time. This requirement has produced changes in the design path, design environments and simulation tools. Simulators that could only deal with hardware needed to be extended to allow for simulation of software modules [48]. Software designers can no longer wait for the system prototype to be completed before testing as the division into hardware and software modules is no longer predetermined.

These problems, and the tighter coupling of hardware and software have spawned new design paths and languages [7, 15]. Some of the new design paths combine new languages and methodologies into a unified design environment, while others try to leverage the existing knowledge base of developers by modifying tools and languages that are in use. Both of these solutions have common elements to the new design path.

In Co-Design, the design path has not changed in essence, except that now there are additional steps earlier on to integrate the software development, and possibly move some of the modules across the hardware/software boundary. This ability to move components across the boundary has necessitated further simulations at the lower levels to ensure that the requirements are still met. Moving modules across the hardware/software boundary changes the cost, flexibility and performance of the system. This relationship can be seen in Figure 2.2. As the figure shows, as more modules are placed in hardware, performance increases while flexibility decreases. Cost may or may not change as modules move across the partition. A low cost generalized processor could be swapped out for a high cost Application Specific Integrated Circuit (ASIC), or the change may mean that some of the hardware is not needed at all.

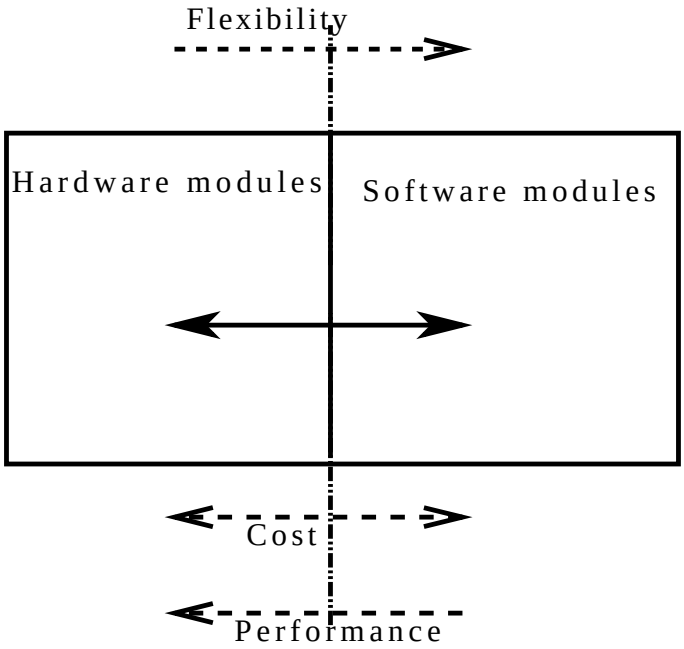


Figure 2.2: The boundary between hardware and software

An optimal configuration of the system could mean lowest cost configuration, fastest, best cost for performance or the system that will be the most flexible. The solution space for this can be seen in Figure 2.3. This figure represents the solutions that have the acceptable cost, performance and flexibility. The zones marked A1, A2, and A3, denote sub-optimal solutions. In particular, A3 represents the solutions that meet the performance and flexibility requirements, but will be too costly to manufacture.

2.1.1 Co-Design environments and Co-Design languages

Hardware description languages, or HDLs, are used to design and test new systems. Two examples of these are VHDL and Verilog. HDLs alone are, however, not capable of expressing the design requirements or describing software systems. Thus, new languages and extensions to the HDLs have been developed to fulfill this need. This section provides a survey of these tools.

The earliest projects in the area were to replace the HDLs with a modeler board

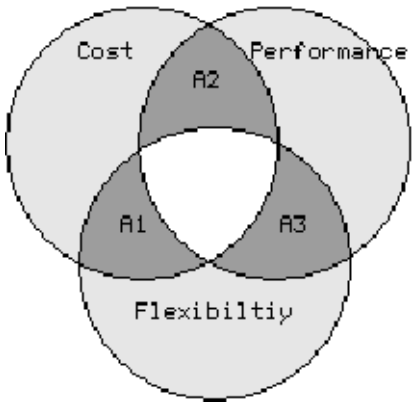


Figure 2.3: Systems configuration solution space

to allow output from software simulators to be used as input data for the hardware simulation. These systems had very long simulation run times [27]. This was due to the fact that the software simulation clock ran slower than the hardware modeler board clock which ran in real time. These early attempts required the hardware simulation to be reset and rerun every few time units of software simulation. This reset was necessary because the hardware simulations ran too fast and produced more data than the software simulation could process. Simulation speed decreased as the simulation progressed. This resulted in a delay for large simulations. The main disadvantage of this entire design approach, however, is that it requires the hardware partition to be set and a prototype created [35].

There are substantial differences between the industrial and academic solutions to the Co-Design problems. These include scalability, ability to leverage previous knowledge, and conceptual cohesiveness. The industrial solutions tend to be pragmatic, scalable solutions are built on/into existing tools and languages. This eases acceptance by industry system designers without extensive re-training. The academic solutions are often fresh starts to the problems which allows for a conceptually complete solution. These solutions are rarely scalable, however, and require learning of new design concepts and languages.

The two main HDLs that have the majority of the industrial market share are Verilog and VHDL [42]. Both of these HDLs have been extended to aid in Co-Design.

Both Verilog and VHDL have been extended to provide a Programming Language Interface (or PLI). The PLIs offer the most basic level of interaction, and may not be supported by the entire tool set. Verilog has been extended a number of times to aid in Co-Design. Under Verilog, there is the PLI [37], a scripting solution called VeriSchemeLog [29], and SystemVerilog[4].

VeriSchemeLog is an automation tool. It uses Scheme (a derivative of Lisp) to provide scripting support for common tasks. It is used to create interfaces between modules and external simulators [29]. These areas are prone to design mistakes and typographical errors. Automating the creation of these interfaces provides internal consistency and eliminates errors caused by spelling mistakes.

SystemVerilog is a standard which was developed by the IEEE and Accellera[5]. SystemVerilog is an additional specification to the Verilog specification that adds system verification elements and co-design constructs to Verilog. With this addition, Verilog will become a full spectrum co-design tool. SystemVerilog is physically implementable, which means that once designs are verified, a prototype can be manufactured without translation to another HDL.

There have been several industrial projects devoted to extending VHDL to enable it to describe a Co-designed system. Examples include the VHDL PLI [27], and Co-ware [50]. VHDL PLI allows designers to add links to software modules in VHDL design. Co-ware is a solution that emphasizes the connections between VHDL and C/C++ modules. These solutions take different paths to solving the problem as they have different views of what is the major problem.

VHDL PLI is an interface first devised to allow VHDL designs to indicate that information is required from an external software module. A PLI now exists in most HDLs, but for simplicity, this thesis will use PLI to subsequently refer to the VHDL PLI. PLI allows VHDL simulators to perform function calls on external simulators to retrieve information. This allows for the inclusion of the Application Programming Interface(API) to be built into the VHDL design. This helps solidify the API and

[4]www.systemverilog.org
[5]www.systemverilog.org

make the software interface less prone to change. The PLI made it possible to simulate software at the same time as hardware, but this did not integrate the software development in the hardware design process.

Co-ware [50] can use different languages for each type of module. It does this by separating the functional units from the communication channels. This allows reuse and refinement of modules without affecting the entire system. Co-ware can leverage both the C/C++ and VHDL knowledge bases with its mixed language approach. Co-ware was designed and created when co-designed systems did not yet have the access to Co-Simulation, but individual tools for simulation and verification were available. This approach allows for better communication between the hardware and software design teams, but the designs are still not integrated and changing a module from hardware to software requires the module to be recoded for the other simulator.

The early systems were then followed by the academic attempts to enable a single simulation/design tool to represent the entire system for its entire design path. A new system needed to integrate the design of the hardware and software components, and the ability to simulate both at the same time. These academic solutions involved completely new design languages and ways of thinking about systems design. Examples of these are Ptolemy [31], QUEST II [53], and Chinook [26]. Ptolemy attempts to find what they termed *better* solutions by looking at the system and its communications as a whole. QUEST II is an extension of the Time Warp simulator to allow for VHDL simulations. Lastly, Chinook looks at behavioral designs, and interactively helps the user determine the design for the system.

The goal of Ptolemy is not to reduce design time, but to improve the quality of design by examining the models of computation that are used [31]. Ptolemy uses the terminology of stars and galaxies to talk about the components and their groupings. This use of different terminology and the use of its own XML derivative language to describe how the "galaxies" are connected differs from the industry standard. Additionally, the language is not physically implementable or easily translated to an HDL that can be synthesized.

QUEST II is part of the SAVANT project at the University of Cincinnati. The

aim of the SAVANT project is to build freely available VHDL analysis tools [52]. The QUEST II project focused on the complexity of the system design as the main problem that was causing slower simulations [53]. Thus Wilsey *et al.* wrote modules for the Time Warp simulator [18] to enable VHDL simulations within the context of QUEST II [47]. Time Warp is a distributed simulator framework for using many interconnected computers to speed up simulation. They found that they could speed up VHDL simulations through distributed simulation.

Chinook is a high level modeling tool that aids developers in the choices for which modules to place in hardware or software [26]. Combined with its companion co-simulation tool Pia, Chinook offers rapid evolution of designs by offering quick high level simulations. Chinook is not designed to be a final development tool, only a preliminary exploration tool to assist with the hardware/software partitioning decision.

There have been two programming language-based systems for the co-design specification problem. These systems are SystemC[6] and SpecC [55]. Both of these new HDLs are based on the C/C++ programming language. They differ on how the hardware portions are specified, and simulated. The simulation portion of this discussion can be found in the next sub-section.

SpecC was proposed by Jianwen Zhu *et al.* [55]. This HDL was designed from the ground up to be a co-design modeling super-set extension to the C programming language. It has the ability to model a system in differing levels of abstraction at the same time. This allows the mixing of high level algorithms with low level RTL simulations in a single simulation. Any valid design, even if made of mixed levels, can be fed into a simulator for execution and testing. This allows modules to be developed and implemented independently. SpecC is a more academic solution and is not physically implementable.

In contrast to SpecC, SystemC has been proposed by a consortia of companies that are involved in systems design [38]. SystemC is also designed to enable designs to have multiple levels of abstraction combined into a single design. SystemC allows

[6]more information at www.systemc.org

the mixing of high level algorithms with low level RTL simulations in the same way as SpecC. SystemC also has a simulation core built into it. Thus, it can compile to a single application with a standard C compiler. This application is a simulation of the design along with its test bench. This inclusion of a test bench in the simulation allows for test bench reuse as the modules are refined. Additionally, the consortia responsible for the development of SystemC have stated that, in future versions of SystemC, hardware modules will be able to go from simulation to physical device with re-implementation of the design in a different HDL.

2.2 Simulation

Simulations can be broken into several groups: un-timed functional simulations, discrete event simulations, and continuous event simulations. Un-timed simulations do not use a clocking mechanism. The difference between discrete and continuous simulations is how event states are handled. While all three types are important tools, this section describes their differences and applicability to Co-Simulation of systems considered in this thesis.

Un-timed simulations do not need a clock of any type. These include Markov models and agent-based simulations, as well as algorithmic simulations. In the design cycle, these simulations are used to prove that an algorithm or process can achieve a design within specifications. Examples of these simulations are programs written to verify correctness of the algorithm. Often it is just the general algorithm that is tested. These simulations typically run at very high speeds and are only done at a preliminary level.

Discrete event simulations treat events and state as blocks and units. The clock cycle used in most system designs makes an excellent division for the blocks of events, and separate the simulation into blocks of time (i.e. a time quantum). Such simulations assume that state changes do not occur between the units, even though we know that, in reality, the value of a system variable may change several times during a clock cycle.

An example of the assumption that state changes are only important at the clock pulses is the modeling of the difference in time it takes a signal to travel over two connections of different lengths. While this time difference is minor, there is a difference. When the time quantum of a simulation is large enough, this difference is ignored. This can cause problems in some circuit simulations, but there are techniques to deal with this problem. Circuit designs which do not use a clock to control state changes, or for which the clock runs extremely fast, are susceptible to this problem. Signals arriving at different times can cause incorrect answers when they fail to arrive in the correct time quantum. In these cases, a discrete simulation with variable length time quantum resolution is used [43].

In continuous event simulations, state change is a continuum. Events can occur instantly, but the change does not have instantaneous effect in continuous event simulations. State changes are not on/off type decisions, but more like the result from an equation. Think of water flowing from a valve. We can look at the water and see that it is flowing, but when we first turned on the valve, was the valve completely on as soon as it was turned? Did the water flow at fully as soon as the valve was on? Continuous simulation has this same view of events and their effects. Events can start at times other than on a time quantum, and their effects are not instant.

How time is treated and used is a major difference between continuous and discrete simulations. In discrete simulations, all events happen at a point in time and all state changes only happen during events. In continuous simulations, events may have a start time, but the properties and state can change at points other than at that events point in time. Discrete simulations can be used to approximate continuous event simulation. Discrete event simulations make up the majority of simulations in design and Co-Design.

As mentioned earlier, the clock tick makes a useful barrier to determine events and state changes. Computers are often thought of as having on/off logic, and the clock as having precise square edges to the pulse. In reality, this clock pulse has a rising time, and a falling time that is non-zero. There are some simulations that use a modified sine wave for the clock pulse in the design and Co-Design process,

but often they are replaced with discrete event simulations that can approximate the results. Thus, they do not provide completely accurate results. Square waves and discrete simulations are used because in most systems, a clock pulse is used to coordinate execution. Continuous simulations are needed to describe circuits that do not use a clock, but that is out of scope of this thesis.

Discrete simulations include any simulation that uses time in non-overlapping blocks. In many discrete simulations, some events are generated from earlier events. Examples of this can be found in most levels of system simulation. A call to use a piece of data may require that the data is loaded from storage before it can be used. Additionally, if a task has been split into subtasks with modules each doing their part and handing the data on to the next module[7], then each module's completion causes the next to start with new data. Since the time to process an event or a module's actions is known, its results will trigger other modules after that period has expired in simulation time. Thus we can insert events in the queue to happen at the correct point in time.

Simulation time does not equate to real time. High level algorithm simulations often run faster than the target system will run, while low level system simulations will run slower than the target system. The differences in speed are due to the amount of detail required, and the complexity of the simulation. The closer the simulation is to simulating actual wires and transistors, the more details that will need to be tracked by the simulator and the slower the simulation.

Detailed Co-Simulation runs very slowly due to its complexity and the number of components that must be modeled. Co-simulations often combine hardware and software simulators. Early attempts at this involved re-running the software simulator until it reached the same time index as the hardware simulator, then processing the next step before restarting the software simulation [27]. This disparity was caused by the software simulator running faster than the hardware simulator could process the input data. This caused the final steps to be slow as they had to wait for the software simulator to continually catch up from the beginning each step. This was

[7]this is called pipelining

later fixed by synchronizing the clock pulses of the two simulators.

Advances have continued to the point where there are now unified simulators that can simulate both the software portions and the hardware portions of a system without the need to run separate simulators, such as: SystemC and SpecC as mentioned in the previous section. This advance has reduced the run time of the simulations [22], but the simulators are now CPU bound [19]. This implies that further run time reduction may be found by reducing or distributing the CPU load.

It is important to note that it is not possible to get the same behavior in a simulation as with the real device. If the design being simulated has some events which happen in a nondeterministic order, then it is impossible to create a simulator that will always give the correct answer, as the answer changes. If two modules write to the same location at the same time, a race condition exists and only one of the writes will appear to have occurred, namely the last write operation to be performed. The simulator does not know which module will execute last in the real device. The order that modules write their data in the simulation may be different in the real execution of the device.

The scope of this thesis is restricted to discrete event simulations of Co-Design systems. The majority of the simulations done during the design cycle are done as discrete event simulations. Continuous simulations are often replaced with discrete event simulations. Additionally, un-timed simulations run at very high speeds and do not need to be sped up further.

2.3 Partitioning

In this thesis, partitioning is used in two separate contexts. The primary aspect of partitioning is the partitioning of modules into hardware or software in the target system. The second aspect of partitioning is the partitioning of simulation modules to run on different computational nodes. Both of these aspects of partitioning have the same background material, but different implications for the work in this thesis. While the background material and problem scopes are related, there are some

differences in these partitionings that must first be discussed before an in-depth exploration of the partitioning problem can occur.

2.3.1 Partitioning for Simulation

Deciding where to run each of the modules, or even how to break a simulation into modules is classified as part of the partitioning problem. There are constraints on what makes a good partitioning. Modules need to communicate with each other in order to pass data and control messages. Distributing the simulation will enable/require some of these modules to run on separate machines. There is a cost in time incurred to communicate between modules across machines instead of within a machine or application. Thus, this partitioning to make simulation modules needs to be done such that it limits the communications cost between simulation modules as much as possible for that system configuration.

There are a great number of possible combinations for this type of partitioning. Given N components, and M computational units to spread them across, this equals $N^M/M!$ combinations to evaluate to determine the best solution. Since each of the N modules can be assigned to each of the M computational resources, but the resources can be equivalent, each partition can be assigned to any computational unit and still have the same result, thus, the M! is divided out to reduce the assignments of the same partitions to different resources. The problem of finding the optimal partitioning is considered to be NP-complete [33].

A common method for determining where to partition a system is called *min-cut* [2]. Min-cut partitioning attempts to break as few communication lines between modules as possible, while creating the required partitions. Not all min-cut solutions reduce communications. If two modules are highly coupled, the communications cost for running them on separate machines could outweigh the benefits. It can be better to cut many rarely-used connections than one often-used connection. Thus min-cut is not always optimal, but it can have good utility when communications costs are low.

Simulation constraints can help with the decision of how to group portions of the

system into simulation modules, but this is not guaranteed to be optimal. Chen *et al.* examined three methods of automatically dividing circuits for simulations [11, 12]. They tested manual partitioning and two methods of automated partitioning (K-FM and K-AFM). K-FM is a 2-way partitioning algorithm by Fidducias-Mattheyeses [20]. K-AFM extends the K-FM model to have acyclic behavior. Both algorithms try to minimize the cut cost. They try to cut as few communication lines as possible while creating M roughly equal sized partitions. The output only indicates where the partitioning is to be made. Neither algorithm creates simulations with the partitions.

Chen *et al.* found that neither K-FM or K-AFM methods were as effective as a hand-partitioning of the circuit for simulation. The test circuits used in the experiments ranged from 6,000 gates to 64,000 gates. These are relatively small designs compared to what is used today. Simulations of modern systems require the simulation of 1,000,000+ gates and software modules.

Even with the small designs used by Chen *et al.*, they found that their least effective partitioning tool could result in a speed up. The amount of speed up found was dependent on the structure of the circuit, and the partitioning method. Simulations of some circuits could only exploit the physical parallelism through manual partitioning or optimistic simulation[8]. No circuit found less than 3 times speed up across 16 computers. Many found their peak speed up, however, across 4 or 8 computers. This is likely due to the small size and complexity of circuits used in the experiments.

Some a priori knowledge can help with the partitioning. There are often areas of a design that require data to be processed at the same time in different modules. Obviously, these represent excellent modules to run on different computational nodes. It has also been shown that some areas of a design are particularly excellent candidates for separation onto distinct compute nodes, specifically hardware modules that demonstrate symmetry [33].

While these automated partitioning techniques can provide an improvement over manual partitioning, or no partitioning, they also create either a delay before the

[8]Optimistic simulations allow portions of the simulation to get ahead of the rest of the simulation, they may need to resimulate if thier inputs change for results that have already been produced.

simulation can be run, or a delay during simulation startup while the partitioning is determined. A delay before the simulation can be run is created by having to run a tool to determine the partitioning and making the partition changes manually. This can be avoided by implementing these techniques in a distributed simulator. However, the cost of the technique is now paid at the start of every simulation.

2.3.2 Partitioning for System Configuration

Partitioning for hardware/software division is done many times over the course of the design cycle as trade-offs between speed, cost and timing are worked out. In this case, the more simulations that the designers have time to complete, the hope is that the closer the final design will be to the optimal partitioning. As more modules are moved into hardware, the cost of the system goes up, but sometimes a hardware module is the only solution which meets the hard real-time requirements. A balance between cost and meeting the timing requirements is worked out over the course of many simulations.

Over the course of the design, several decisions must be made:

- Which algorithm to use for each of the modules?
- How to implement the chosen algorithms?
- How will the implementation be translated to its final form?

An example of these decisions that happen in a hardware module is deciding if the use of adders and multipliers would be more efficient, in both cost and time, or whether the design should be implemented with adders and shift registers. Figure 2.3.2 shows an example of this decision process. This example shows how a single component decision could need seven simulations to determine its final configuration. It is important to note that the simulations would need to be done in combination with other components and their multiple configuration options.

As systems increase in complexity, so does the number of simulations that are needed to find a optimal partitioning. The design process of partitioning modules

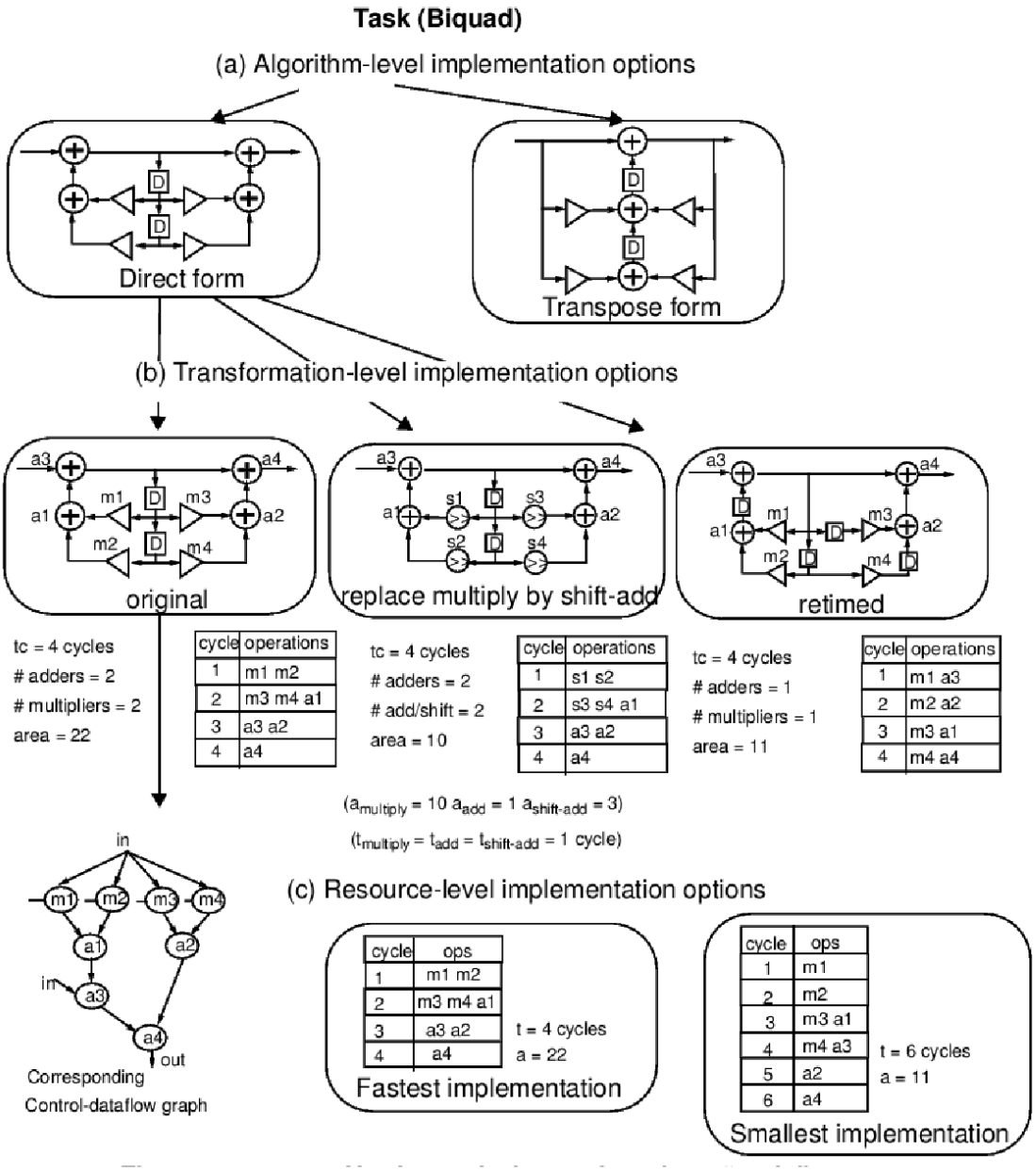


Figure 2.4: Partitioning steps of a single element of a Co-design, from Kalavade and Lee's partitioning paper [30]

into hardware and software says that if there are N modules to determine the final configuration, and each module has only to decide between hardware and software, then the number of simulations required to exhaustively search for the optimal configuration is greater than 2^N. This would imply that that there is only one implementation option for each of the hardware and software configurations. In general this is M^N where M is the fewest number of options for implementing a module. Thus the general form is closer to the number of simulations required for an optimal partitioning. Modern systems have large numbers of modules, which makes finding an optimal partitioning require an impractical number of simulations. It is important to note that two hardware implementations of the same algorithm can have different simulation results, and thus the base of 2 is a lower bound. A priori knowledge and trend analysis will reduce this number significantly but the problem of finding an optimal partitioning is still NP-Hard [30].

Partitioning into hardware and software modules is part of the design process. This thesis does not change the design process, but demonstrates a tool to reduce the time spent in simulations. This allows for two things: more simulations to be run, and/or less design cycle time spent on simulation.

2.4 Parallel Processing

Parallel processing can often complete a a task quicker by attacking it with multiple processors. Both CPU (working time) and I/O (waiting for resources) bound tasks can benefit from parallel processing if it is done right. Breaking up a problem into tasks or sections for parallel processing can be done two basic ways. The task can be broken into sub-tasks and each task handled by a different unit. Alternately, the data can be broken into groups and have each compute node do all of the work for that chunk of data. These techniques are called code and data partitioning [34].

Complex tasks often require many things to be done before the next step can be accomplished. As an example, think of the steps to painting a room:

1. Wash the walls

2. Tape/cover areas that need protection
3. Paint the trim/edges
4. Paint the main areas
5. Clean up

This is an example of a set of concurrent tasks that can be completed faster if done in parallel. Two people working on this job will finish it faster. The two people could each work on half of the room, or they could each work on the next step as the other person finishes the previous step in another portion of the room. These are data partitioning (where all work is split between the workers), while the latter would be code partitioning (where the workers specialize in tasks).

There are limits to the advantage of parallel processing [24]. Amdahl's law clearly states the speed up is limited by three factors: 1) the amount of sequential processing, 2)the amount of parallel processing, and 3) the amount of resources. Ahmdal's law can be written as $SpeedUp = (T_{sequential} + T_{parallel})/(T_{sequential} + (T_{parallel}/M))$. This tells us that adding resources will only speed up the sections of the simulation that have inherent parallelism. Kiovisto did an excellent analysis of the implications of Amdahl's law with respect to how much speedup can be found through exploiting parallelism for given ratios of sequential to parallel sections of an application [32]. He showed that the speed up factor in a set of parallel tasks would be reduced if there was sequential processing that took more than 1% of total execution time and that the reduction would be proportional to the amount of sequential processing over 1%

There are several major techniques that aid in programming parallel processing applications. Each corresponds to a different method of connecting the compute nodes[9]. The techniques are message passing interface (or MPI), inter-process communications (or IPC) and specialized communications routing software [44]. MPI is used in low latency inter-computer applications like cluster computing, while IPC is typically used in multiprocessor environments, such as SMP, where communication does not leave the computer. The specialized communications software referred to

[9]A compute node is a single location for processing the data. A computer may have more than one compute node.

in this thesis is used in grid computing systems that allow an application to share computing resources over large areas and may even support compute nodes being added or dropped.

Inter-process communication (IPC) is a general term that covers any type of communication from one distinct process to another. The term IPC in this thesis refers to communication between processes within a machine. It can be used to synchronize or exchange data across separate processes. It can be used on SMP machines for communication, but specialized communications protocols have also been developed to aid with developing applications for parallel computing [9].

A common communication and synchronization protocol that is used in parallel programming is called MPI, or Message Passing Interface [8]. MPI is a standard that sets out methods of communicating and commands to use to facilitate it. Various vendors supply MPI libraries for developers to use when doing parallel programming. The developer is still required to decide how to design the application, but the communication between the parts is simplified.

Sometimes, custom communications software is needed to address special issues for a parallel computing task. Typically, this happens when the machines involved do not have constant available resources, in that compute nodes may removed from or added to the pool of nodes. Custom software handles routing information to its destination and restarting elements of the task that have failed due to nodes leaving the system [28].

There are three major classes of parallel processing machines: multiprocessors, clusters, and grids. Each of these have increasing communication overhead, but higher flexibility. Multiprocessors offer the lowest communications delay as the compute nodes are tightly coupled. Cluster computers require low communications latency and high bandwidth between several closely related machines. Grid computers often use the Internet for the interconnection between machines. These machines vary in computing power, memory and processor type.

Multiprocessors are computers with more than one CPU or core. There are three types of multiprocessors in use today. SMP machines are common and have shared

memory banks. DMPs have distributed memory banks (or separate memory for each processor), and are mostly used in specialized high performance computing. There are also machines called vector processors which use multiple data paths and memory banks, but the processors all do the same task [25]. Vector processors and DMPs will not be covered in the scope of this thesis.

SMP machines can have a large number of processors, but the cost and complexity of the machines rises quickly as more processors are added. Alternatives to SMP machines are found in clustering, grid computing and cloud computing.

Cluster computing uses many machines connected together on a dedicated high-speed network. The machines do not need to be identical. MPI or another communications protocol is required for the processes to communicate across the machines. The advantage of clusters is a higher work/cost ratio at the expense of a higher communications cost.

Grid computing is related to cluster computing, but there are two major differences. The primary difference is that the machines on a grid are not on a dedicated network. Secondly, the machines may or may not be dedicated to the parallel processing tasks. In grid computing, machines can be added or dropped from the network of machines solving the problem at hand. In grid computing, specialized communications and control systems are used to migrate the tasks to available resources. This is done while keeping the actual location of a running process invisible to the program and programs that interact with it [21]. There are some grids that have been deployed that span the globe [41] [17].

MOSIX is an interesting type of grid computing [6]. MOSIX takes the view that the grid should be transparent to the user and designer. From the designer's point of view, a MOSIX cluster can be treated as an SMP machine[10]. IPC can, therefore, be used to communicate between processes. This allows testing of SMP parallel processing programs on inexpensive hardware. MOSIX takes care of all of the load balancing, data transmission and communication between nodes.

[10]See http://www-106.ibm.com/developerworks/edu/l-dw-linuxmosix-i.html

2.5 Distributed Simulations

As stated previously, distributing simulations over many compute nodes may reduce the simulation run time. In a discrete event simulation, there are often many tasks that need to be done before the clock can be advanced. If any of these can be done at the same time on separate nodes, the simulation time should be lower as long as the communications delay does not negate the speedup. If the time to pass the data between nodes is higher than the time to complete all tasks on a single processor, then the centralized simulation would be faster.

There are three major categories of distributed simulation: code division, and synchronous and asynchronous simulators with work division. Code division can be used for distributed simulations by examining the simulation and breaking the task of simulation into modules and pipelining the execution. Asynchronous and synchronous distributed simulations look for tasks that are being simulated that can be done in parallel. Synchronous simulations require that the simulation time in all modules be the same. Asynchronous simulations do not have this requirement. Time skew in an asynchronous simulation can cause problems, this is discussed later in this section. In general, a synchronous distributed simulation will be slower than the same simulation done as an asynchronous simulation. However, as the synchronous simulation does not have to deal with time skew, it is easier to create and validate [21].

Code division can be used for distributed simulation, but it does not provide the scalability that data division offers. Code division separates areas of the simulation that can be done at the same time. Examples of this are simulations where the data is processed multiple times. Each of the processes can be scheduled onto a separate processor. As shown in Figure 2.5, Luksch [34] shows that when the simulation in his experiment is spread across three or more processors, a theoretical three times speed up could be found. Due to poor communications design and implementation in his experiment, no speed up was found. The time spent synchronizing data overwhelmed the speed up from the parallel processing.

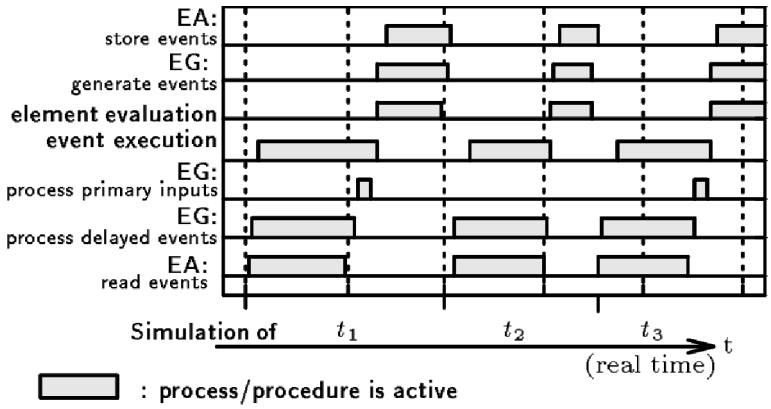


Figure 2.5: Code partitioning example [34]

Synchronous simulations do not process data until the unified clock reaches the time where it would be processed. This means that if one module takes longer to process its data than the others, it becomes a bottleneck. A good partitioning can reduce this, but not entirely. Synchronous simulations are not as fast as asynchronous simulations, but they are easier to write and to modify without introducing errors into the simulation as there is no need for roll-back or anti-messages.

The main difference between synchronous and asynchronous distributed simulations is the unified clock. If one module of an asynchronous simulator can go faster, it will continue to simulate until input is not available. When it needs input, it has to wait. This leads to portions of the simulation being ahead, and possibly working with data that will change before the answer is needed.

Asynchronous simulations can have modules that provide erroneous data. The possibility of a module being ahead so far that it is working with the wrong data has been addressed in two different ways. The solutions are termed optimistic and conservative. Optimistic simulators assume that the data that a module currently has, will not change before the output is required [18, 52]. Conservative simulators only produce output when a module's data cannot change before its output is needed [31].

In an optimistic simulator, when the input for a module changes after the output is produced, the simulator must send out *anti-messages*, or rollback to a previous

state. This ensures that any modules that used possibly wrong data can use the new data and reissue any messages. While the anti-messages and rollbacks take time, if there are only a few rollbacks, then the optimistic simulations are faster than the conservative simulation.

Conservative simulators do not have to deal with roll backs or anti-messages [3]. Conservative simulators only prepare output if it can be guaranteed that the input will not change before the output is needed. This causes conservative simulators to run slower than optimistic simulators, but faster than synchronous simulators.

There is a hierarchy of possible speed up in distributed simulations. As mentioned earlier, this ordering is as follows, from slowest to fastest:

- Monolithic simulation, or non-distributed simulation.
- Synchronous distributed simulation,
- Asynchronous conservative distributed simulation, and
- Asynchronous optimistic distributed simulation.

The possible speed up is also inversely proportional to the complexity of creating a simulator with that technique. The monolithic simulators are the easiest to create, but they are also the slowest.

Some projects that have used synchronous and asynchronous optimistic simulation in system design[11] are found in table 2.5.These projects show that the simulations can be distributed in each of these ways. Both of the asynchronous simulator projects predate research into Co-Design. While SystemC MPI and Distributed SystemC are proof of concept projects. Parallel SystemC is from a 2009 paper [19] and is a parallel work to this thesis.

Time Warp is an optimistic distributed simulation framework. The framework uses plug-ins to enable different simulations. It has been extended in the SAVANT project to simulated VHDL designs. The sub-project TyVis produced the plug-ins and translation engine for VHDL into a form that could be simulated by the TyVis

[11]TyVis predates Co-design, thus the generic term of system design has been used

Table 2.1: Reviewed Distributed Simulators

Simulator	Type
TyVis	Asynchronous/Optimistic
Cpoker	Asynchronous/Conservative
SystemC MPI	Synchronous
Distributed SystemC	Synchronous
Parallel SystemC	Synchronous

kernel. The translation engine created extended C++ code that could be simulated in Time Warp or any other optimistic simulation framework [52].

Cpoker [3] was a simulator designed to explore the overhead and communications cost as well as the run time advantages of an asynchronous conservative hardware simulator. They found that the run time could be reduced, but not linearly as CPU resource were added. However, their research focused on the costs of maintaining the state of the simulator and the communications cost.

The SystemC MPI project demonstrated that MPI calls could be included in SystemC Modules[12]. The MPI calls can than be used to communicate between SystemC simulators or other systems. These calls can be added to any module, but they have to be explicitly added and they do not follow the SystemC model.

Distributed SystemC offers explicit connections between simulators [49]. It allows developers to explicitly create multiple separate simulators, and control what is in each of them. This forces the simulation partitioning problem onto the developer. The developer needs to choose which modules belong to each simulator. The interaction between modules is modeled as SystemC communications channels. Signals are sent over TCP/IP. This makes the connection semantics integrate naturally into the design, but it also means that the systems that the simulators will be run on is decided at compile time due to the configuration of the communications channels.

In contrast, during the same period as this thesis, Ezudheen *et al.* from the National Institute of Technology Calicut developed a parallel SystemC simulator

[12]see http://www5a.biglobe.ne.jp/ hamabe/index.html

[19]. This simulator creates groups of modules and associates them with a CPU at run time based on which partitioning algorithm was in use for each experiment. They examined manual partitioning, two work sharing algorithms, and a work stealing algorithm. The work sharing algorithms collected the activated modules into chunks to be allocated explicitly to processors. This was done either as a division of the number of modules by the resources, or by measuring the load of each CPU and allocating more modules to the less loaded resources. In contrast the work stealing algorithm allows an underutilized CPU to steal modules that are waiting on another CPU.

These four partitioning algorithms were studied under three conditions: 1) as the number of CPUs changed, 2) as the number of modules changed, and 3) as the run time for a module changed. All of these were measured against the run time of the simulation and compared with the unmodified SystemC simulator. A comparison of their simulator to the simulator proposed in this thesis can be found in section 4.3.

Some simulations cannot be helped by distribution. Those that can have their simulation time reduced, exhibit either parallelism in the processing, or data, and/or require multiple independent processing passes of the data. Hardware simulations exhibit all of the requirements. The method of distributing the simulation can also affect the speed up. A synchronous simulator is the next step to speeding up Co-Simulation. The complexity of creating the synchronous simulator is well within reach, while still offering some speed up.

2.6 Summary

This chapter has covered background information that is necessary to understand the motivations and experimental system design of this thesis. The design path and its changes due to the development of Co-Design as an industrial design technique were reviewed. Additionally the general types of simulations were reviewed, as was the differences between simulation and Co-Simulation. The general theory of the partitioning problem was presented and related to this work. Parallel processing systems,

their advantages and disadvantages were presented and briefly discussed. Finally a review of the distributed simulator techniques and projects that relate to this thesis was reviewed. Now that the problems in the design path have been explored, a solution to the run time issue caused by the need to test hardware/software partitions can be explored. This proposed solution used the simulator built into SystemC as a base for a synchronized distributed simulator.

Chapter 3

Prototype and experiment design

This chapter describes the design of PDQScS (Parallel Distributed Quick SystemC Simulator), the evaluation techniques and the performance experiment design. PDQScS is a prototype synchronous distributed simulator derived from the simulator included with SystemC. PDQScS is compared to the SystemC simulator to verify correctness of the computational capabilities of the new system as well as run time performance characteristics.

PDQScS is a modification of the existing multi-threaded co-simulation tool/language SystemC. As a language, SystemC is an HDL based on C++, but it also has a reference simulator for the language. PDQScS retains the simulator portion of SystemC, but modifies it so that it runs as a multi-process application. These processes are then distributed to separate CPU resources as available.

The design goals of the prototype for this thesis are the following:

1. a system that does not increase work for the designer of a Co-Design system,
2. scalable in both Co-Design system design size and number of CPU resources, and
3. allow existing system designs to be simulated without modification.

The scaling criteria requires that the PDQScS system be tested with varying numbers of available CPU resources.

SystemC is an open source initiative to create a combined hardware description and programming language that is well documented and readily available [38]. SystemC uses the C/C++ programming environment to describe the hardware and software modules, their testbenches, communications and the simulation system. As

a side benefit, SystemC is currently being used in industry[1]. Many industry designs were evaluated for the final experiments, but none met the needs of the experiment. These designs are discussed later in section 4.1. An artificial design with controllable work load will be used instead. This design is covered in Section 3.3.1.

There are two main aspects to this solution: the modification of the existing SystemC simulator, and the distribution of the work to separate compute nodes. The distribution is invisible to the user/developer of PDQScS. A user of an SMP machine or MOSIX cluster has their processes allocated to available CPU resources as needed by the operating system. PDQScS is written for the SMP architecture, and thus, there is no need to rewrite the system for use under MOSIX. For this thesis, this allows concentration on the simulator side of the problem.

This chapter has five sections. Section 3.1 is a overview of some supporting works by others. Section 3.2 contains the details of the modifications that need to be done to the SystemC simulator to create a synchronous distributed simulator. Section 3.3 is a section detailing the metrics, factors and the techniques for acquiring and evaluating them. It also provides a discussion of the resources that were used in the experiments. Section 3.4 contains a discussion of the experimental platforms and DUSs that were considered and eliminated. Section 3.5 is summary of the chapter.

3.1 Supporting Evidence

There are two major sources of evidence identified before the start of the project's implementation phase that indicated that it would be successful: Other studies of a related nature and code walk-though of the SystemC simulator. In 1993, Luksch produced a distributed version of a hardware simulator. His study showed that there is un-tapped parallelism in hardware simulations. SystemC is not an HDL but an SDL and infrastructure, complete with basic data types for hardware and software modules and a simulator. This simulator functions in a way that partitions the

[1]Design Automation Conference 2009 hosted the 11th annual North American SystemC Users Group Meeting with presentations from Mentor Graphics, Synopsis and others.

design under test into modules which it runs separately in threads, but only allows one thread to run at a time.

The modification to SystemC in this thesis work is the same type of modification that Luksch made to a hardware simulator in 1993 [34]. There are, however, a few differences. In the Luksch study, a synchronous simulator was not produced or studied. They studied two asynchronous simulators based Time Warp for DMPs. As well the scope of the study was limited to hardware simulation, not Co-Simulation. The Luksch study does indicate that it should be possible to distribute a Co-Simulator and achieve a reduction in the runtime as a portion of a Co-Simulation is hardware simulation.

Studies of the source code for the SystemC simulator and the documentation manuals that accompanies it indicated that the simulator would be easily modified. Code traces and debugging walk throughs of the SystemC simulator narrowed the locations that needed to be edited to create PDQScS. The scheduler and module initialization routines needed some modifications, but none that introduced errors into the simulation. This was verified by comparing the output of simulations before and after modification.

The SystemC simulator is programmed in C++ and integrated into the testbench framework that comes with SystemC. The object oriented design of the simulator made it possible to modify the basic objects and have the changes reflected in all of the inheriting modules. There needed to be some study of how many of the SystemC base objects, like *sc_object* and *sc_module*, have their communications routines overridden by their descendant classes for a general case solution to be produced. For the experimental system, only the SystemC communication modules required by the design under test were modified to use shared memory.

3.2 Modifications to SystemC

SystemC is an ideal candidate for modification to a multi-process architecture. SystemC already has the following: a multi threaded simulator, a central control mech-

anism, modular based units to describe portions of the system and an open source license. The first two reasons, an existing simulator and central control, mean that once these elements are understood, they can be modified to be a multi-process instead of a multi-threaded application. The modularity makes it possible to distribute the work with minimal work. Tho open source license provides access to the source code.

The simulator built into SystemC is a multi-threaded simulator with a central scheduler that controls when each module/thread runs. The scheduler uses a multi-pass system to ensure that all modules that need to run during the current time quantum have been activated. Once activated, a module runs until it returns control to the scheduler. If several modules are required to run during a quantum, then those modules are activated sequentially. The first pass of the scheduler activates all the modules that had received new data in a *previous* quantum, but were required to process that data in the current quantum. During the next pass, the scheduler activates all the modules that are activated by modules in the first pass, but must work in *the current* quantum. This second pass repeats until there are no more modules to activate in this quantum, then the first pass is done again at the next quantum.

Since the simulator built into SystemC uses a central scheduler, it makes for an excellent insertion point for a central server to control a synchronized distributed simulator. The multi-pass technique is retained, but instead of waiting for each *module* to return, the scheduler waits at the end of each *pass*. This synchronization achieves a lower speed up factor than an asynchronous simulation, but is easier to implement.

The SystemC scheduler documentation claims that the order of module activation is non-deterministic. Starting all modules at the same time should mimic the real-world device such that the order of execution does not matter. The simulator would not need to wait for modules to complete within a pass. Unfortunately, code traces showed that the implementation did not follow the documentation. The simulator has the same effect as one which is non-deterministic in all but one case. The

documentation and code traces show that if a user created a design in which multiple modules changed the same variable to different values within the same simulation quantum, the answer stored would be the last value to be written. In the SystemC simulator, each module would each write to the data during its turn, but the order of the turns is intended to be non-deterministic in the actual system. In the SystemC simulation, however, it is based on the order that they emitted their write request signal, which in turn is based on their creation order in the simulation setup phase. In a distributed simulator, the first module to finish its work and take the write lock would write first, and thus have its data over written by the last module to change the data.

The current SystemC simulator uses a multi-threaded design that associates the modules with threads. These threads are created at module initialization time. The thread creation can be changed into process forking. Inter-thread communication can then be changed to use IPC with shared memory and semaphores to control access. The basic structure is retained; only the execution semantics are modified. These changes do not introduce side effects, but they do introduce some constraints on the types of systems that can be simulated. These constraints are discussed later in this section.

The easiest form of IPC is to use shared memory for any information that might need to be shared between processes. This allows each process to send and receive information from all the processes. Only processes that are watching the shared memory location will note the communication, but potentially all processes have access to the information. Using shared memory does require a method to control concurrent access so that multiple processes don't write to the data locations at the same time. Semaphores are a common control structure for this purpose.

The use of shared memory for the IPC channel places some constraints on the systems that can be simulated. The primary constraint is that all cross-module objects must have their shared memory fully-allocated during module loading and before initialization. This ensures that the shared memory is initialized in all processes by allocating the memory before the child processes are forked. The second constraint

is that only SystemC data types can be used for inter module communication as these have been modified to use shared memory.

The modules each specify the interface and logic for a single part/item in a design. A module can contain logic, communication channels, or other modules. Since the interface of every module defines how the simulator must connect the modules together, the modules make an excellent place to divide the simulation into parts for distribution. This distribution is sub-optimal, but optimizing the distribution is outside the scope of this thesis.

3.3 Research Methodology

The main goal of this thesis is to show that a SystemC distributed simulator is

1. possible
2. faster
3. does not introduce errors
4. scalable.

The experiments are designed to capture data to prove or disprove these points. The first and third items are items that will be satisfied by the successful creation of the simulator. The third point simply requires that the output of the simulator match the output of the original SystemC simulator on designs that do not have inherent non-determinism in their physical behaviour. This leaves the second and fourth items from the list as the only items which are variable and measurable.

Run time is the main metric for the experiments, while the amount and type of distribution will be experimental factors. Run times of the simulations will be compared against a baseline simulation run time. With many different machines being used during the experiments, direct comparison of run time between experiments might not be valid. A normalization factor will first have to be determined.

Normalization of the run time metric requires a baseline value. The baseline is calculated by using the SystemC simulator runtime for each computer using only a single processor. Once each computer configuration has a baseline value, the run time

for each subsequent experiment can be normalized to that baseline. The baseline is then compared against the PDQScS run time under the same conditions.

Comparing the baseline for the single processor experiments to the PDQScS results potentially shows us two different sources of delay introduced by the modifications: setup delay and communications delay. PDQScS may have extra setup time required by the creation of the processes and the shared memory. The shared memory and the locking it requires may have introduced delay in the communications link.

The setup delay can be measured by running the experiments with no extra work load and the simulation control loop set to 0. The simulation control loop set to 0 will short-circuit the simulation and allow the simulation loop to run only once. This creates a run of the simulator that is all setup and no simulation. The difference in the baseline runtime and the PDQScS runtime for this experiment is the introduced setup delay. In this experiment, there is no advantage for the multiprocessor machines, and the data point can be compared on all machines.

To determine the communications delay, the baseline and the PDQScS from the single processor experiments can be compared. The difference between the two will be the communications delay, as no extra resources have been added. The experiments where the same workload was created with different values for the command-line variables show the greatest insight into this delay. If the main control loop has more iterations for the same workload, then it will require more iterations through the communications sections of the simulation. Any introduced communication delay will be amplified by the increased runs through the communications sections, and measured by comparing identical workloads on a single machine.

In order to prove scalability, machines with different numbers of processors are needed. To accommodate this, the following machines were used:

- a hyper-threaded machine with two virtual processors
- a single processor dual core processor SMP machine (2 effective CPUs)
- a two processor dual core SMP machine (4 effective CPUs)
- a two processor quad core SMP machine (8 effective CPUs)

These machines allow for a view of how the simulation run time is affected by the number of available processors.

3.3.1 Design Under Simulation

Testing PDQScS requires a system with known attributes to be simulated. The simulated system consists of a Design Under Simulation (DUS) and a testbench. The Design Under Simulation is the component, device or system that the user wishes to simulate. The testbench exists to provide the DUS with its external structures, support and provides methods to measure and test the DUS.

Four existing designs were tested as possible DUSs. These designs were used to validate PDQScS, but were deemed to be insufficient for the performance experiments as their run times were too short. These designs are discussed further in section 4.1.

A simple design that only used SystemC primitives was needed for the final experiment. A simple 8-bit order-inverter was created that met the limitations of the modifications to SystemC. An overview of the design is shown in Figure 3.1. The design works as follows:

- The Input to the design is a single 8-bit line; this line is connected in parallel to four copies of module A.
- Module A selects the appropriate 2 bits from the input line and reverses their order on the output line.
- The output lines from the A modules are connected to the inputs of the B modules.
- The B modules are simple connectors. Module B takes two 2 bit inputs and combines them into a 4 bit output. Module B2 does this with two 4 bit inputs into a 8 bit output.
- The Monitor Out line is a testing lead that enables checking of intermediate outputs.

The DUS is embedded in a testbench that provides a framework for testing the circuit and checking the results. As can be seen from Figure 3.2, there are two copies of the DUS connected to the monitor module. The stimulus module combined with

the clock module produce a monotonically increasing integer. The DUSs use the lowest 8 bits of this integer as their input.

The two copies of the DUS enables the monitor process to detect if the outputs of two copies of the DUS have diverged. A divergence would indicate that the simulator was not creating reliable results. The results of the two DUSs should always be the same given the same input.

The original design under simulation resulted in a very fast simulation runtime. This design was extended to provide an artificial workload. Module A (the 2 bit order inverter) was extended to also run floating point and integer math calculations in a loop as part of its work when activated. Additionally, the testbench control loop was extended to run over an arbitrary length of time. Thus, there is an inner loop (Module A) and an outer loop (the testbench control loop) that combine to vary amount of work that must be done before the simulation completes.

Two command line parameters were added to regulate the loops, and thus the workload. The first command line variable controls the inner loop that regulates how many times the artificial work load will be run during a module A cycle. The second variable controls the main simulation control loop; it limits the maximum value that the stimulus module counts to before shutting down the simulation. If it was only necessary to control the workload, only one variable would be needed. The two variables combine to create a workload, but they combine to create workloads that while they have the same load, they have different communication and distribution characteristics. When the outer loop is set to 5 and the inner loop is set to 2, the effective workload is 10 times that of the base experiment. However, there are 5 times as many cycles through the simulation control loop, and 5 times more overhead from the inter-module communication. While setting the inner loop to 5 and the outer loop to 2 also has a workload of 10 times the base, it only has twice the inter-module communication overhead. Thus, two control variables allow for a single experiment series to explore how and why the system may suffer from excessive overhead due to the modifications.

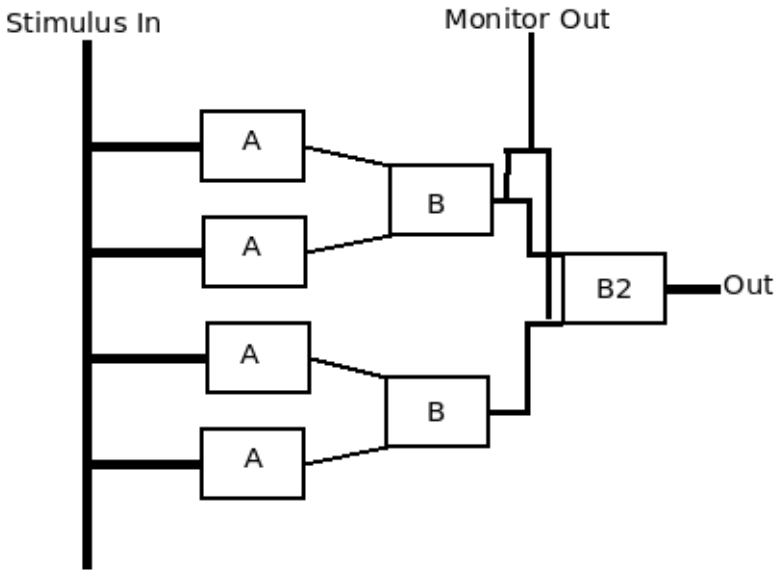


Figure 3.1: Design under simulation

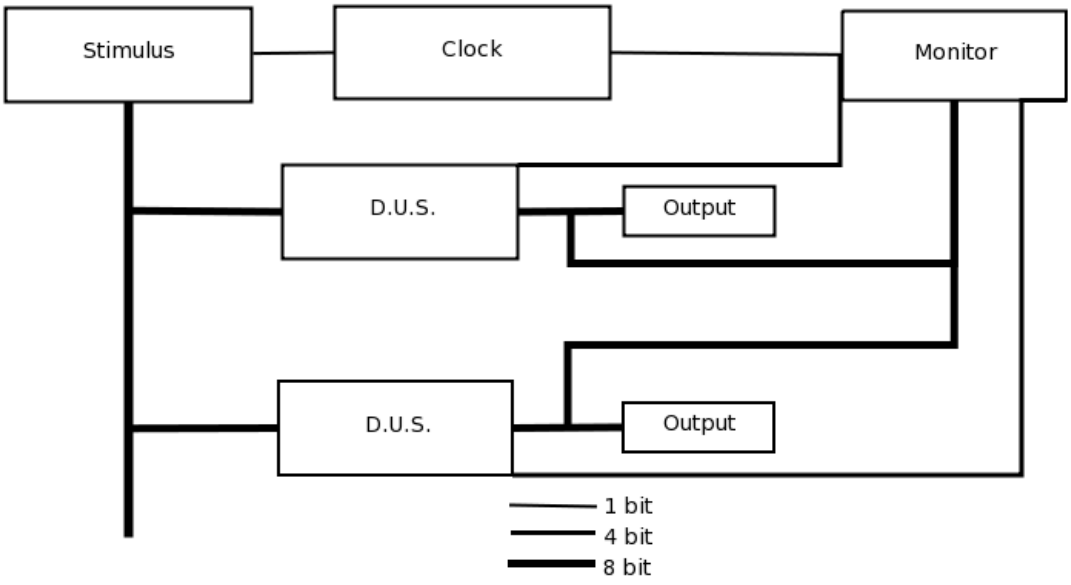


Figure 3.2: Testbench setup

3.4 Selection of experimental systems

When the research for this thesis started in 2004, SMP and other forms of multi-processor machines were expensive, and rare outside of servers. MOSIX was looked at as a way to cheaply create computing clusters with existing desktop or laptop resources. This is no longer the case. As of second quarter 2009, Intel[2] and AMD[3] sell mostly multi-core processors. Thus any new desktop, laptop, compute node or server will have simultaneous access to more than one processing unit. While this strengthens the utility of the work in this thesis, it reduces the utility of MOSIX. Combined with the unstable nature of the MOSIX kernels that were used in the preliminary studies for this thesis, and the fact that the OPEN MOSIX group has ceased development [5], MOSIX experiments for this thesis have been dropped for the performance results.

Currently, AMD and Intel are transitioning their offerings to be mostly multi-core processors. As of January 2009, Intel sold only four models of its oldest processor as a single core processor. At the same time, AMD sold only 2 non multi-core CPU models of its oldest processor as single core processor. An executive from SUN Microsystems predicted in 2005 that in the future multi-core processors will be 100% of the market [36]. As this trend is well on its way, the advantages of MOSIX are waning.

Moshe Bar officially closed the openMOSIX project on March 1st, 2008. In July 2007, Moshe Bar said "The increasing power and availability of low cost multi-core processors is rapidly making single-system image (SSI) Clustering less of a factor in computing."[4] This led directly to the end of life for the project, and though the source code is available, no developers have remained involved in the project. The MOSIX project, a proprietary software project related to openMOSIX still exists, but does not garner the attention that it once did.

[2]The AMD product list can be found at http://www.amd.com/us-en/Processors/ProductInformation

[3]The Intel product tree is found at http://www.intel.com/products/processor/index.html

[4]Moshe Bar announced this on www.sourceforge.net/openmosix

The closing of the openMOSIX project by itself is not enough to cause MOSIX experiments to be dropped from this thesis. There have been additional complications, however, that have also led to this decision. The main complication was the requirement to patch MOSIX with a shared memory migration (MIGSHMEM) patch to allow processes that used shared memory to be used with MOSIX[5]. Unfortunately the patch did not support the Linux 2.6 versions of MOSIX, and was not stable. Many times it would cause the cluster to freeze or to stop sharing jobs. Any results produced with this patch would not have been reliable or reproducible. Further discussion of the patch is found in section 4.2.1.

3.5 Summary

This chapter has outlined the parallels and differences with prior work and examined the solution that this thesis proposes. Additionally, this chapter has examined the experimental framework that was used for the investigations and has explained how the DUS was designed. Finally, this chapter provided a review of some proposed DUSs and the limitations of PDQScS eliminated them from use for the primary investigation of this thesis.

PDQScS has been designed to take advantage of the non-determinism in the SystemC simulator and allow for all of the modules that have been activated by the clock pulse or by a delta cycle to run simultaneously. This offloads the scheduling of the work to the kernel scheduler. The designer does not need to guess as to how to group the modules to assign them to each compute resource or rely on the simulator to group them correctly.

[5]The package can be found at http://howto.krisbuytaert.be/openMosix-Migshm-rpm

Chapter 4

Results

Evidence presented in this chapter shows that the distribution of some classes of Co-Simulation can reduce the simulation time. In the experiments, simulation time reduced linearly with the number of processors used with PDQScS. Furthermore, the results showed that the communication between the distributed portions of the simulation did not have a significant impact on the simulation run time. These results combine to show that a simulation where there are high amounts of CPU work, and low overhead from I/O and communication, can be divided into separate processes efficiently and can be executed at a rate proportional to the number of available CPUs.

4.1 Motivating experiments

To provide evidence that the modifications proposed by this thesis would be successful, preliminary performance measurements were taken with four example designs. The following measurements were taken with the SystemC simulator: run time, CPU load, and memory usage were recorded for each run. These measurements would show if the simulations used large amounts of memory or had very high CPU loads. A high CPU load would indicate that distributing the problem properly could speed up the simulation. If the designs had instead had low CPU usage, it would have indicated that the simulations were I/O bound and unlikely to show improvement with the techniques used in this thesis. High memory usage would also indicate that additional design effort would be required for the simulator modifications to be successful, while low memory usage would suggest that the envisioned modifications

would function as intended.

The machines used for the motivating experiments are a cross section of the types of machines that could typically have underutilized resources when performing simulations. Preliminary measurements were taken on three systems. These systems are described in table 4.1. The single processor system is capable of executing multiple instructions at the same time depending on the instruction mix. Distribution could enable idle hyper-thread resources to be used. The multiple processor systems are limited to a single process due to the nature of the original SystemC, but a distributed simulator could use all available CPUs. The first system used was a single

Table 4.1: Motivating experiment machines

System	Processor	Speed	CPUs	Memory
Hyper-threaded	P4	2.8 GHz	1	512 Mb
SMP	Ultra-Sparc	10. GHz	4	16 Gb
MOSIX	P3	850 MHz	2	256 Mb

processor hyper-threaded P4 2.8 GHz machine with 512 megabytes of RAM. The second system used was a Ultra-Sparc 4 processor SMP machine running at 1.0 GHz with 16 gigabytes of RAM. The third system was a two processor MOSIX cluster consisting of 2 Pentium 3 850 MHz machines with 256 megabytes of RAM each.

There were some minor differences in the machine architectures and operating systems used in the preliminary experiments. The P4 machine and MOSIX cluster machines both ran Mandrake 10 Linux, while the Sparc SMP machine ran Solaris 8. The P4 machine and the MOSIX cluster use different kernel versions. The P4 machine used the 2.6.8 kernel, while the machines in the MOSIX cluster use the 2.4.21 kernel. This is due to the lack of availability of the kernel modifications for MOSIX in the newer kernels.

The systems simulated in these preliminary experiments are small component systems that could be created in hardware or software. These simulation definitions represent devices of fewer than 2000 lines of SystemC code and fewer than 10,000 gates. These devices do not take long to simulate, but are used to judge the CPU

and memory load that these simple device simulations place on the hardware. These components are often used as parts of larger systems. Thus, the performance of these simulations is representative of a portion of the simulation of larger systems. If the simulations show that some of the simulations are CPU bound, then this indicates that the larger simulations may also have portions that are also CPU bound.

Four sample devices were simulated:

- an RSA encoder/decoder[1]
- a pipe
- a packet switch (called pkt)
- a simple FIFO producer/consumer model(called perf).

In the case of perf, this simulation is the type of exploration of division between hardware and software commonly done in Co-Design. The producer creates a consumable for which one of a number of consumers waits. When a consumer obtains an item, it then performs the required operations and when finished, waits for another item. This design does not implement a specific implementation of a FIFO, but a generalized form of a FIFO. This device is commonly used in cases where splitting the data into units of work for multiple consumers is more efficient than having a single consumer do all the work. This implementation creates a token that is consumed by taking it out of the FIFO with no processing on that token by the consumer. The RSA decoder is a detailed simulation of a hardware device. RSA could also be implemented as software and either implementation could be included in a larger design. The pipe and the packet switch simulations represent components of larger designs. They are simple components that could be used many, many times in a single, larger system. Though each component is simple and quick to simulate, this computation becomes significant once there are large numbers of these components in a system.

The run times for simulations in the motivating experiments can be seen in Figure 4.1. These simulations are quite quick because of the size of the devices simulated.

[1]RSA is an encryption/decryption algorithm. R,S, and A are the initials of the creators of the algorithm

Each bar represents the mean run time for 5 runs of the experiment. All of the simulations took less than one second on all machines. Note however that none of the machines have consistent performance across all of the designs. This is due to the fact that these machines are all different processors and architectures. The SMP machine is the slowest of the machines and this fares poorly in raw CPU based tasks. This machine does have a high throughput data transfer system, and thus, it is able to do well with the pipe design. These simulations will not be useful in the final experiment. Speed up on these simulations would be very hard to detect, and the small magnitude of each run time would cause rounding errors when calculating the speed up. Overhead from setup could overwhelm the speed up of the simulator. A simulation that had an extra 500 msec of startup would be incapable of speeding up a 500 msec simulation.

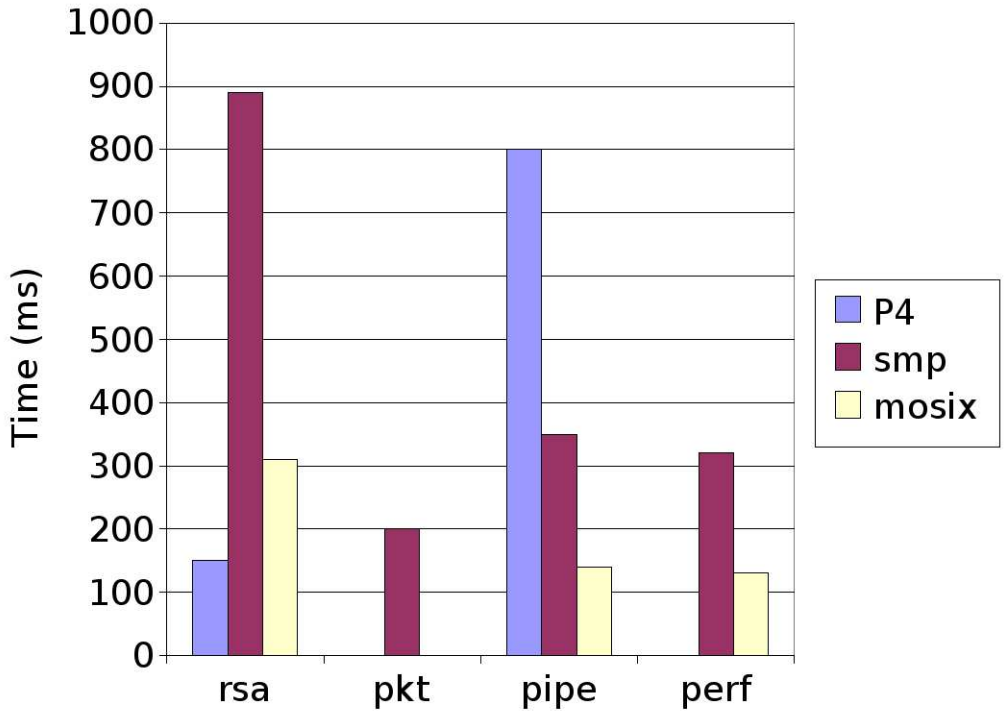


Figure 4.1: Motivating Experiment Run Times

The CPU load for the four preliminary experiments is shown in Figure 4.2. The CPU loads for the SMP experiment have been normalized to the use of a single

processor. Thus, the values reported by the SMP system during the experiments were relative to the entire 4-CPU system. This distorts the CPU load values for this machine in comparison to the other machines. The MOSIX cluster and single processor machine both used only one processor, and reported their loads as such. For the SMP machine, 100% CPU use on a single CPU is reported as 25%. Thus, the SMP loads reported in Figure 4.2 are four times the measured value.

As can be seen from the figure, the MOSIX machine consistently had the highest load. This machine was the slowest of the machines, and has some overhead of monitoring processes for migration of processes to other resources. It is interesting to note that the load on the SMP machine seems to be variable. This may be due to poor load measurement on this machine. Load measurement on the Linux-based machines was measured as part of the *time* command. This tool has different behavior under Solaris, and the GNU version of *time* was not available on the SMP machine. On the SMP machine, the *timex* command was used and returned consistent values in each of the repeated runs of each simulation. Overall, CPU load was high, indicating that distributed simulation may reduce some of the simulation time.

The RSA, pkt, pipe and FIFO system simulations used in these motivating experiments were not used for the main experiments. There were three factors in the elimination of these simulations. Firstly, the run times were too short. A speed up of even two times would have made the simulations extremely hard to measure. Secondly, while all simulations had 100% CPU use on at least one architecture, a more consistent system design that has the potential for improvement on all architectures was preferred. Finally, the system designs did not meet the restrictions of PDQScS. These systems required the use of many non-standard SystemC communications modules that would have to be modified, which was significantly more work than needed to show the benefit of the modifications. These constraints are discussed later in this section.

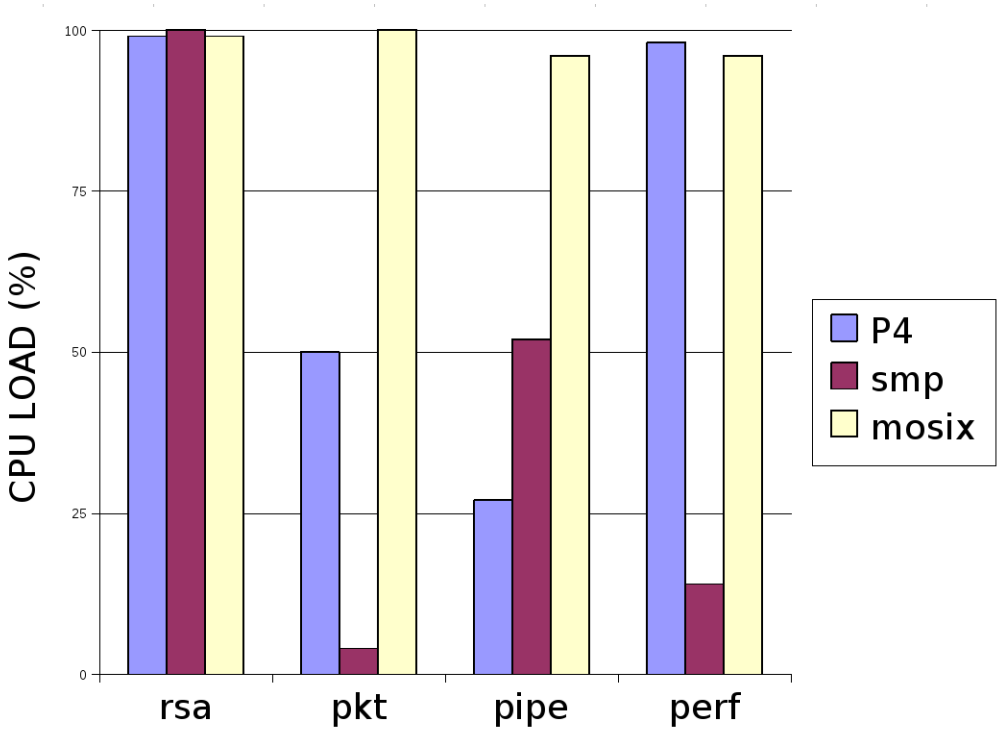


Figure 4.2: CPU Load for Motivating Experiments

4.2 PDQScS

This section outlines the development process results for PDQScS as well as proof of concept experimental evaluations. PDQScS development was not without its challenges, but they were overcome and the modifications to the SystemC simulator met the goals. To prove this, the system design from chapter 3 was simulated with 4 different machines with varying CPU resources, and compared to the SystemC simulator. There are two significant performance results that will be discussed. The analysis of the performance experiments leads to the conclusion that PDQScS can speed up some classes of simulation. A further analysis of the results shows that the communications overhead of multiple processes does not unduly hamper PDQScS.

4.2.1 Development process results

The development of PDQScS posed challenges and learning opportunities. The use of MOSIX during the design stages created challenges as did the use of Solaris. Additionally, the use of Shared Memory requires the user to be careful to prevent memory leaks and requires additional setup and tear down steps. This section will discuss these challenges that were overcome during the development.

The original design for PDQScS called for the use of MOSIX systems and Shared Memory for IPC. An unforeseen incompatibility in the design of PDQScS was that MOSIX does not natively support Shared Memory. A patch does exist for MOSIX on the Linux 2.4 kernel to support Shared Memory [10]. The patch allows applications that use only a single piece of Shared Memory to take advantage of a MOSIX cluster. These applications had to be explicitly told that MOSIX could work with them, but the patch was able to support shared memory for the sample applications and test suite which was provided with the patch.

Once the Shared Memory patch was installed and working with the supplied test benches, attempts to incorporate it in the working prototype were set back by simulations that would fail to progress. Investigation showed that the semaphores used to control access to the Shared Memory variables were required to transition from a positive value to zero to indicate that all threads had completed (i.e. become available for access). This is inverted from the standard definition of a semaphore that uses zero as a value to indicate that no new accessors can enter. Once the use of semaphores was altered to conform to this specification, the simulations progressed properly.

Shared Memory provides a mechanism for processes to communicate without explicit message passing, but Shared Memory uses a different API from standard memory functions. These API calls are not swap-in replacements for the non-Shared Memory API calls. Shared Memory does not have an owner process that will clean up the memory and return the physical memory to the operating system when the process exits. Additionally, there are a limited number of Shared Memory pointers

available to processes for the entire OS. An initial test design of PDQScS worked well with all simulation device modules initializing their own Shared Memory. This allowed easy conversion of SystemC primitives to compatible primitives that used Shared Memory instead of the heap. Unfortunately, this did not function properly as the simulated design size grew. The machine running the simulation would run out of Shared Memory handles and then simulations no longer completed successfully.

Investigations eventually revealed the cause of the simulations failing to complete successfully. Due to the fact that small systems did not display the problem, isolating and discovering the problem took some time. Once it was discovered that only large systems exhibited the problem, the large system was instrumented to determine the differences it had from the small systems. Shared Memory allocations were logged to determine how many were in use, and how large they were. Allocated memory *size* was eliminated as a cause by allocating larger Shared Memory blocks than necessary in the smaller systems. This left the *number* of Shared memory allocations as a possible source of the problem. The solution was to initialize a large amount of memory in a single Shared Memory block during simulation initialization and write a very basic memory allocation routine to control access to the Shared Memory. These memory modifications (implemented as a new module called *jdcmem*) to PDQScS were designed to cause a simulation to exit with a known exit code if the amount of memory requested was ever larger than the amount that was pre-allocated. The size of the preallocated Shared Memory block for jdcmem is set at compile time. Increasing the size of Shared Memory block is as simple as changing the value and recompiling the PDQScS library.

The final problem that was overcome was a problem that only occurred after a few runs of PDQScS. Once the defect started expressing itself, no new simulations would return correct answers until a OS reboot. Eventually the problem was traced down to the operating system running out of semaphores. After a few runs of PDQScS, all of the operating system semaphores would be used up and the simulations would fail. This was traced down to a cleanup routine that was not running properly during simulation shutdown. Two solutions presented themselves. The first solution was to

simply cleanup the semaphore and Shared Memory handles from the command line after the simulation completed. This proved that the problem was as suspected, but would have been tedious to employ in the final implementation and testing phases. Once this was proven, the second, and preferred solution was to correct the defective clean up routine to return the system semaphores and Shared Memory handles to the operating system during exit.

These obstacles provided a significant increase in experiential knowledge of many principles and details of systems programming and software development. In particular, Standards are not always followed, and sometimes for a good reason. The Shared Memory semaphores enabled a cleaner implementation of the Shared Memory monitor in the kernel. While opposite of what is taught in classes about semaphores, only having to check the Shared Memory for dirty bits when the semaphore transitions to zero reduced the overhead of the routines. The Shared Memory obstacle showed that one should not assume that memory would be properly cleaned up once a process was done with it.

4.2.2 Performance results

The run time of the simulations was measured with the UNIX *time* command. This produced output with the total amount of real time used for the simulation, and the total CPU time. The real time of the simulations is the metric that is important. The total CPU time should remain relatively constant, as the work that is completed doesn't change: more CPUs are added, not less work completed. There will be some additional work with respect to the communications overhead. The real time should reduce as CPUs are added. Amdahl's law states that adding a second CPU will not halve the run time [24]. It should reduce the run time, but due to overhead and sections of the simulation that do not have any parallelism, a $1/N$ reduction of run time for N CPUs is not possible.

The computer systems used for these experiments were the following: 1) a single processor hyper-threaded P4 2.8 G-Hz machine with 512 megabytes of RAM, 2)a dual-core AMD64 Opteron machine with 2 gigabytes of RAM, 3) a 2 processor dual-

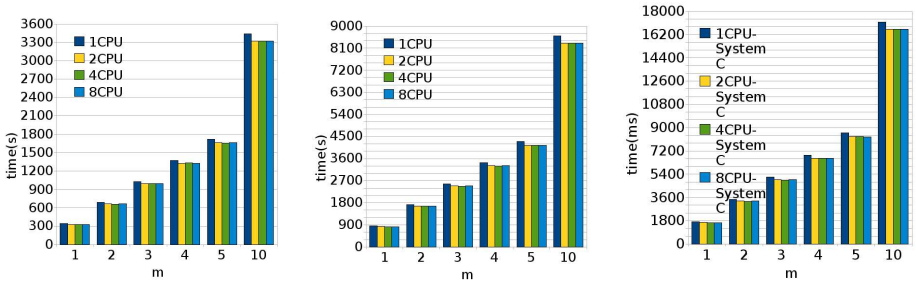
core AMD64 Opteron with 4 gigabytes of RAM, and 4) a 2 processor quad-core AMD64 Opteron with 16 gigabytes of RAM. There were some minor differences in the architectures and Linux distributions used in the experiments. The P4 and Opterons are both Linux systems, but ran different distributions. The P4 system ran Mandriva 10, while the AMD64 machines ran Centos4. For this experiment, all machines ran a variant of the Linux 2.6 kernel. Thus, the results are easier to compare and were more consistent as the tools for measurement were the same on all machines. The MOSIX and SMP machines from the preliminary experiments were dropped for these experiments. This decision was discussed earlier in Section 3.4 and has implications in section 4.2.1.

Baseline measurements

Baseline measurements were taken on all four systems used. The baseline measurements are used to calculate the speed up factor obtained with other experimental configurations. Figures 4.3(a),4.3(b), and 4.3(c) show the results of the baseline measurements for the SystemC simulator. N and M are the multipliers for the external control loop(N) and the additional work loop(M). These results have been normalized for the single CPU, two CPU and 4 CPU cases. The eight CPU machine consistently ran 6% faster than the other machines[2], and the results have been normalized for these graphs for comparison. The results show very little difference as the number of CPUs increase. This is due to the fact that the SystemC simulator can only take advantage of a single CPU.

Two conclusions can be drawn from this figure. First, it can clearly be seen that a second processor reduces the run time a minor amount, and that it does not reduce the run time linearly to problem size. Additional processors beyond the second do not also impart this advantage. This advantage is only caused by other processes on the test machine running on the extra processors, increasing the available CPU time per second available to the simulation on a dedicated processor. The machines were all dedicated to the experiments and had no other users or other user applications

[2]probably due to CPU clock rate, but this has not be quantitatively determined.



(a) 2 work units (b) 5 work units (c) 10 work units

Figure 4.3: Unmodified SystemC Simulator Run Times

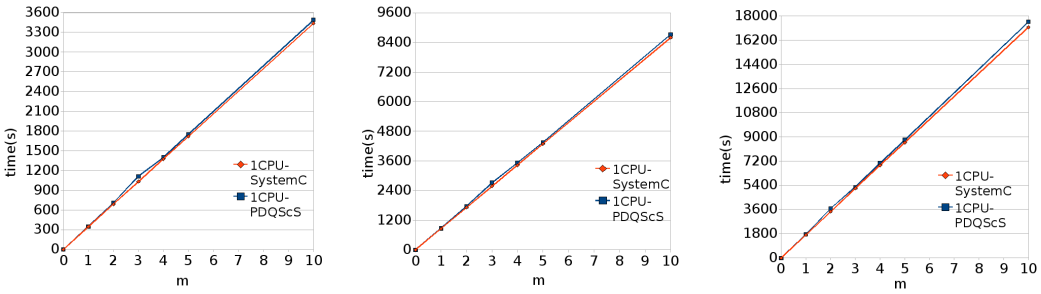
running during the experiments, but there is still some overhead of processor use by the kernel and other background OS activity. Secondly, the figure also shows that the run time does scale linearly as work is added. Therefore, run time is dependent on the total work. Recall from Chapter 3 that work can be added in two ways. Work can be added by increasing the work per simulation cycle(M), and thus not affecting the amount of communication, or work can be added by increasing the number of simulation cycles(N), and therefore increasing the amount of communication. In this baseline experiment, adding extra communication did not show a slow down in the simulation. The extra communication in the unmodified simulator is just the overhead of extra function calls, and was not expected to create any delays.

Modified Simulator Correctness Verification Results

The first experiment with the modified simulator was to verify that the new simulator performed correctly. For this activity, the output from the unmodified simulator was stored and compared with the output from the modified simulator. This step was done for every run of the modified simulator. Any errors exposed a defect in PDQScS. This happened twice due to coding errors. The shared memory handle exhaustion was the first time, and the kernel semaphore exhaustion was the second. Once these defects were identified and repaired, the PDQScS experiments were reset and started over. The final results presented in this section represent the final modified simulator that was able to reproduce the original output in every run (196 simulations were

performed for a single round of data collection[3]).These results are the average value over five rounds of each experiment.

The first experiment also provided data to show that the modified simulator does not run significantly slower than the SystemC simulator using exactly the same settings. Figures 4.4(a), 4.4(b), and 4.4(c) shows a comparison of the single CPU system results with and without modification. The graph shows that while the new simulator is slower, the slow down is in the range of 2.3% +- 1%. The simulations should have some slow down from the overhead of creating the new processes, managing the Shared Memory and interprocess communication. For the shorter simulations, this overhead is greater in percentage of total run time, but in these simulations a distributed simulator is less necessary. The overhead from creating the processes can be measured when both control loops are set to zero. This causes the simulation loop to run once with none of the simulated workload added and an immediate exit. This value was measured at 0.01 seconds on the single CPU system.



(a) N=2 (b) N=5 (c) N=10

Figure 4.4: PDQScS 1 CPU Run Times

Figure 4.5 shows what happens when the simulation control loop is set to 0. In all of these experiments, the simulation loop does not run. This case is only a comparison of the setup and tear down speeds of the two simulators. All the modified simulator results were on the range of 5 to 10 times slower than the baseline for these experiments. These simulations are excluded from the remaining results discussed as they do not represent an actual run of the simulator.

[3]N=[0,1,2,3,4,5,10]=7, M=[0,1,2,3,4,5,10]=7, 7*7*4 Machines=196 Simulations

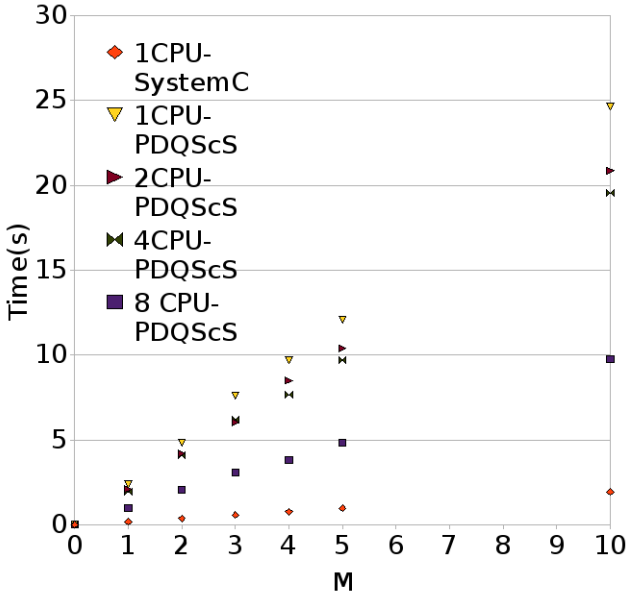
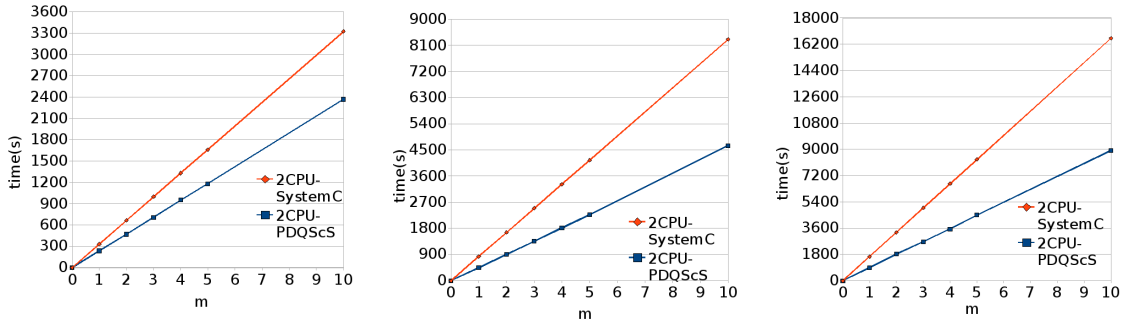


Figure 4.5: Run Times with No Simulation Loop



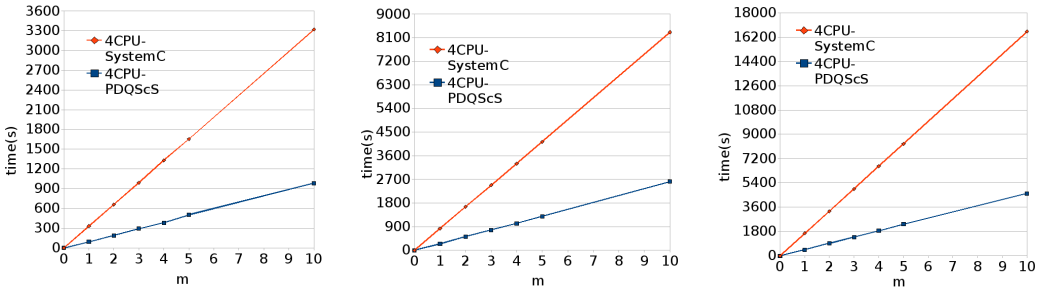
(a) N=2 (b) N=5 (c) N=10

Figure 4.6: PDQScS 2 CPU Run Times

Figures 4.6(a), 4.6(b) and 4.6(c) show the results of simulation run time with 2 CPUs. These graphs show the average time for 5 runs at each workload size. Thus, 5 runs of each of 49 experiments were performed in total. The overhead introduced to the system is slight and simulations are able to perform quicker with a single additional CPU. Even short simulations easily overcome the overhead. Examining the results for two CPUs shows that all results achieved improved performance. When the outer loop is set to 2, the total work ranges from 2-20 work units. This can be seen in figure 4.6(a). All of the simulations in this example had a smaller run time than the baseline.

As the number of work units are increased, the overhead becomes less of the total time. As can be seen in figure 4.6(b), where the outer loop is set to 5 and in Figure 4.6(c) where the outer loop is set to 10, the longer simulations maintain the advantage, but stay within 70-75% of the original simulation time. None of the simulations were able to reach 50% of the SystemC simulator runtime. This is to be expected as the simulation core and communications were not parallelized.

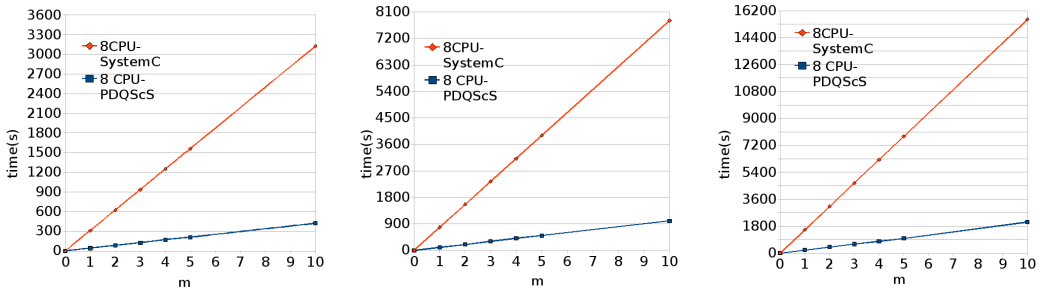
As adding even one additional processor can offset the overhead introduced by the modifications to the simulator, the next question becomes, "How does this scale to a larger number of processors?" A second round of data points was taken with a 2 processor dual-core machine that provided 4 processing units. Figure 4.7(a) shows the equivalent graph of figure 4.6(a) for the four processor system. These experiments showed that, like the two processor experiment, the four processor system had no problems overcoming the overhead. This case has a run time that is 35-40% of the baseline. Note that, like the two processor system, all of the results are faster than the SystemC simulator, but never a one to one ratio with added resources. In this case, the runtime approaches 28% of the baseline run time. Figures 4.7(b) and 4.7(c) show that this trend continues as the simulation becomes longer. This trend can also be seen in the 8 processor experiments in figures 4.8(a), 4.8(b) and 4.8(c). As can be seen in these graphs, the run time fell to %14 of the SystemC simulator run time.



(a) N=2 (b) n=5 (c) N=10

Figure 4.7: PDQScS 4 CPU Run Times

Figure 4.9 shows the speed up factor of the modified simulator vs the SystemC simulator for all experiments and all configurations. While the previous figures clearly showed that new simulator was faster, it is unclear from those graphs how



(a) N=2 (b) N=5 (c) N=10

Figure 4.8: PDQScS 8 CPU Run Times

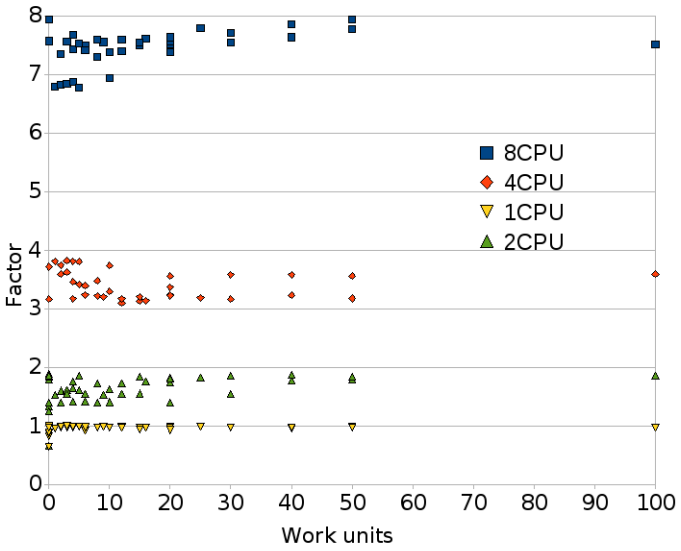


Figure 4.9: Speed increase vs work units

much faster the new simulator performs in the various scenarios when compared with each other. From this graph we can see that for the single CPU case, where the number of work units is low, the modified simulator can be as slow as 1.5 times slower, but simulations are only 2% slower on average. The 2 CPU case clearly shows that for a non-zero value of work units, the new simulator provided between 1.5 and 1.8 times speed up, and the 4 CPU results showed between 3 and 3.6 times speed up. The 8 CPU results show a speed up of between 5.8 and 7.2.

The main conjecture of this thesis is that long running SystemC simulations that are CPU bound should be able to be sped up. These results show that for

this design, and designs like it that have a large portion of the simulation that is CPU bound, using a parallelized SystemC simulator, like PDQScS, can reduce the simulation run time. Furthermore, these results suggest that adding resources to the problem will continue to reduce the simulation run time for the range of CPUs tested. Larger systems were not available for test purposes. Simulations of larger machines would have been possible, but then the simulations would be of simulations, and the accuracy of the conclusions in that setup would be questionable.

None of the experiments used a machine that had more CPU resources than parallelism in the design under simulation. If this case had been examined, the performance increase would have leveled off at the point where CPU resource equaled the parallelism. This needs to be studied further in future work.

4.2.3 Analysis of Processing vs. Communication

Deliberate communications were not needed in the single process simulator case, there were no distributed part that needed to communicate. The multi-process case needs every process's view of the data to stay synchronized. One concern when taking a single process simulator and turning it into a multi-process simulator is the effects of the communications delays due to moving to multiple processes. Much of this complexity was reduced by the use of Shared Memory, but the Shared Memory in turn required the use of semaphores to control access to any values that were stored in Shared Memory possibly marking them as dirty for write operations. Only the memory used to communicate between modules was moved to Shared Memory, reducing the work of interacting with Shared Memory. This has the added benefit of ensuring that communication between modules was only done though SystemC communications paths.

The setup of the experiment offers several simulations that have the same number of total work units, but differing amounts of communication overhead. This is achieved by the work section being called more times, or the work section working longer for each call. Figure 4.10 shows a comparison of these simulations where the number of work units is the same. The graph shows four combinations of outer and

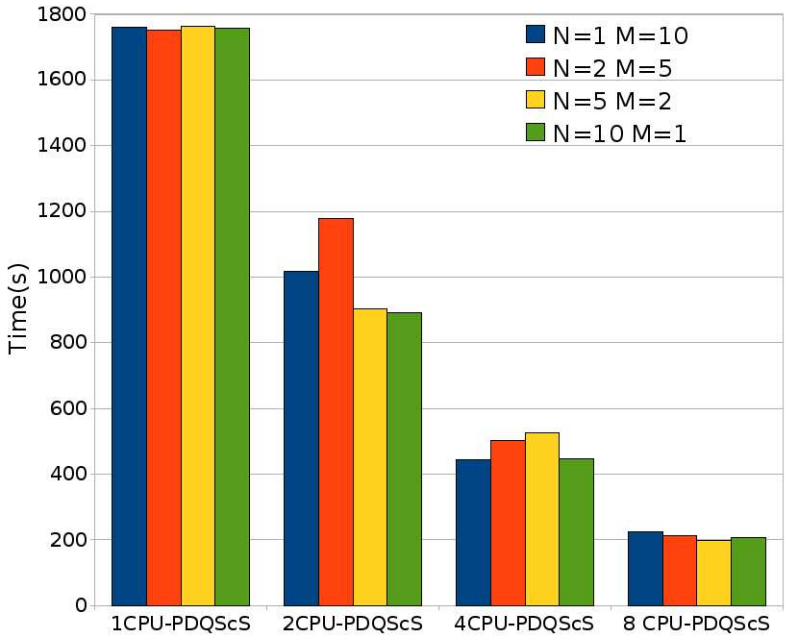


Figure 4.10: Communication vs processing

inner control loops that have 10 total work units. These results are from the $1 * 10$, $2 * 5$, $10 * 1$, and $5 * 2$ experiments. It can be seen by the variance in the groups that the increased communications does have an impact, but the nature of that impact needs further study. On average, within a group there is less than 10% difference between the slowest and fastest result. Given that the average speed up is in excess of 1.6 for 2 CPUs, this 4% worst case is not significant, but should be studied further in future work.

4.3 Comparison to Parallel SystemC

The Parallel SystemC simulator proposed by Ezudheen *et al.* [19] is similar to PDQScS. Both simulators were developed around the non-deterministic execute order within a quantum of the SystemC scheduler. Parallel SystemC uses threads and explicit partitioning and load balancing algorithms, while PDQScS uses processes for all modules and relies on the Operating System's scheduler to handle the load balancing.

Both simulators perform partitioning and load balancing, but do it different ways. Both simulators removed the single module lock from the SystemC scheduler, but replaced it differently. In Parallel SystemC it was replaced with four different partitioning/load balancing algorithms to sort the modules to available CPU resources. In PDQScS the single module lock was replaced with a counter to ensure that all started modules finish before the simulation continues. Both changes allow all the modules to run in parallel instead of sequentially.

Ezudheen *et al.* studied their simulator under some of the same conditions as this thesis, as well as with varying the number of modules in the design. This thesis studied longer running simulations than they did, but with a different DUS. Both studies were able to find speed up through distributed simulation, but without the same DUS used as a common factor it is impossible to compare the results. The main differentiating factor between the two works is the use in this work of a naive partitioning.

4.4 Summary

Table 4.2: Results Summary

CPUs	Average Speed up	Max speed up	Min speed up
1	0.96	1.01	0.65
2	1.65	1.89	1.25
4	3.31	3.82	3.1
8	7.94	7.45	6.77

PDQScS has shown that for some classes of DUS, a parallel distributed simulator can reduce simulation run time. Simulation run time was reduced linearly as CPU resources were added. While there was no benefit to running the PDQScS simulator with a single CPU, the overhead for doing so was only 10% at maximum and 4% on average. A summary of the speedup results can be seen in table 4.2. Results for which either of the control loops was set to zero have been filtered.

Chapter 5

Conclusions and Future work

5.1 Conclusions

The goals of this thesis were to produce a reliably correct, distributed simulator that could reduce run time, require no special skills from the Co-Design systems designer, and scale with the resources and system under simulation. The PDQScS system has been implemented and evaluated as a proof of concept of these ideas. The results clearly show that it is possible to create a simulator that can reduce the run time using parallel programming techniques. The consistency of the results was also shown though both the reproduction of the original results and the small number of changes to the simulator source code required to transform the SystemC simulator into PDQScS. The skills to take advantage of the parallelism are the same skills as the designer needs to run the original simulator, namely to be able to link their design against the simulator library. Finally, the results showed that as the CPU resources were added, the run times for the design under test did decrease. All of the goals of this thesis have been met with varying degrees of completeness, and these will be discussed in this chapter.

It is possible to create a design under test that would produce different answers with PDQScS than with the SystemC simulator. A design that had multiple modules write to a single location would have non-deterministic results in the real world and in PDQScS, these results may or may not also be non-deterministic under the SystemC simulator. The two access patterns form the simulators are both correct, and could both give different answers. PDQScS could even produce answers that are not consistent from run to run. They would, however, be correct. In this case, it is

the design that has a fault, not the simulator.

Many previous efforts in multi-CPU enabled simulators required the user to make decisions regarding partitioning the design. These partitions could be based on separating a design into portions for the hardware and software simulators, or based on the goal of reducing the run time though optimal simulation partitioning. The solution in this thesis eliminates both of these problems. SystemC allows the user to move portions of the simulation easily between hardware and software as the same simulator is used for both. The modifications proposed in this thesis potentially allow the designer to ignore the partitioning for run time optimality, by letting SystemC partition the system into modules and allowing each module to access as much processor as needed and available. By providing a API compliant library, no additional configuration activity is required of the designer to use PDQScS .

One implications of this thesis is that the thread lock in the SystemC simulator should be examined to determine if it is actually providing benefit to the simulation, or slowing it down. Further investigations into other methods of allowing the inherent parallelism in hardware and Hardware/Software Co-Design system should be followed. The most recent version of SystemC added a threading mode that could use kernel threads as there was some concern that the user space threads were not efficient enough. Kernel threads can be scheduled to separate processors without the need to use Shared Memory. This version of SystemC does, however, still have the single thread lock[1]. This lock removes any advantage provided by the kernel threads. This thesis shows, that with some work, the thread lock could be removed to speed up simulation run time.

5.2 Future Work

There are many new projects that could use this thesis as a starting point. A re-implementation of the work presented in this thesis could remove some of the limitations of PDQScS. Work with real world designs could be done to investigate

[1]See the design notes for PDQScS in section 3.2

how much parallelism is actually available to be exploited. A broader range of computer systems could be used to gain further insight into how the run time is affected by adding more resources. Finally, work could be done with the linking pattern of the simulation library so that users would not have to recompile to take advantage of the increased speed.

The modifications to SystemC that were done for this thesis imposed some very hard restrictions on the types of Co-Design systems that could be run. Designs were limited to designs that used the SystemC communication primitives due to the need to have these primitives use Shared Memory. This, in turn, was a restriction needed for work with a multi-process design. A re-implementation that used kernel threads for the threading would not have these restrictions. With these restrictions, removed many more real world designs could be used to explore the amounts of parallelism available to users to reduce simulation time. The solution could then be easily used in industry.

Amdahl's law shows bounds on speed up due to the time spent in sequential and parallelized processing with the addition of resources. This thesis showed that within the bounds of available parallelism, this holds true. The available experimental systems did not extend past the available parallelism in the design under simulation. There are two possible ways to gather results from extremely parallel architectures. Experiments could be done with a Co-Design system that allows for dynamic changing of the parallelism in the design. This study would be able to explore the behavior of the simulator as resources become greater then the parallelism. Alternately, a series of larger SMP machines could be used to attempt to provide more resources than could be used by the design under simulation. As in previous work on parallel algorithms and parallel architectures, it is expected that performance will only improve where there are underutilized physical resources. The time spent in parallelism of Amdahl's law is limited by the access the resources have to that parallelism.

The experiments in this thesis did not attempt to change the volume of data transferred, only to change the number of times the communication was required.

The impact of additional data, or varying volumes of data would change these results and thus should be studied in future work.

SystemC simulations that transfer large amounts of data, and are I/O bound for portions of the simulation, should still see run time improvement. There is negligible communications delay introduced by the parallelization of the simulator. As long as there is a portion of the simulation run time that is CPU bound, the delay due to communications, should be overcome the speed up due to parallel execution.

References

[1] Jay K. Adams and Donald E. Thomas. The design of mixed hardware/software systems. In *Proceedings of DAC 96*, pages 515–520, Las Vegas, NV, 1996.

[2] Charles J. Alpert, Jen-Hsin Huang, and Andrew B. Kahng. Multilevel circuit partitioning. In *Design Automation Conference*, pages 530–533, Anaheim,CA, 1997.

[3] Mary L. Bailey and Michael A. Pagels. Empirical measurements of overheads in conservative asynchronous simulations. *ACM Transactions Modeling and Computer Simulation*, 4(4):350–367, 1994.

[4] R. Banos, C. Gil, M.G. Montoya, and J. Ortega. A parallel evolutionary algorithm for circuit partitioning. In *Proceedings of 11-th Euromicro Conference on Parallel, Distributed and Network based Processing*, pages 365 – 371, Genoa, Italy, 2003. IEEE Computer Society Press.

[5] Moshe Bar. openmosix project end of life announcement. `http://sourceforge.net/forum/forum.php?forum_id=715406`, 2007.

[6] Amnon Barak. Scalable cluster computing with mosix for linux. In *Proceedings of Linux Expo 99*, pages 95–100, Raleigh, NC, 1999.

[7] M. Bombana and F. Bruschi. SystemC-VHDL Co-Simulation and Synthesis in the HW Domain. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, volume suppl., pages 101–105, Munich, Germany, 2003.

[8] Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Cătălin Roşu, and Ray Strong. Efficient message passing interface (MPI) for Parallel Computing on Clusters of Workstations. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 64–73, Santa Barbara, CA, 1995. ACM.

[9] Matthias Brune, Jorn Gehring, and Alexander Reinefeld. A lightweight communication interface for parallel programming environments. In *HPCN Europe*, pages 503–513, Vienna, Austria, 1997.

[10] Kris Buytaert. Migshm rpms. `http://howto.krisbuytaert.be/openMosix-Migshm-rpm`, 2000. Accessed July 2009.

[11] Yu-an Chen, Vikas Jha, and Rajive Bagrodia. Parallel switch-level simulation of VLSI circuits. Technical Report 950020, UCLA, 12, 1995.

[12] Yu-an Chen, Vikas Jha, and Rajive Bagrodia. A multidimensional study on the feasibility of parallel switch-level circuit simulation. In *Workshop on Parallel and Distributed Simulation*, pages 46–54, Lockenhaus, Austria, 1997.

[13] International Roadmap Committee. International Technology Roadmap for Semiconductors: 2004 update. Technical report, International Technology Roadmap for Semiconductors, 2004.

[14] International Roadmap Committee. International Technology Roadmap for Semiconductors: 2008 update. Technical report, International Technology Roadmap for Semiconductors, 2008.

[15] Jon Connel. Early Hardware/Software Integration Using SystemC 2.0. Technical report, Synopsys Inc., 2002.

[16] Jeanine Cook, Richard L. Oliver, and Eric E. Johnson. Toward Reducing Processor Simulation Time via Dynamic Reduction of Microarchitecture Complexity. In *SIGMETRICS Performance Evaluation Review*, volume 30, pages 252–253. ACM, New York, NY, 2002.

[17] Brian Corrie. AccessGrid in the Canadian HPC scene. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 307, Tampa, FL, 2006. ACM.

[18] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. TW: A Time Warp System for Shared Memory Multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, pages 1332–1339, Lake Buena Vista, FL, 1994.

[19] P. Ezudheen, Priya Chandran, Joy Chandra, Biju Puthur Simon, and Deepak Ravi. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 80–87, Washington, DC, USA, 2009. IEEE Computer Society.

[20] C. Fiduccia and R. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 175–181, Washington,DC, 1982.

[21] R.M. Fujimoto. *Parallel and Distributed Simulation Systems*, volume vol.1, pages 147–157. Wiley-interscience, 2001.

[22] A. Ghosh, S. Tjiang, and R. Chandra. System Modeling with SystemC. In *Proceedings of 4th International Conference on ASIC*, pages 18–20, Shanghai, China, 2001.

[23] Sylvain Girbal, Gilles Mouchard, Albert Cohen, and Olivier Temam. DiST: a Simple, Reliable and Scalable Method to Significantly Reduce Processor Architecture Simulation Time. In *Proceedings of the 2003 ACM SIGMETRICS*

International Conference on Measurement and Modeling of Computer Systems, pages 1–12, San Diego, CA, USA, 2003. ACM Press.

[24] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

[25] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, chapter 6, pages 549–560. Morgan Kaufmann, 3rd edition, 2003.

[26] Ken Hines and Gaetano Borriello. Pia: A Framework for Embedded System Cosimulation with Dynamic Communication Support. Technical report, University of Washington, 1998.

[27] H. Hubert. A survey of HW/SW Cosimulation Techniques and Tools. Master's thesis, Royal Inst. of Tech., Sweden, June 1998.

[28] Adriana Iamnitchi and Ian Foster. On fully decentralized resource discovery in grid environments. In *International Workshop on Grid Computing*, pages 51–62, Denver, CO, 2001.

[29] James Jennings and Eric Beuscher. Verischemelog: Verilog embedded in Scheme. *SIGPLAN Notices*, 35(1):123–134, 2000.

[30] Asawaree Kalavade and Edward Lee. The Extended Partitioning Problem: Hardware/Software Mapping, Scheduling, and Implementation-bin Selection. In Giovanni De Micheli, Rold Ernst, and Wayne Wolf, editors, *Readings in Hardware/Software Co–Design*, chapter 4, pages 293–312. Morgan Kaufmann, 2 edition, 2002.

[31] Asawaree Kalavade and Edward A. Lee. Hardware / Software Co-Design using Ptolemy - a case study. `http://www.cse.iitd.ernet.in/esproject/docs/projects/sushant_vivek/sem2/end_term/report/rep/node45.html`, 1992.

[32] Darryl Koivisto. What Amdahl's Law Can Tell Us about Multicores and Multiprocessing. Technical Report 238112, EETimes network, 2005.

[33] Jan Lemeire, Bart Smets, Philippe Cara, and Erik Dirkx. Exploiting Symmetry for Partitioning Models in Parallel Discrete Event Simulation. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pages 189–194, Kufstein, Austria, 2004. IEEE.

[34] P. Luksch. Evaluation of three approaches to parallel logic simulation on a distributed memory multiprocessor. In *Proc. 26th Annual Simulation Symposium*, pages 2–11, Arlington, VA, 1993.

[35] James McCoy and Phil Dreike. Co-Simulating Software and Hardware in Embedded Systems. *Embedded Systems Programming*, June 1997. TechInsights.

[36] Luke Meredith. The future of multi-core processors. Technical report, SearchDataCenter.com, 2005. http://searchdatacenter.techtarget.com/news/interview/0,289202,sid80_gci1141368,00.html.

[37] Swapnajit Mittra, editor. *Principles of VERILOG PLI*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[38] OSCI. SystemC, 2005.

[39] Sunwoo Park. *Cost-Based Partitioning for Distributed Simulation of Hierarchical Modular Devs Models*. PhD thesis, The University of Arizona, 2003.

[40] K.S. Perumalla. Parallel and Distributed Simulation: Traditional Techniques and Recent Advances. In *Winter Simulation Conference*, pages 84–95, Monterey, CA, Dec. 2006.

[41] Larry Peterson and Timothy Roscoe. The Design Principles of PlanetLab. *SIGOPS Oper. Syst. Rev.*, 40(1):11–16, 2006.

[42] Graham Prophet. Sytem-level design languages: to C or not to C. In *Electronic Design Automation Magazine*, pages 135–142. EDN Europe, October 1999.

[43] D. Ramanathan, A. Dasdan, and R. Gupta. Timing-driven HW/SW Codesign Based on Task Structuring and Process Timing Simulation. In *Seventh International Workshop on Hardware/Software Codesign*, pages 203–207, Rome, Italy, 1999.

[44] Sumit Roy and Vipin Chaudhary. Evaluation of Cluster Interconnects for a Distributed Shared Memory. In *Proceedings of the 1999 IEEE Intl. Performance, Computing, and Communications Conference*, pages 1 – 7, Scottsdale, Arizona, 1999.

[45] F. Slomka, M. Dorfel, R. Munzenberger, and R. Hofmann. Hardware/software codesign and rapid prototyping of embedded systems. *Design & Test of Computers*, 17(2):28–38, Apr–Jun 2000.

[46] Mark G. Stoodley and Corinna G. Lee. Vector Microprocessors for Desktop Computing, 1998.

[47] K. Subramani. *The Design of a Time Warp Synchronized VHDL Simulation Kernel*. PhD thesis, University of Cincinnati, February 1997.

[48] Bassam Tabbara, Enrica Filippi, Luciano Lavagno, Marco Sgroi, and Alberto Sangiovanni-Vincentelli. Fast Hardware-Software Co-simulation Using VHDL Models. In *Proceedings of Design Automation and Test in Europe*, pages 309–316, Munich, Germany, 1999.

[49] Mario Trams. Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead. @ www.digitalforce.net, Feb 2004.

[50] D. Verkest, K. Van Rompaey, and I. Bolsens. CoWare – A Design Envoronment for Heterogeneous Hardware/Software Systems. In Giovanni De Micheli, Rold Ernst, and Wayne Wolf, editors, *Readings in Hardware/software co–design*, chapter 4, pages 412–426. Morgan Kaufmann, 2 edition, 2002.

[51] E. Viaud, F. Pecheux, and A. Greiner. An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles. In *Design, Automation and Test in Europe*, pages 1–6, Leuven, Belgium, March 2006.

[52] P. A. Wilsey, D. E. Martin, and K. Subramani. SAVANT/TyVIS/WARPED: Components for the Analysis and Simulation of VHDL. In *VHDL Users' Group Spring 1998 Conference*, pages 195–201, Santa Clara, CA, 1998.

[53] Phillip Wilsey. QUEST II: Parallel Simulation of VHDL. `http://www.cs.uc.edu/~paw/quest/`, 1998.

[54] Koybayashi Yukio. SoC design method for no Re-spin: STARC's measure for Design For Manufacturing. *Semiconductor FPD World*, pages 36–37, August 2005.

[55] Jianwen Zhu, Daniel D. Gajski, and Rainer Doemer. Syntax and Semantics of the Spec C+ Language. Technical Report ICS-TR-97-16, University of California Irvine, 1997.