9-7-2005

RITSim: distributed systemC simulation

David Richard Cox

RITSim:
Distributed SystemC Simulation

by

David Richard Cox

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Assistant Professor Dr. Greg Semeraro
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2005

Approved By:

Dr. Greg Semeraro
Assistant Professor
Primary Adviser

Dr. Roy S. Czernikowski
Professor

Dr. Muhammad Shaaban
Associate Professor

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: RITSim: Distributed SystemC Simulation

I, David Richard Cox, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

David Richard Cox

Date

Dedication

To my parents, for never passing up an opportunity to support my education.

To Heather, for her silent support and occasional wake-up call.

Acknowledgments

I would like to thank Dr. Greg Semeraro for advising this work, pushing me to complete it, and sparking my interest in simulation research. Also, thanks to Dr. Czernkikowski and Dr. Shabaan for serving on my committee.

Special thanks to Paul Mezzanini for his setup and maintenance of the research cluster for my experiments. Finally, thanks to Spencer MacDonald for his work in characterizing the test model and setting up some of the RITSim infrastructure.

Abstract

Parallel or distributed simulation is becoming more than a novel way to speedup design evaluation; it is becoming necessary for simulating modern processors in a reasonable timeframe. As architectural features become faster, smaller, and more complex, designers are interested in obtaining detailed and accurate performance and power estimations. Uniprocessor simulators may not be able to meet such demands.

The RITSim project uses SystemC to model a processor microarchitecture and memory subsystem in great detail. SystemC is a C++ library built on a discrete-event simulation kernel. Many projects have successfully implemented parallel discrete-event simulation (PDES) frameworks to distribute simulation among several hosts. The field promises significant simulation speedup, possibly leading to faster turnaround time in design space exploration and commercial production. However, parallel implementation of such simulators is not an easy task. It requires modification of the simulation kernel for effective partitioning and synchronization.

This thesis explores PDES techniques and presents a distributed version of the SystemC simulation environment. With minimal user interaction, SystemC models can be executed on a cluster of workstations using a message-passing library such as the Message Passing Interface (MPI). The implementation is designed for transparency; distribution and synchronization happen with little intervention by the model author. Modification of SystemC is fashioned to promote maintainability with future releases. Furthermore, only freely available libraries are used for maximum flexibility and portability.

Contents

List of Figures

Glossary

C

causality constraint For a discrete event simulation, the rule that the execution of any event must not be affected by an event with a lower timestamp (to avoid backward causality).

conservative PDES synchronization technique that only allows execution of safe events.

D

DES Discrete Event Simulation - A simulation methodology using a queue of timed events and a discrete clock.

E

ECOT Earliest Conditional Output Time - the earliest next possible message timestamp an LP will send, provided no other messages are received.

EIT Earliest Input Time - the smallest timestamp for which an LP may expect to receive a message.

G

GVT Global Virtual Time - the minimum timestamp of all unprocessed events in the distributed simulation system.

H

HDL Hardware Description Language - A programming language used for the design and simulation of integrated circuits.

L

LAM LAM-MPI - An open-source implementation of the Message Passing Interface from Indiana University.

lookahead Amount of time beyond the local simulation time during which a process will not send messages to another.

LP Logical Process - In a distributed simulation, a thread of execution that models a physical process of the system, and contains its own event list and local simulation clock.

M

MPI Message Passing Interface - A specification for sharing data in a multi-computer environment via messages.

MPMD Multiple Program, Multiple Data stream - Parallel programming paradigm in which different executables operate on different data.

N

null messages Events with no content, but with a timestamp to notify other processes of a safe time to advance to.

O

optimistic PDES synchronization technique that allows for speculative execution of unsafe events, and must recover in the case of an error.

P

PDES Parallel Discrete Event Simulation - Parallel execution of a discrete event simulation across multiple hosts.

R

rank Relating to MPI, a process ID within a parallel program.

RITSim An effort at Rochester Institute of Technology to create a high-performance, scalable, flexible, detailed, and accurate micro-architectural simulator.

RTL Register Transfer Level - A level of abstraction for a hardware description language that allows for synthesizable models.

S

safe event An event in a discrete event simulation that cannot be altered by any event with a lower timestamp.

SPMD Single Program, Multiple Data stream - Parallel programming paradigm in which one executable is replicated to operate on different data.

straggler In a distributed simulation, a remote event that arrives with a timestamp less than the local simulation clock.

SystemC A library providing HDL constructs within C++ to allow for whole system design and simulation.

T

top-level module In Distributed SystemC, top-level modules represent the LPs to be distributed and may share data only through distributed channels.

X

XMPI An X11 program that creates visual representations of a parallel MPI program for debug and analysis.

Chapter 1

Introduction

The RITSim project aims to provide a powerful, accurate, and modular simulation infrastructure. In an effort to facilitate advanced research in low power and high performance computing platforms, RITSim promotes detailed and accurate models with flexibility to quickly test and analyze new ideas. The academic community needs an open simulation framework that is easy to modify, reuse, and analyze [57].

SystemC [32, 52, 53] is gaining popularity as a platform on which to build better hardware models. The C++ implementation lends itself naturally to modeling hierarchical components, with flexibility to interchange modules of different abstraction levels. SystemC draws upon the strengths of object-oriented design, HDL hardware concurrency, and classic C programming for hardware/software co-design. These aspects, along with the active open-source nature of the project make SystemC a logical choice for RITSim.

However, innovation in processor design does not come easily. Smaller features, faster speeds, and intricate digital logic require more simulation resources to accurately gauge the worth of a new design. Low voltage, high-leakage transistors also drive a need to simulate analog and power aspects of new technologies. Computer architects do not wait around for Moore's law to enable higher clock rates; they want to optimize existing parameters and explore creative new designs. A good simulator can only grow more complex (and take longer to run) as more facets of total system performance are studied.

RITSim plans to implement the accuracy, performance, and power necessary to enable advanced microarchitecture research. Eisenbise's cache model [22] is a flexible SystemC

design intended to eventually simulate very sophisticated cache hierarchies. For example, NUCA [37] and NuRAPID [19] present non-uniform cache designs wherein cache accesses do not always incur the same delay. Both strategies attempt to relocate the most-used blocks to the portion of the cache that is physically closest to the CPU core. Shorter wire delays result in lower latency for the most common transactions. Of course, each approach incurs its own implementation penalties. Designers would like to simulate tradeoffs in performance and power before committing to any such design change. A cache hit or miss can no longer simply be modeled by a discrete number of clock cycles; there will be dynamic differences in both power consumption and latency. RITSim models like Eisenbise's [22] permit such versatility.

Other research surrounds making computer systems more reliable. Memik *et al.* [46] point out that the register file, one of the most frequently accessed parts of a CPU, is very susceptible to transient errors. They propose a scheme to duplicate in-use registers into those not currently active. While creating a more robust system, the authors also present simulation data showing a dramatic improvement in energy consumption by the register file. The duplication technique allows for elimination of other power consuming error correction circuitry. If ideas such as this are to be explored further, a complex but flexible simulation tool should allow tracking of all these variables in its register file model.

Not only do researchers want to model and track power consumption, but they also wish to feed that knowledge back into processor designs. In particular, power dissipation needs to be controlled to keep die temperatures at reasonable levels. High temperature ASICs have consequences in long-term reliability of the silicon as well as increased system cooling costs. Skadron *et al.* [58] investigate modeling of processor power and heat for use in design of dynamic thermal management components. They point out how modeling becomes crucial to this task; overall chip or functional unit power dissipation is not accurate enough to infer fine-grained temperature changes. They propose an RC-based model to take into account the entire thermal stackup and interactions between architectural components. A poor model could lead to effort spent cooling parts of the die that never really get hot.

Also, the need to model sensor latency and inaccuracy complicate the problem further. Skadron *et al.* present work on a detailed yet flexible model that invokes confidence in their dynamic temperature management ideas and simulation results.

As the flexibility and complexity required for innovation moves forward, traditional simulators will begin to lag, delaying the benefits of better data. Parallel programming methods are often applied for breaking up large mathematical problems for distribution on several computers. For fast and accurate simulation of advanced designs, similar techniques have been well studied and applied (see Chapter 3). We propose to apply principles of parallel discrete event simulation to speed simulation results. Sequential discrete event simulation kernels such as SystemC are modified to exploit parallel computers. Each host machine can simulate part of the target architecture, communicating through a network and libraries such as MPI [38]. Faster turnaround for simulation cycles will generate volumes of data for making well informed decisions regarding design tradeoffs. Better, faster simulation tools will enable researchers to drive CPU performance and efficiency to a new level.

This document is organized as follows. Chapter 2 explains the motivation behind parallel simulation, in particular a SystemC implementation. Chapter 3 provides background regarding sequential and parallel discrete event simulation, applications thus far, and specific algorithms studied for this work. Chapter 4 examines the design goals of the Distributed SystemC library, its design and implementation, and user interface. Chapter 5 outlines evaluation methods, metrics, and tools. Chapter 6 contains analysis of performance results. Finally, Chapter 7 presents future outlook and recommendations for subsequent work.

Chapter 2

Motivation

Trends of recent microarchitectural conferences indicate that modern superscalar processors continue to employ increasingly complex features to extract maximum performance. Researchers and product designers must not only verify functionality of architectural innovations, but also measure performance to determine feasibility of a new implementation. Skadron *et al.* [57] raise several important issues regarding the challenges of microarchitecture simulation. Where computer simulation has long served as a powerful tool for functional and performance analysis, it is quickly becoming necessary to more accurately gauge the effects of design changes. Architects need to produce detailed and accurate processor models in order to judge the validity of new techniques; results are no longer obvious after running a few benchmarks with only high-level simulators. Some authors worry that progress can be held back if the computer community does not have faith in published results based on sub-par simulators [57]. A good simulation methodology should not have to make a tradeoff between flexibility and accuracy.

Moreover, shrinking transistor sizes and higher clock frequencies spawn interest in simulating many more design statistics, such as power dissipation and temperature [64, 58]. Such data might be easily computed when the design has been specified at the gate level, but is not available early in the design cycle. As a result, computer architects seek tools that allow them to model a system at several different layers of abstraction, from a high-level functional view down to a detailed register transfer level (RTL) or gate level simulator.

Efforts to simulate entire computer systems, such as SIMICS [43], encourage hardware-software co-design and performance prediction of end-user applications. The ability to reuse different versions of component models can have a great impact on simulation flexibility, intellectual property (IP) sharing, and time-to-market [32].

SystemC [32, 52, 53] is becoming a popular framework for expressing many models of digital systems. Built on top of C/C++, it allows the full object-oriented power of the language, while providing constructs to easily describe concurrent hardware behavior. The simulation kernel is based on a discrete event simulation (DES) engine, where individual events are queued and handled at specific simulated times. SystemC libraries provide more complex primitives such as signals, processes, and channels that use discrete events to drive execution. Users may also build libraries to further extend SystemC's capabilities; some analog and mixed-signal extensions are already in the works [21]. Reusability, flexibility, and use of a familiar language has helped researchers and industry create simulation infrastructures that need not be torn down whenever a new project begins. Using the de-facto standard of C/C++ shortens the learning curve for engineers and designers to more quickly implement and simulate their ideas. RITSim hopes to leverage these benefits of SystemC as it moves toward an improved simulation methodology.

However, SystemC is designed to run the entire simulation solely on one processor. Though designers can model concurrent threads, those threads are executed sequentially as one process. Desire to accurately simulate more complex designs to a more precise level of detail creates a significant problem. Subtle microarchitecture changes will need accurate comparative performance data before changes are adopted. Each processor component will be scrutinized in detailed simulation. Smaller transistor features demand tightly refined models to maintain low-level accuracy and track power alongside performance. The processors of today cannot always simulate the architectures of tomorrow within reasonable time and memory constraints. Designers would like to take advantage of massively parallel machines or workstation clusters to hasten feedback on new ideas. Inherent parallelism in current architectures (pipelines, multiple functional units, *etc.*) seems to offer promise for

better simulation performance in a distributed or shared-memory environment.

Parallel or distributed simulation has been studied and implemented in several projects over the past 25 years [47]. In particular, parallel discrete event simulation (PDES) aims at breaking up the event-driven simulation problem onto multiple hosts that communicate results as needed [24]. Debate continues over the efficiency of different strategies, but the potential for increased performance exists nonetheless. Most research revolves around proposing new synchronization algorithms. Implementations are steered toward evaluation of those algorithms with custom simulators. Other solutions are application-specific, proprietary, or require much additional work by the model designer. Currently, no open platform provides a way for microacrhitecture research to easily move models to a PDES simulation environment.

A Distributed SystemC library may combine the ease-of-use and popularity of SystemC with the performance gains of a custom parallel discrete event simulator. The ability of SystemC to enable accurate, flexible modeling at several levels of the design hierarchy makes it an attractive choice for RITSim. Familiar C++ structure will be used to implement the base structure of the RITSim model. Then, as the simulator progresses toward greater detail and complexity, Distributed SystemC enables faster design space exploration without the need for drastic simulator modifications.

Chapter 3

Background

3.1 Parallel Discrete Event Simulation (PDES)

Computer simulation is not a new field by any means. Applications for simulating and evaluating physical systems are vast, and faster methods are always welcome. Introducing the ideas of parallel and distributed simulation in the 1980s and 1990s, surveys by Misra [47] and Fujimoto [24, 25, 26, 27, 28] explain the basics of discrete event simulation. These works summarize the simulation concepts upon which PDES algorithms are built. A condensed version of this background is presented below.

3.1.1 DES and SystemC

A typical discrete event simulator like SystemC consists of simulated state, an event list, and a global clock [24]. State variables track the current operation of the model and any simulation statistics the programmer wishes to record. An event list is a priority queue of events sorted by their pending simulation times. Finally, the global clock advances by discrete amounts of time as events are processed. The basic operation of the simulator is to remove the next scheduled event from the list and execute it. State may change, more events may be scheduled, and the simulation time is advanced to the timestamp of the just-handled event. Delta cycles simulate infinitely small advancements of time that occur when events are dynamically created with the current timestamp.

While it is possible to program models using only simple events, SystemC and other modeling languages provide constructs such as signals and clocks, that use the event-driven kernel transparently [32]. For example, a synchronous digital circuit could be modeled by a process that waits for an action event that is periodically triggered by another process (in this case, a clock). SystemC simplifies this scenario greatly by offering the `sc_clock` object. Such a structure can encapsulate several events, such as rising and falling edges. A built-in SystemC process triggers those events when appropriate. SystemC provides constructs by which to notify other model threads, without explicitly providing access to the internal events.

3.1.2 PDES Concepts

In general, PDES strategies consist of *logical processes* (LPs) that represent physical processes of the modeled system. In the case of computer architecture, an LP might be the branch prediction algorithm, or a floating point unit. These logical processes are essentially self-contained discrete event simulators [6], with part of the system state, queues of pending events, and a local clock. However, LPs may affect each other by sending messages – events with corresponding simulation timestamps. Thus, each LP handles both events generated locally, and events communicated by other processes. As in sequential simulation, events should be handled in monotonically increasing timestamp order. The difficulty lies in the fact that each LP does not inherently know the content or timing of other LP's messages until they are received.

Fujimoto's survey [24] points to the paradoxical nature of such a simulation system; applications may contain parallelism yet are difficult to parallelize. Fujimoto brings to light issues that must be addressed by anyone implementing a PDES system. First, shared state between LPs implies shared memory, and should not be allowed. Distributed processes are often not executed on shared memory multiprocessors, but rather on clusters of networked computers. Thus, assuming shared state between LPs could require a lot of additional overhead to maintain coherency. Since the nature of PDES is such that distributed LPs

will process different simulation times simultaneously, sharing state would involve careful synchronization. Thus, remote state may only be altered by events that trigger action in that remote process.

Furthermore, a *causality constraint* must be met to maintain accuracy. Different processors are handling events concurrently that may or may not affect one another. Even if an LP executes only the smallest timestamp event, a *straggler* event could arrive in the meantime with a smaller timestamp. Any previously changed state alters the course of execution for the next event, resulting in backward causality. Any event must not be affected by an event that occurs at a later simulation time. The fundamental problem with PDES becomes determining whether and how LPs interact with one another. This interaction is dynamic, changing with simulation stimuli.

3.1.3 Basic Synchronization Techniques

To combat this uncertainty, Fujimoto [24] continues by describing the two basic synchronization approaches that have been popularly developed. Conservative methods process only those events which are deemed unable to affect other unprocessed events, while optimistic methods allow speculation and recover from any resulting causality violations. Reynolds [55] points out that these techniques emerge from choosing among several design variables. Variations on these basic themes account for not only conservative and optimistic approaches, but also many other methods that might be tailored for specific applications.

Safe events are defined as those events that the simulator is free to process because it is impossible to receive an event with a smaller timestamp that would alter the outcome. Conservative LPs can execute any safe event, but must block if no events are safe. For the basic conservative scheme, static links are defined between processes that may send one another messages. Each LP, which keeps its own simulation time, is required to send messages on those links in non-decreasing timestamp order only. Then, each incoming event queue is associated with the timestamp of the unprocessed event at its head (or the event just handled, in the case of an empty queue). The queue with the smallest associated

timestamp is selected. The LP blocks if that queue contains no events.

While conservative synchronization guarantees correct causality, cycles of dependent processes make it prone to deadlock. This situation is ameliorated by either the use of *null messages* or by *detection and recovery*. Null messages are sent by LPs as "empty" events with timestamps. The timestamp is a promise that no more events with a timestamp prior to the null message will be sent on the active link. This contract time is calculated based on the LP's unprocessed events, and the minimum simulation time required to process those events. The latter quantity is called *lookahead*, and is often explicitly specified by the application programmer. Detection and recovery schemes do not require explicit lookahead, but additional overhead is needed to detect and break the deadlock.

Optimistic synchronization, on the other hand, lets possibly erroneous events execute speculatively. This allows for greater exploitation of parallelism since processes only block if an error actually occurs. Under the first popular optimistic synchronization scheme, Jefferson's Time Warp [35], process state is saved as each branch of the parallel simulator executes. If an event is received with an out of order timestamp, state is "rolled back" to the time prior to receipt of the straggler message. *Anti-messages* are sent to other affected LPs, and the process is repeated recursively until simulation again proceeds normally. Since each LP keeps its own simulation clock that can move either forward or backward, a *Global Virtual Time* must be maintained to determine what saved state to discard safely. GVT is simply the minimum timestamp of all unprocessed events in the simulation system, and is also used to permanently commit operations that cannot be rolled-back, such as I/O transactions.

Both families of synchronization mechanisms have their benefits and drawbacks. Conservative, while easier to program, may not take full advantage of some parallelism that could be uncovered by optimistic execution. State-saving and recursive rollbacks, however, are processor, memory, and code intensive tasks. Performance of each method is highly associated with the amount of lookahead present in the model application. Whether components explicitly specify their next event time, or optimistic execution takes advantage

of infrequent process interaction, good lookahead will be exploited by an efficient parallel simulator [24].

In the case of very detailed models like RITSim, the amount of processing time between distributed messages should reveal inherent parallelism. The simpler conservative approach may be able to yield acceptable results; especially since the model detail and processing time will only increase as the project matures. Semeraro envisions a simulator that will model "anything (and everything) in a microprocessor design" [1]. Energy and power models will be analyzed throughout the simulation to gain an in-depth understanding of the tradeoffs among different designs and technology. Other tools (like SPICE simulations) may even be executed from within the SystemC model to collect data at the lowest levels possible. A detailed memory subsystem will encompass physical design and other necessary circuits such as termination or error correction. All of these components will have to be flexible and extendible for adaptation with future designs and technologies. Such a complex collection of software will benefit greatly from parallel execution.

3.2 Parallel Simulation Projects

Over the past two decades, many research and commercial projects have attempted parallel simulation in a variety of applications, with mixed success [2, 43, 54, 48, 45, 18, 29, 51, 30, 5, 4, 44, 14, 42, 49]. While the primary goal of most of these works is to decrease execution time [54, 48, 18, 29, 30, 4], some other efforts aim at bringing together different simulation resources that may be geographically distant [2, 43, 28, 51]. For example, Amory *et al.* [2] present a method to allow interaction of several types of simulators to facilitate hardware-software codesign. This particular implementation suffers from too much communication overhead to feasibly speedup sequential simulation, but offers unique advantages for flexible model reuse and IP encapsulation. A system designer can use previously modeled components without access to the source code. Simics [43] is a commercial platform that supports full system simulation. Hardware, software (including the operating system), and

I/O functions are all simulated. Though it is not explicitly a distributed simulation, multiple hosts may run independent, coordinated Simics environments, in order to model real-world computer systems. For example, a web server with multiple clients is simulated on a cluster, with diverse host and target architectures.

Most parallel micro-architectural simulations have focused on exploiting inherent parallelism by modeling chip- or shared-memory-multiprocessors. The Wisconsin Wind Tunnel [54] implements barrier synchronization techniques to accurately simulate a shared memory target. Each processor is simulated in lock-step, one *quantum* at a time; each quantum is small enough to ensure causality constraints are satisfied. Since it is executed on a similar shared-memory machine, processor LPs partition and map naturally, with low communication overhead. The Wisconsin researchers succeeded the Wind Tunnel with Wind Tunnel II [48], attempting to make the system more portable. An abstracted synchronization and message passing interface allows ports to different host platforms, but the Wind Tunnel programs are still limited to simulating parallel computers.

Similarly, Shaman [45] simulates shared-memory-multiprocessors by mapping each target processor LP to a host node. These front-ends operate independently but may request access to a back-end process representing shared memory. Creative *reference filtering* techniques save performance that would have been hampered by frequent communication and high network latency. Chidester and George [18] perform extensive experiments with simulated shared-memory configurations and the Message Passing Interface (MPI). They are able to adjust the communication network to achieve good speedup when simulating a chip-multiprocessor on a cluster of workstations. Finally, more application-specific machines are simulated by George and Cook [29] in the form of parallel DSP architectures. Arrays of digital signal processing blocks are connected to perform specific tasks; the authors want to distribute the blocks throughout a cluster. The parallel programming language Linda is adapted for simulation; it uses virtual shared memory to coordinate each DSP block. Speedup, however, is limited by the frequent high latency communication required to implement the virtual shared memory space.

Other efforts have focused on replicating execution of the same model across multiple hosts. EMSim [51] allows a micro-architectural model to be simulated with varying parameters using a cluster of message-passing workstations. Each simulator runs independently, and the results are collected to determine benchmark performance of the system under study for ranges of parameters. While this type of distributed simulation may ease performance evaluation of a finalized design, small incremental changes still use the full sequential simulation time to validate functionality or performance. Researchers at Paris South University understand this aspect in the creation of DiST [30]. Distributed micro-architecture models execute only part of a benchmark program, but automatically *warm-up* in an attempt to maintain accuracy. Program portions not assigned to a certain processor are emulated, so that each distributed portion of detailed simulation can be overlapped. The authors show promising simulation speeds, reasonable accuracy, and small communication overhead. However, some detailed simulations like RITSim do not tolerate *any* of the inaccuracy that comes from emulation. Also, these examples only model processor architectures; a more general approach is desired for distributed simulation of other types of hardware and software. The discrete event simulation paradigms must still be followed.

To provide a more general framework for parallel simulation, several PDES language extensions or libraries have been developed. Bagrodia [5] surveys several of these environments. Operating systems like TWOS (Time Warp Operating System) provide programming interfaces to hide the underlying optimistic synchronization protocols. On a smaller scale, libraries like SPEEDES may be used by C/C++ modelers to provide function calls that allow construction of parallel simulations. Language extensions like Maisie create new primitives that enable a user to build parallel simulations more efficiently than with add-on libraries. Parsec [4] further extends Maisie to support different synchronization algorithms and host architectures. Wu *et al.* [63] outlines the extension of OMNeT++ for distributed simulation of queuing telecommunications using synchronization over MPI. Bagrodia also

describes additions to parallel languages to more easily support simulation. Another approach called ParaSol [44] creates threads and LPs that may dynamically migrate to different processors, instead of sending event messages. ParaSol's designers believe the *active-transaction* view can allow for a more natural modeling of some applications, along with increased performance from dynamic load balancing. A more recent effort by the Department of Defense has led to the High Level Architecture (HLA) [28] specification to standardize interactions among different simulators. The HLA is primarily used to coordinate simulations of geographically distributed resources and personnel, rather than purely for simulation speed. It defines a set of design guidelines for those wishing to create their own parallel simulators.

All of these projects follow the traditional PDES paradigms of partitioning a simulation into LPs with event lists. Each provides different levels of abstraction, language semantics, or modeler input. Most of the concrete implementations using such frameworks revolve around telecommunication networks, where components are often represented directly by queues of events. These custom and diverse examples, like the multi-processor simulators described earlier, have not quite succeeded in pushing distributed simulation into the digital hardware design mainstream.

3.3 Distributed HDL and SystemC

Attempts are being made, however, to integrate PDES into popular hardware modeling standards. Chamberlain [14] writes an early survey of the work involved with bringing parallel simulation to VLSI logic. He mentions many of the same concerns addressed by other authors, including partitioning, synchronization, and granularity. Subsequently, parallel VHDL simulators began to emerge [42, 49]. Naroska [49] describes partitioning a VHDL model using graph theory, where the weights of links between nodes are used to schedule and synchronize events. The same ideas are elaborated in other conservative PDES approaches.

Lungeanu and Shi [42], assimilate each VHDL signal or process to an LP in the simulation; several tightly coupled LPs may be mapped to the same processor for more efficient execution. The authors claim that any VHDL circuit may be simulated, and no explicit lookahead information is required (though it is optional to increase performance). Each LP can dynamically adjust to an appropriate conservative or optimistic synchronization method, allowing potentially high speedup [41]. An algorithm to calculate GVT runs in parallel with the simulation. It is used both to reclaim storage from optimistic state-saving, as well as a safety condition to ensure progress in the face of deadlock-prone, zero lookahead loops.

A recent framework for Distributed Verilog Simulation (DVS) is presented in [39], that automatically parses and partitions a netlist into a graph-like structure similar to the VHDL approaches. The object-oriented design uses an optimistic simulation engine powered by MPI. Related signal or gate LPs (that might be part of a system subcomponent) are clustered together into one process to avoid heavy message traffic. Performance is disappointing so far, but the authors are confident that future improvements will make distributed Verilog much more effective.

While VHDL and Verilog are actively used in the hardware modeling world, SystemC is emerging as a new standard for system design and simulation [32]. Not only can SystemC model signals and gates like other hardware description languages, but the hierarchical nature of object-oriented C++ makes it a natural choice for modeling at different levels of abstraction. Furthermore, it is already capable of hardware-software codesign, without the need for special modifications. With system-on-chip (SoC) designs requiring fast validation, both hardware and software engineers' familiarity with C++ helps to push SystemC to the front of the EDA community. With an event-driven simulation engine at its core, it is only a matter of time before PDES techniques are applied in full force.

A few efforts have already been made toward such an end. SimCluster [34] is a commercial tool by Avery Design, Inc. that integrates several popular VHDL and Verilog simulators. The individual models execute on distributed processors such as a cluster of workstations, and are coordinated by a central server process. Avery advertises substantial speedup and the ability to combine diverse models. The SimCluster datasheet indicates that C/C++, Perl, or SystemC models may also be integrated and simulated on the distributed network. However, this integration requires setup by the system modeler and it is unclear whether the same performance gains are possible.

Meanwhile, a few individuals in the SystemC community are busy working on parallel models. Hamabe [33] performs a manual graft of MPI library calls into a few simple SystemC example programs. Each MPI process executes an identical SystemC model, and is assigned a *rank* (process ID) by MPI. The behavior of each instantiated model is defined by that process' rank. The most complex example consists of three identical processor models exchanging data over both shared and dedicated buses. While these examples show the adaptability of SystemC to use C/C++ libraries like MPI, there is no interaction with the event-driven kernel. The distributed processes are homogenous and only act differently due to branches in the code. Hamabe has also begun to develop his own simplified SystemC-like simulation kernel, in which he plans to include MPI support.

The work of Mario Trams [59] represents a much more applicable endeavor. Trams develops a plug-in library to be used in conjunction with SystemC, without disturbing the SystemC kernel at all. In a tradeoff for modularity, this approach does not attempt to provide a transparent solution to automatically distribute existing SystemC code. As a result, Trams quickly rejects an optimistic approach that would require modifying SystemC to save process state and set simulation time backwards. Thus, a conservative route is taken in which the model programmer explicitly specifies lookahead parameters for outbound LP signals. Library calls setup the model partitioning and communication topology, and synchronize inbound and outbound signals based on the provided lookahead values. Explicit lookahead may suffice for the synchronous designs Trams targets, but complex, detailed

processor interactions need to be modeled by asynchronous events. Trams [59] indicates that these goals can probably not be achieved without significant reworks within the SystemC kernel.

A more transparent solution is possible by combining some of the techniques applied to VHDL toward a PDES-aware SystemC simulation engine. While some user guidelines may be necessary to achieve reasonably efficient performance, synchronization should be seamless. Restrictions are necessary, but cannot severely limit a model designer's ability to express the hardware fluently. The full power of SystemC primitives must be used to execute different models of computation in parallel. RITSim provides such a solution. Extending conservative parallel discrete-event simulation to SystemC is the next step toward quick and reliable evaluation of tomorrow's complex microarchitectures.

3.4 Conservative Algorithms

Though a brief overview of PDES techniques reveals a simple taxonomy divided between conservative and optimistic synchronization, there are many other variations on the algorithms. Much effort has been spent developing and tailoring conservative techniques to achieve better performance without the need for state saving and rollbacks. To this end, Fujimoto [24] and Nicol [50] stress the importance of lookahead. Nicol [50] elaborates on the subtleties of lookahead; it can have context beyond a simple timestamp. For example, lookahead may represent a bound rather than a concrete time. Furthermore, a process may pass along information about how it will affect others, rather than just when (Nicol calls this *content lookahead*. Lookahead also implies a certain scope; it is important to know whether future contract times apply to all LPs, a local group, or specific processes along a defined path. Attempts to increase performance of a conservative parallel distributed simulation usually involve exploitation of lookahead, when available.

3.4.1 Beyond Null Messages

The seminal work by Chandy, Misra [16], and Bryant [11] implements null messages as described earlier. Often called CMB null messages, the method allows lookahead information to be passed along the normal communication path of the logical processes. To reduce null message traffic and processing, lookahead information may be added onto the standard event messages required for the distributed simulation.

Bain and Scott [8] go one step further toward eliminating unnecessary null messages. They propose a demand driven technique, in which an LP requests to advance its simulation time when it would otherwise block. This method relies on the static communication topology of the distributed simulation. The simulation structure can be represented by a directed graph. As requests are sent to predecessor processes and passed on through the network, queues of request history must be maintained to detect feedback loops. Replies based on each LP's state will propagate lookahead and release blocked processes.

In another adaptation of conservative synchronization, Nicol [50] defines *appointments* as promises by LPs to not send messages on certain communication links beyond the appointment time. While this seems very much like a null message, the receiving LP can use the lookahead information immediately. A null message will not be processed until the LP's local clock has reached that of the null's timestamp. An appointment, however, can be viewed at any time and used to make decisions on how to calculate lookahead and set subsequent appointments. An efficient implementation of an appointment protocol will take into account model details that allow fast propagation of lookahead values throughout the simulation network, ideally leading to less blocking and better performance. Groselj and Tropper [31] implement a version of appointments with the "time-of-next-event" (TNE) algorithm. A link time, which is a lower bound on the next output message, is computed by each LP for each outgoing link. Like Nicol, TNE [31] uses knowledge of queued events, model-specific lookahead, and current link times (*i.e.* appointments) to propagate lookahead through the LP network and ensure safe advancement of simulation time.

Like demand-driven and appointment techniques, the *carrier-null* method tracks message route information to expose loops, prevent deadlock, and propagate lookahead. Cai and Turner [13] presented these route-aware null messages to make better use of content lookahead in simulation of logic circuits. The original approach used two parallel threads for each LP; a synchronization thread handles message traffic and computes new upper bounds on the simulation timestamp for the main thread. Carrier-null synchronization dynamically discovers loops through the route information.

Wood and Turner [62] later refined the method to allow for deadlock-free execution of nested loops within the distributed simulation. The topology of the partitioned processes is determined statically at the start of the simulation; each LP has a graph representation of the simulation world. While the authors did not achieve particularly impressive speedup with this attempt, they did show a substantial reduction in synchronization messages. A well-partitioned model could take advantage of this technique if it offered enough simulation complexity. However, a complex partitioning and mapping of LPs onto processors could negate the benefit. For higher complexity, larger route traces would be necessary, and messages would traverse many more nodes before providing conclusive lookahead results.

3.4.2 Lacking Lookahead

Some simulation applications offer very good lookahead. For example, models of queuing networks used in much of the work mentioned here inherently have good lookahead; each process knows what its simulated service time (or bound on that time) will be for a particular event. Other target simulation models do not have such nice characteristics. In particular, a VHDL model that contains delta cycle delays may not be able to predict if/when future messages will be sent out. Moreover, attempts to modify current sequential simulation infrastructures for PDES should allow any model to still execute correctly, regardless of lookahead. User-specified or dynamically discovered lookahead should act only as a bonus to increase performance.

Blanchard *et al.* [10] attempt to increase performance by allowing some knowledge of

LP state to be shared within localized regions of processors. Their *cooperative acceleration* algorithm targets simulations with very little (or no) lookahead and high uncertainty of future events. By maintaining additional state that can be dynamically shared among LPs, the algorithm identifies groups of processes that lag behind the others. Special "hypothesis" messages are used to identify a simulation time to which the group can collectively and safely advance.

Cota [20] seeks simulation primitives to provide an up front formal description of modeled process behavior. Knowing the characteristics of the physical system and derived simulation model, the simulator constructs a control flow graph of the process interaction. If all that information is not available through explicit model constructs, then some preprocessing must take place to build the graph. After the graph has been constructed, Cota's algorithms can extract lookahead automatically. While this approach does not directly solve the problem of poor lookahead, it proposes some concepts for simplifying model specification that could lead to exposure of lookahead without the model writer's interference. Many of the distributed simulation algorithms discussed have similar themes to the control flow graph; they differ on the level of information explicit in the model, and how the graph is utilized for performance gains. The classic tradeoff exists as to whether a distributed modeling language should rely on such information, or be able to survive without user interaction.

Another interesting approach to eliminating the dependency on lookahead is laid out by Lin *et al.* [65]. Lin examines the fundamental reason why a classic conservative method (CMB null messages) is prone to deadlock – feedback loops with no lookahead. If we can eliminate these loops from the distribution, the simulation network becomes feed-forward, allowing safe advancement of time even without good lookahead. Model preprocessing can be used to determine which sets of LPs are "strongly connected," *i.e.*involved in a low- or no-lookahead loop. The simulation partitioning is reconfigured so that these clusters execute in the same sequential process, eliminating the circular dependency and preventing deadlock. Parallel speedup will be limited by the number of such clusters. Lin formalizes

the idea that a distributed simulation in which every process affects every other (particularly those with low lookahead) is bound to suffer poor performance. A less complicated web of interacting events should be better suited for parallel execution. Though a dynamic repartitioning algorithm is beyond the scope of this thesis, Lin's paper shows promise for this technique to extract improved performance from even the simplest conservative synchronization. For RITSim, performance will depend on a good up-front partitioning of the model by the user. Appropriately balancing the modules for distribution will be left to the architect with knowledge of the model's behavior.

3.4.3 Synchronous and Other Methods

Along with the ideas of null messages, Chandy and Misra also pioneered the application of deadlock detection and recovery schemes for distributed simulation [15]. This method is conservative in the sense that it does not allow any process to execute unsafe events. It does not employ null messages; if the distributed processes are well balanced, the simulator can take advantage of good lookahead implicitly. However, when those processes block, a separate mechanism must be available to compute lookahead and release blocked processes. A parallel algorithm to detect when the simulation has deadlocked is employed and will generate communication traffic alongside the simulation.

A final category of parallel discrete event simulation algorithms is conservative in spirit but does not rely on traditional null message passing. In general, *synchronous* methods will perform some global reduction across LPs to determine a safe simulation time [50]. For models with good lookahead, a long period of parallel computation will occur between synchronization periods. In the worst case, low lookahead results in frequent synchronization that can nullify parallel speedup. Several researchers have investigated synchronous algorithms on their own, and as part of a hybrid approach with some other technique.

Lubachevsky [40] uses a synchronous global lookahead calculation coupled with information about the interconnection of the processes. The "Bounded Lag" algorithm identifies loops within the network, and can determine lookahead for *spheres of influence*. A group

of processes can compute a lower bound on the earliest time that sphere may be affected by an external LP. The work primarily targets queuing network simulation, or other SPMD-type applications, such as a toroidal network of processors. What separates Lubachevsky's research from other methods that use network graphs is the use of synchronous global calculations, rather than directed messages, to propagate lookahead through the simulation.

Chandy and Sherman [17] approach the problem by relating parallel simulation to sequential discrete event simulation. They point out that on any event list (sequential or distributed), there are *conditional* events. That is, events may be preempted, removed, or altered by the receipt of another event. The job of the simulator is to determine which events have become unconditional and can therefore be executed. In a sequential simulation the task is easy - the event with the smallest timestamp is always unconditional and should be executed next. For a distributed simulation, since there is no global event list, a reduction operation must be performed to mark events as unconditional. In the worst case, the event with the lowest timestamp across all LPs is the only safe event. This becomes a calculation of GVT, like that used in state-saving optimistic synchronization for re-claiming memory. However, model connectivity information and lookahead can be used to generate a better bound on simulation time and allow speedup. Jha and Bagrodia [36, 7] implement a version of the conditional event algorithm and evaluate its performance. They also combine it with null messages, allowing good lookahead to be passed on when it is available, and the synchronous reduction to prevent deadlock when lookahead fails. The synchronous method echoes the deadlock recovery algorithms, but without the need for a mechanism to detect the deadlock in the first place.

The conditional event approach ties together elements from traditional conservative synchronization, while still providing a functional simulator for models without lookahead. For the latter type, models with large parallel workloads can see reasonable speedups. A variant of this algorithm is appropriate for implementation into SystemC, where a pre-established simulation protocol provides no explicit access to model lookahead information.

Chapter 4

Distributed SystemC Kernel

4.1 Goals and Tradeoffs

The following are goals of an ideal Distributed SystemC program within the context of RITSim:

- Speed - A PDES application for SystemC should create potential for decrease in simulation time.
- Transparency - Model writers have minimum interaction with the distributed simulation mechanism. Beyond initial setup parameters, compile switches, or basic runtime arguments, a designer creates and executes a parallel simulation in much the same way as a sequential one.
- Accuracy - Deterministic models execute with the same results, regardless of the number of processors used.
- Compatibility - Previously written SystemC models run in parallel, with little or no changes required to the model source code. Parallel or sequential execution is easily selectable.
- Portability - Only free, open tools, such as the Message Passing Interface (MPI) [38] communication framework, are used to encourage portability across different architectures and commodity off-the-shelf (COTS) components.

- Maintainability - While modifications to the SystemC kernel may be necessary, encapsulating these additions in specific libraries or classes is encouraged in order to promote integration with future releases.

Some of these goals are conflicting. For example, speed becomes a tradeoff with accuracy, compatibility, and transparency for some applications. In order to support a wide range of SystemC models that may not inherently fit the parallel simulation paradigm, a slower synchronous algorithm is chosen. Also, changes to the SystemC kernel are made in order to achieve a higher level of transparency. Finally, the goal of transparency suffers for the desire to keep this first Distributed SystemC library somewhat simple; we do not provide a way to automatically partition models for distribution, but only the means for building and executing pre-partitioned models. As discussed in section 7.2.3, automatic partitioning is not a trivial task, but should be possible as an extension of the work already provided. An automatic partitioning algorithm could support optimizations for achieving maximum speedup for different types of models, but is beyond the scope of this work. However, the underlying goal is straightforward: create a new SystemC that can take advantage of model parallelism without levying a heavy burden on the average developer.

4.2 Design and Implementation

4.2.1 SystemC and Distributed Model Structure

The first major design decision for the Distributed SystemC library surrounds how a large sequential model is divided and built for execution on a computer cluster. Parallel programs are often classified as SPMD (single program, multiple data stream) or MPMD (multiple program and data streams) [61]. With SPMD, one set of instructions is replicated across each processor, but each operates on separate data. Different parts of the source code may operate on different portions of data depending on their processor assignment, but there is

only a single executable. For example, a parallel image processing algorithm could run several copies on multiple processors, each working on a subset of the image's pixels. MPMD programs, however, consist of two or more executables working together on different processors. Each operates on its data independently, but coordinates through message passing or shared memory when necessary.

Since each logical process of a distributed discrete event simulation can be thought of as a separate sequential simulator, RITSim will use the MPMD paradigm to distribute SystemC. Each LP is contained within its own executable linked against the Distributed SystemC library. Then, all executables are launched together so that they may communicate using message passing (MPI) routines. Most of the discrete event engine is implemented within SystemC's `simcontext` class. There are numerous other classes to represent events, data structures, and simulation objects, but only the simulation context code is modified for the sake of the distributed library. Thus, each logical process has its own `simcontext` to control the simulation flow.

It is important not to confuse SystemC processes (`SC_THREAD`s and `SC_METHOD`s) with the PDES logical process as defined by this work. The SystemC constructs represent concurrent activities within the model; for example, individual drivers in parallel along each address bit of a bus. Zero or more of these threads can be contained within any `SC_MODULE`, and modules may be nested. Following the previous example, the output drivers of a memory bus might all be encapsulated within one `SC_MODULE`, that resides within another module representing the entire memory controller. Though it is an over-simplification of communication within SystemC, modules generally exchange information through *channels*, such as `sc_signal`. Channels are connected to input and output ports, and activity on those ports triggers events that the kernel uses to activate the various concurrent threads. All of these constructs can be contained within a single LP in Distributed SystemC, using whatever hierarchy the model programmer chooses.

However, we define a new level of the hierarchy called the *top-level* module. A top-level module is just another `SC_MODULE`, but we impose some artificial restrictions that allow

for the distribution scheme to work. First, a top-level module must not be contained within any other module, hence the name. Furthermore, it cannot be allowed to share data with any other top-level module, except through SystemC ports. Even though C++ easily allows passing pointers or retrieving class variables through method calls, it is necessary to avoid any direct sharing among LPs (see Section 3.1.2). Inputs and outputs of top-level modules are specified as `SC_PORT` objects, as with any `SC_MODULE`. Figure 4.1 shows a sample construction of part of a distributed model. Note how SystemC modules and processes can maintain their hierarchy within the top-level module.

4.2.2 Message Passing Between LPs

Instead of connecting to a normal SystemC channel, ports of top-level modules connect to special primitive channels that derive from the same base class, `sc_prim_channel`. Figure 4.2 shows the structure of these additional channels related to the original SystemC base classes for the signal type. Though they perform all the same functions as the standard primitive channels (and could actually be used as such), `sc_channel_distributed` objects add special functionality for passing messages across LP boundaries. Only two types of these channels are currently implemented: `sc_signal_distributed` and `sc_buffer_distributed`. If a distributed channel is mistakenly connected to a module that is lower in the hierarchy, it simply behaves as any other SystemC channel.

When a distributed simulation begins, an initialization period registers all such distributed channels that are connected to the top-level module. Since SystemC assigns all objects a unique string name, we use that identifier to symbolize a connection between top-level modules. Each LP will send initialization messages indicating the signal name and value at time zero. Through acknowledgments, each simulation kernel tracks the other LPs to which it is connected. During simulation, a part of the user's model may write to a distributed channel, causing MPI messages to send a new value to any affected processes. Each fully implemented distributed channel has a `write` function that updates the new

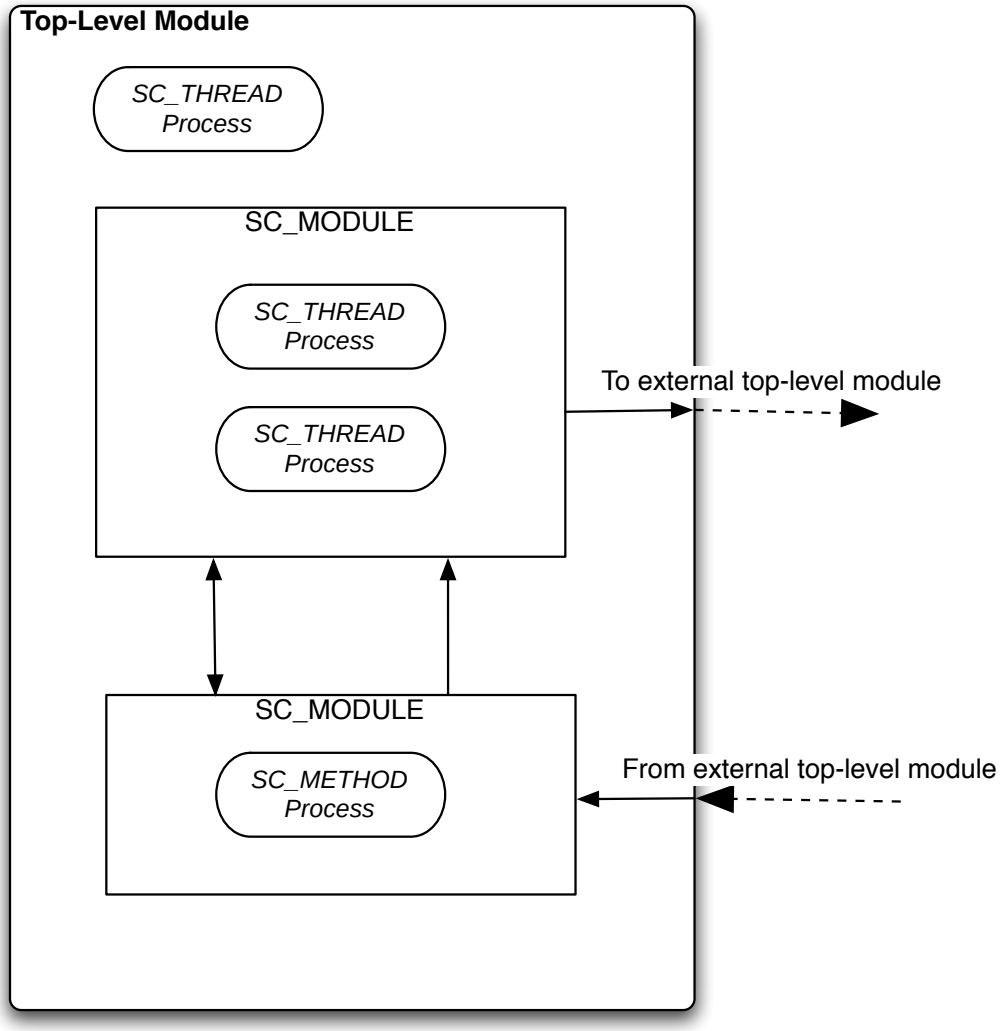

Figure 4.1: Sample model hierarchy for a distributed top-level module

value and calls `send_update_messages`, a function inherited from the parent class (see Figure 4.2). This structure helps to maintain the user-extendibility of SystemC; other primitive channels could be implemented for distribution without knowledge of the MPI calls.

Messages contain the channel name, timestamp, and new value. When triggered at the appropriate local simulation time, the kernel translates the MPI message back into a SystemC event. On the receiving side, a virtual function called `write_remote` is added by

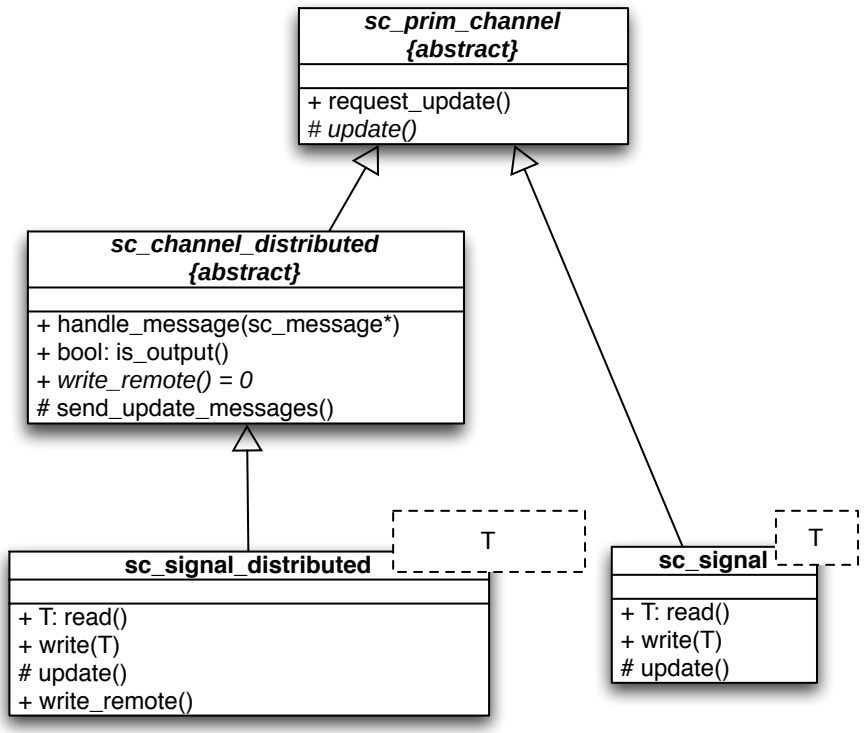

Figure 4.2: Hierarchy of SystemC and Distributed primitive signals, showing relevant operations.

`sc_channel_distributed` that must be implemented by child classes. This function serves the purpose of creating an event local to the LP where a message is received. The simulation kernel calls `write_remote` when acting upon a received message, to notify the LP that one of its input channels has changed value. In turn, the previously mentioned `write` is called, but circular MPI messages are not sent because the caller is the kernel thread, and not a user's model thread. An overview of this process is represented by Figure 4.3.

Support for channels of different datatypes is built in, as in standard SystemC. Users specify the datatype as a template argument, for example, `sc_signal_distributed<int>`. All native C/C++ datatypes are stored in a union and sent as raw bytes in MPI messages. Vector types that include string characters (such as `sc_logic_vector`) will be sent as ASCII strings. As channels are matched across LPs during the initialization, the datatype is

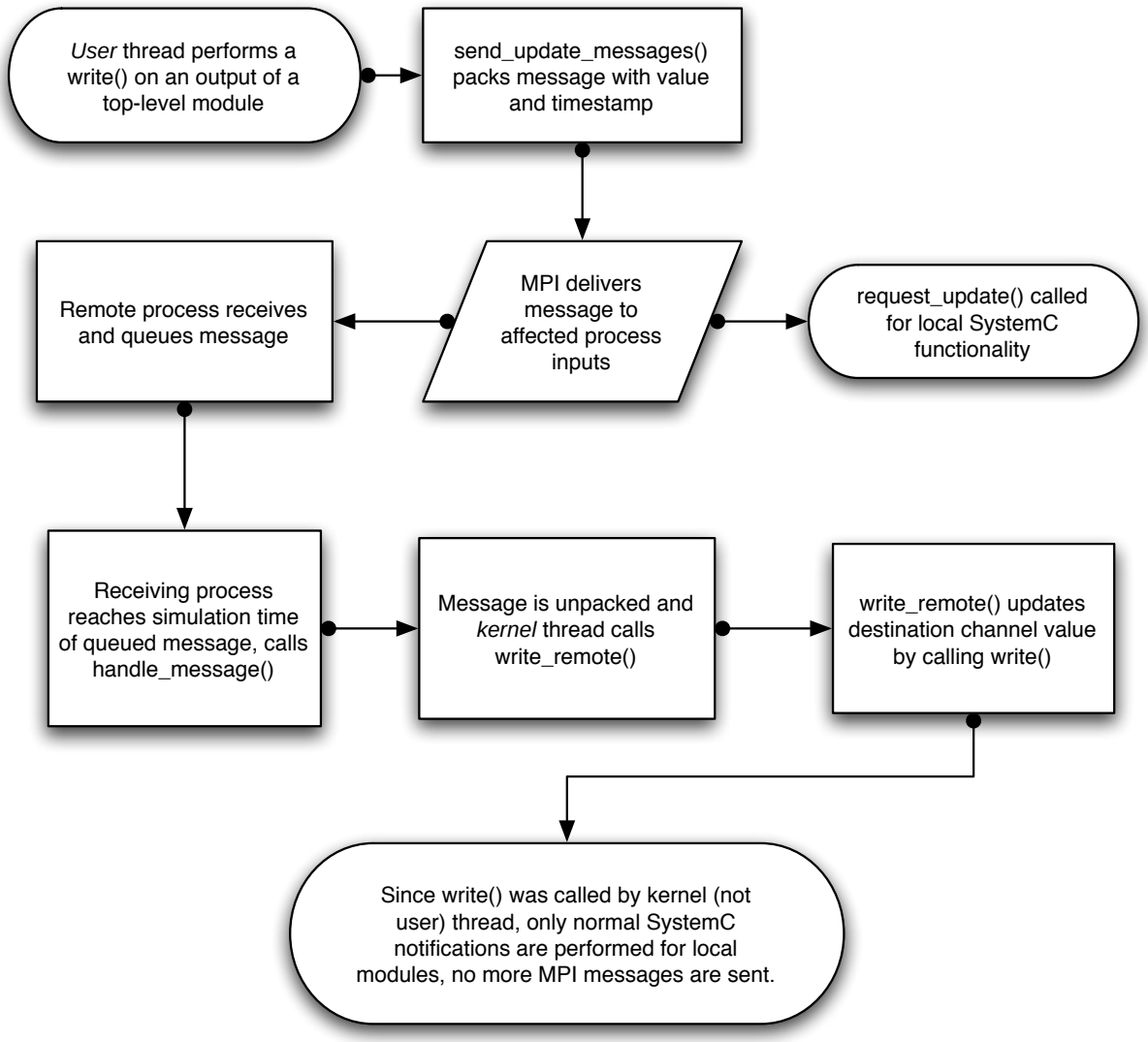

Figure 4.3: Sending and handling MPI messages initiated by writing to distributed channels

indexed for fast lookups of the correct bytes upon receipt of a message. Custom user datatypes can also be used, as long as stream insertion (<<) and extraction (>>) operators are provided for use in conversion. Native types, of course, will require less overhead when packing messages and should be used whenever speed is crucial.

4.2.3 Simulation and Synchronization Algorithm

In the attempt to support a wide variety of models that might be generated by a SystemC user, we choose a robust synchronous algorithm that does not require explicit lookahead information. The algorithm chosen is a variant of the conditional event approach [17], like the one implemented in [7] and discussed in Section 3.4.3. It is added into the SystemC evaluate/update protocol [52] with the intent of maintaining the same basic simulation flow. Handling remote events and synchronizing distributed processes are just additional steps grafted onto the SystemC methodology.

Each LP handles events in increasing timestamp order only, whether they are local events or queued from receipt of a remote MPI message. Since RITSim employs a purely conservative approach, only safe events may be processed at any given simulation time. As simulation progresses, time cannot be advanced beyond the *earliest input time* (EIT [36]) – the smallest timestamp for which the LP may expect to receive a message. Whenever all of an LP's events are beyond EIT, global synchronization must occur to update EIT and allow for advancement of simulation time.

This safe time is computed collectively by all LPs. Each LP determines its *earliest conditional output time* (ECOT [36]) and communicates that time to all other processes. ECOT can essentially be defined as the earliest next possible message, provided no other messages are received. If models have lookahead information, it would be included in this value. For instance, an LP might have a next internal event e at time t_e. Without lookahead, t_e is the ECOT. If the model understands that all events like e will not send external messages until t_L afterward, ECOT could be $t_e + t_L$. For the time being, RITSim models cannot include lookahead in ECOT computation.

Since communication is not instantaneous, we must take into account the possibility of in-transit messages at the time of synchronization. Thus, the ECOT is adapted to become the minimum of the ECOT computed above, and the timestamps of possible in-transit messages sent since the last computation. Each LP updates its EIT (to determines safe events) to the global minimum of all ECOT values, and simulation proceeds.

Delta-cycle accuracy is maintained by considering the delta cycle as an extension of the timestamp. When any LP creates a new local event at the same discrete timestamp, a delta cycle must be added to simulate the infinitesimal change in time yet still maintain correct causality. Thus, a delta-delay notification from one distributed process can trigger events in the correct order on any remote process at the receiving end. This level of accuracy creates a lot of synchronization for tight cycles among distributed processes wherein signals are triggered on the next delta cycle, rather than a discrete amount of time later.

4.3 Using Distributed SystemC

For the RITSim Distributed SystemC library to achieve its goal of transparency, the design details described in the sections above should not become a burden to the model writer. To that end, the library implements several levels of automation for transforming sequential SystemC code into an automatically distributed model. However, there are a few guidelines to writing such a model to satisfy the structure defined in Section 4.2.1:

- *Partitioning.* As mentioned in Section 4.2.1, the model writer must determine how the distributed model is partitioned into top-level modules.
- *Shared Data.* Again, no data may be directly shared across the boundaries of top-level modules. All such communication is achieved through distributed channels.
- *Datatypes.* Distributed ports support ALL built-in C++ and SystemC datatypes, though the primitive types will have better performance. User datatypes may be used if they provide stream insertion and extraction operators.
- *Trace Files.* Individual trace files within a top-level module may be used, but they cannot trace across distributed boundaries.
- *Main Function.* `sc_main` should NOT be defined in the source file of any top-level module, but rather in its own file.

4.3.1 Fully Automated Distribution

There are a few other minor restrictions regarding file names and directory structures; these instructions are spelled out in the library's README files. In general, the procedure for using the automation tools is very similar to creating of any other SystemC model:

1. Create model components in the normal sequential manner, following the guidelines above for top-level modules.

2. Create an `sc_main` function that instantiates and ties together all top-level modules with normal SystemC signals. Signal names are helpful but optional, since SystemC guarantees uniqueness.

3. Specify optional timing information, like time resolution or default time unit. They will automatically be copied into each distributed module.

4. Instead of calling `sc_start`, call `sc_start_distributed` with the same parameters.

5. Create a Makefile (or other build script) to compile and link your module with the sequential `sc_main`. Use the MPI C++ compiler with `-lsystemc` and `-loompi` link options

6. Set the `SYSTEMC_HOME` environment variable to the install path of the Distributed SystemC library.

7. Run the executable.

When a properly formatted sequential model calls `sc_start_distributed`, the automation tools create several configuration files and start the parallel execution. This work is shared between the C++ code and a perl script.

4.3.2 Automation Details and Customization

Figure 4.4 shows the various steps for the automation tools to build and execute a distributed model. Any of the input files can be modified, and the user can specify which steps to run selectively. The flowchart shows the files that are generated, described below.

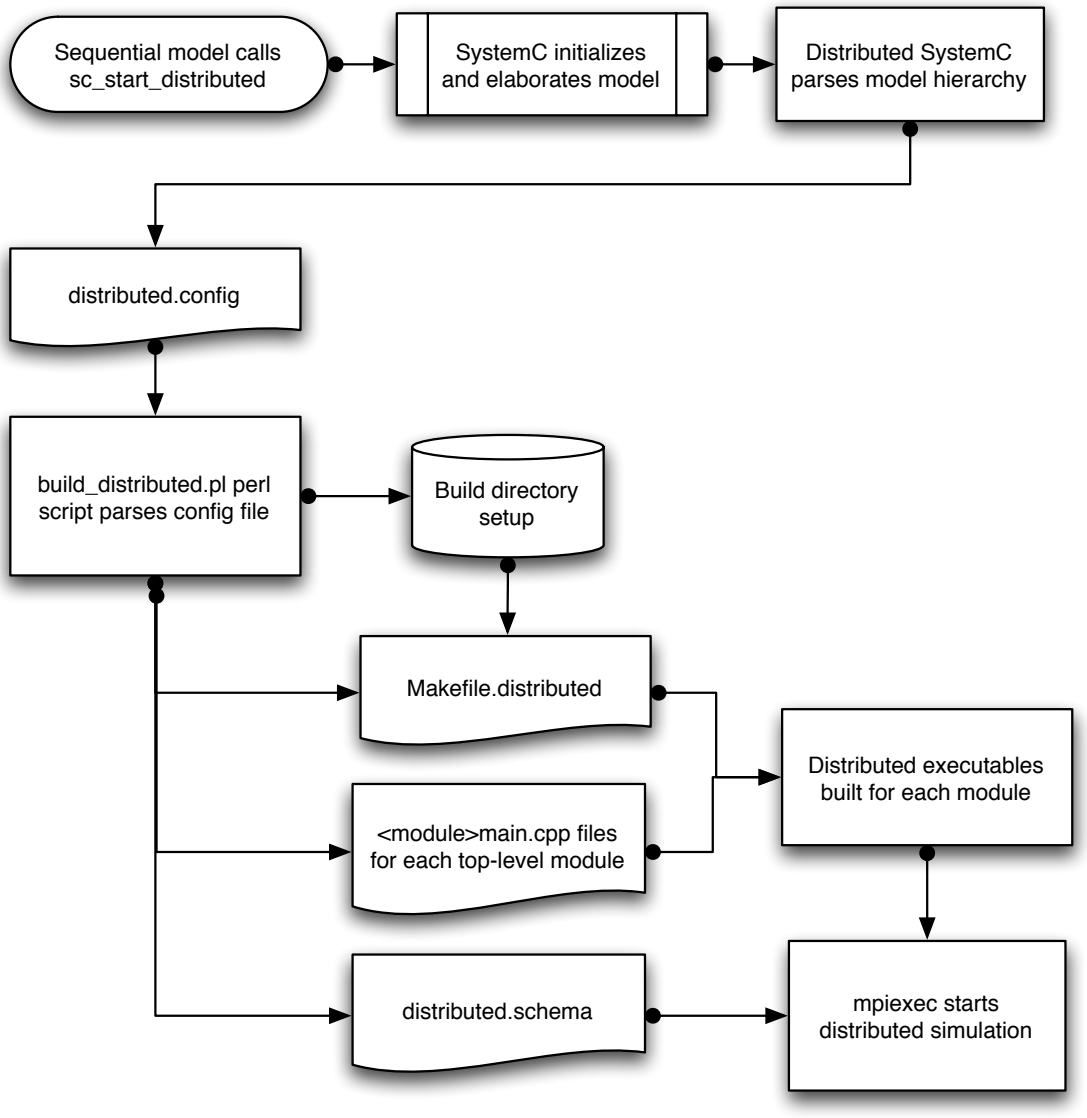

Figure 4.4: Automated distribution process and output files

distributed.config

This file contains a specification of the top-level module and port hierarchy, and the channels that form their connections. The configure file is a set of lines of semi-colon delimited values of the form TAG; VALUE; a sample format with example values is shown in Figure 4.5. A starting file is generated by the library as it extracts sequential model information. However, a user may wish to edit or create a new configuration file. For example, one may want to pass additional arguments to module constructors.

```
SYSTEMC_HOME; /home/dcox/distributed_systemc
RESOLUTION; 1; SC_PS
UNIT; 1; SC_NS
TIME; 60; SC_NS

CLOCK; clock_name; 20; SC_NS; 10; SC_NS; true

MODULE; my_module; my_module_name; arg1; arg2
PORT; channel_type; data_type; channel_name
```

Figure 4.5: Sample distributed.config File

It is important to note that matching port and clock names is the mechanism by which remote modules connect their ports. Again, this is all handled transparently when the automation code is used. In the case of customization, however, rules for various tags in `distributed.config` are as follows:

MODULE & PORT Specifies the structure and connection of each top-level module.

- Each MODULE declaration must be followed by PORT declarations for all of its connections.
- The PORT declarations must be in the same order as they are declared in the module source code.

- argN are arguments for the module constructor.
- All the PORTs for a module must be declared before the next MODULE tag.
- channel_type may only be sc_signal or sc_buffer at this time, but others may be allowed in the future.
- data_type can be any C++, SystemC, or valid user datatype.

CLOCK Defines clocks that are used by two or more top-level modules.

- For each CLOCK tag, only the name is required. The other arguments are the same as those passed to an sc_clock constructor and are optional. If the start time argument is used, however, make sure to include the time unit.
- Full automation does not support start times other than SC_ZERO_TIME, or starting edges other than positive. The config file should be edited manually if these options are required.

RESOLUTION & UNIT These tags are for the SystemC default time resolution and default time unit as defined by the SystemC specification [52].

TIME This is the length of simulated time for which to execute the model. If omitted, the simulation will execute until there are no more pending events.

module_main.cpp

As the perl script parses `distributed.config`, it will create a C++ source file for each top-level module. This is a very simple SystemC program with an `sc_main` function to instantiate and connect the top-level module in question to the appropriate distributed channels.

Makefile.distributed

A makefile, based on a template included with the library, is created by the perl script. Targets are generated for each executable associated with each top-level module. The resulting executables are named `<module_name>_main.x`.

distributed.schema

Finally, a file like Figure 4.6 is generated to instruct MPI how to launch each individual process. Though not required by an MPI implementation, the MPI specification [38] sets forth guidelines for how such a schema file should be used. LAM and other popular MPI implementations include the `mpiexec` program that understands schema files. The Distributed SystemC automation script will assign modules to processors in a round-robin manner (noted by c*N* in Figure 4.6). For further customization, or to add command-line arguments, the schema file may be edited manually.

```
c0 /home/dcox/model/moduleA_main.x
c1 /home/dcox/model/moduleB_main.x
c2 /home/dcox/model/moduleC_main.x
. . .
```

Figure 4.6: Sample distributed.schema File

4.3.3 Summary

To summarize the major steps when `sc_start_distributed` is called:

1. Distributed SystemC generates `distributed.config` based on the object hierarchy of the sequential model, then calls the perl script `build_distributed.pl`.

2. Perl script reads the configuration file, builds a `<module>_main.cpp` file for each top-level module.

3. Perl script creates a Makefile, MPI application schema file, and calls `make`.

4. The command `mpiexec -configfile distributed.schema` kicks off a distributed simulation.

Chapter 5

Evaluation Methods

5.1 Platform

The RITSim Distributed SystemC library was developed on Linux using the open source implementation of MPI from Indiana University, LAM-MPI [12].

Distributed models execute on a commodity cluster of dual AMD Athlon MP 2600 processor machines (1 GB RAM per processor) connected via a gigabit ethernet switch. Performance results are gathered at speeds of 100 and 1000 Mb/s. User access is restricted during collection of performance data.

5.2 Functional Verification

Before judging the performance of the RITSim Distributed SystemC implementation, we must ensure that it is functionally sound. A goal of the simulator is to restrict SystemC functionality only where necessary for distributed modules. Users should be able to employ most of the SystemC constructs that they use in sequential models.

To this end, numerous simple examples were utilized throughout the development of the library. Many of these test programs are modifications of the examples included with the SystemC 2.1 distribution. Calls to `sc_start_distributed` are added to allow for automatic distribution of the sequential models. All individual steps of the automation process are tested. Functionality is confirmed by examining the outputs of three different executions of

the same model:

1. Sequential model linked against original SystemC 2.1 library
2. Distributed model linked against Distributed SystemC, but run as a single process (uses `sc_start` instead of `sc_start_distributed`)
3. Distributed model linked against Distributed SystemC, run in parallel on the target cluster.

While minute adjustments were made during development to test individual features (clocks, datatypes, *etc.*), five examples are included in the RITSim distribution that encompass a good set of SystemC features. They can be used by editing the makefile to point to the local installation of the Distributed SystemC library. Then, a user can compile and run as normal. The examples also serve the purpose of providing templates for creating more distributed models.

Note that the distributed execution time is much longer than the original sequential models for the functional examples. This is due to the extreme simplicity of the example models. The purpose of this library is to distribute much more complex models that perform heavy processing between necessary communication and synchronization periods. The simple distributed examples are overwhelmed by communication costs. As a proof of concept, however, XMPI [60] traces are collected to show that even the simplest models can show potential for parallel computation. XMPI is a debug and profiling tool for MPI programs that creates visual representations of parallel execution.

5.3 Representative RITSim Model

The primary purpose of a distributed simulation is to reduce simulation time for a particular application. In the case of RITSim, that application is a complex microarchitectural simulator. Since the RITSim model is not yet complete, we create a model that is representative of the target CPU microarchitecture.

5.3.1 CPU Test Model

Making use of Semeraro's work [56], the model architecture of Figure 5.1 is abstracted in SystemC. The model is constructed following the rules for automatic distribution. There are six different top-level modules for distributed execution – integer unit, floating point unit, front end, load/store unit, L2 cache, and main memory. No functionality is actually implemented, and the model tracks no data. Rather, skeleton modules are created with the requisite distributed inputs and outputs.

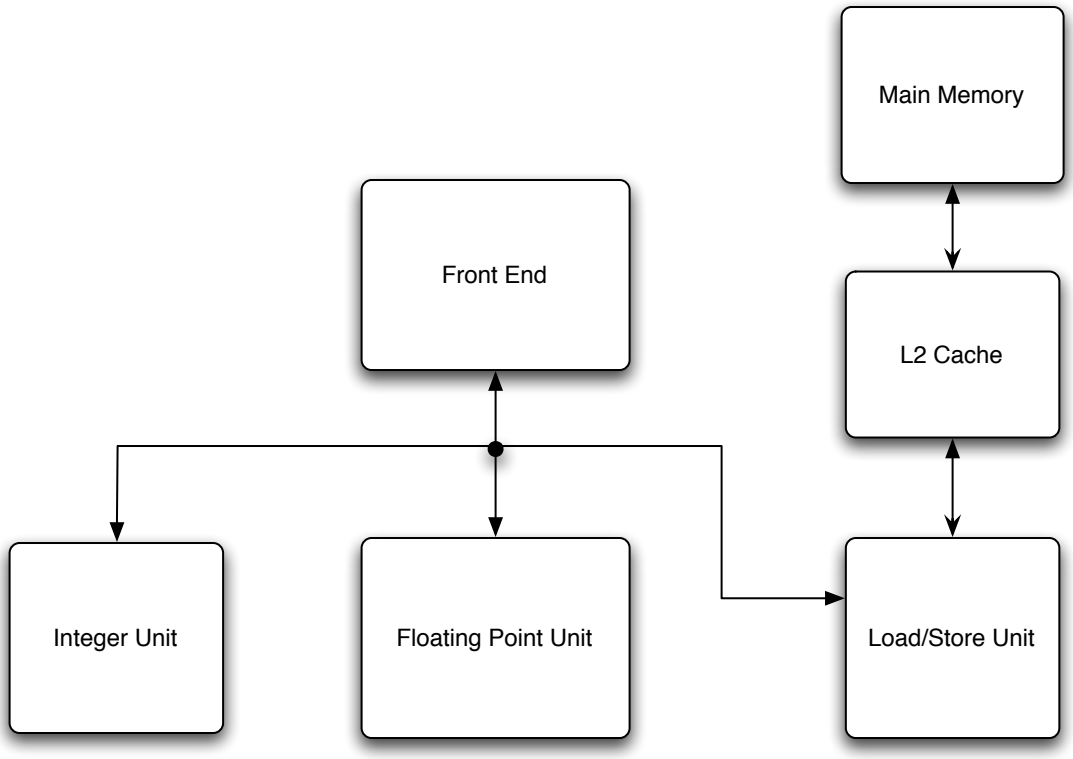

Figure 5.1: Skeleton architecture of representative model. Six top-level modules communicate via MPI at rates based on SimpleScalar analysis. From [56]

These modules are encapsulated in a `sc_main` function and connected using `sc_signal` objects. A call to `sc_start_distributed` performs the automatic distribution routines and launches all necessary processes. Each of the six modules becomes an LP distributed across

the cluster. Thus, the ideal speedup over the sequential model is six.

5.3.2 Configuration

In order to accurately represent the simulation of such an architecture, a modified version of SimpleScalar [3] developed in [56] was characterized. The simulated target architecture follows that of Figure 5.1. For a number of benchmarks, the average communication rates for each of the interconnection signals was found. Furthermore, the code was profiled to discover how much work is performed in each module.

This information, along with other setup parameters is input to the sample model via a configuration file similar to Figure 5.2. Each module performs "fake" work in proportion to the characterized workload. Furthermore, output ports are updated stochastically at the rates in which the SimpleScalar signals are written.

To represent variable model complexity, a scaling factor is applied to the work performed by each module. A higher scaling factor represents a more refined model, simulating accurate model details and the gathering of more data.

5.3.3 Running the CPU Test Model

The configuration file must be accessible by each of the distributed modules. The filename could be hardcoded, but instead it is just added to the distributed.config file. Then, the filename is automatically passed from the sequential executable to each node.

A final optional command-line parameter is included to allow the user to run selected parts of the automatic distribution mechanism. This was intended as a development/debug convenience, but also provides an easy way to execute sequentially, or to prevent the system from rebuilding the distributed configuration file. The input is an integer formed by adding together all the steps that are desired. It is then passed through to `sc_start_distributed`. Table 5.1 lists each of these codes. Thus to perform a distributed simulation:

```
% testCPU.x [inifile] [code]
```

```
[main]
timeToRun = 500
benchmark = average
workload = workload;
seed = 0x45FF3AF
workscale = 1000;
ratescale = 1;

[Average]
ch1Rate = 1.405051929
ch2Rate = 0.38066428
ch3Rate = 2.174293013
ch4Rate = 18.09198233

[workload]
work1 = 1.173055155
work2 = 4.055616877
work3 = 1.173055155
work4 = 3.829836912
```

Figure 5.2: Sample CPU.ini File (truncated)

5.4 Performance Metrics and Instrumentation

To evaluate the performance of a parallel simulator, execution time and speedup over a sequential model are the primary concerns. For better understanding of the interaction between communication and model computation, the following performance metrics are tracked:

- *Total execution time* - For a sequential model, the elapsed wall clock time to run the simulation. For a distributed model, the maximum of all such times across top-level modules.
- *Speedup* - Ratio of sequential execution time to distributed execution time.

Action to Perform	Defined constant	Hex Code
Nothing	SC_MPI_NONE	0x00
Create distributed config file	SC_MPI_CONFIG	0x01
Setup the distributed build directory	SC_MPI_SETUPDIR	0x02
Write main.cpp files for distributed processes	SC_MPI_MAINCPP	0x04
Write the distributed Makefile	SC_MPI_MAKEFILE	0x08
Create the MPI application schema	SC_MPI_SCHEMA	0x10
Build the distributed model	SC_MPI_MAKE	0x20
Execute the distributed model	SC_MPI_EXEC	0x40
Execute the serial model	SC_MPI_SERIAL	0x80

Table 5.1: Distributed Execution Codes

- *Blocked time* - Waiting inside a blocking MPI routine (such as `MPI_Wait` or `MPI_Recv`) for communication to complete.
- *Initialization time* - Time required for a parallel model to exchange information among distributed modules, including initial values of signals and communication topology.
- *Percent blocked time* - For the entire distributed simulation, this is the ratio of the sum of each module's blocked times and the sum of each module's individual execution times.
- *Synchronization cycles* - the number of times the distributed model must enter a global synchronization phase.

All times are calculated using calls to `MPI_WTime` that return the wall clock time. The difference of subsequent invocations is used, for example, to track the time until a blocking MPI routine returns. The overhead of these calls should be negligible compared to all other kernel activity. Nevertheless, all such instrumentation code is contained within `#ifdef` blocks and are only compiled in the development evaluation library.

Chapter 6

Results

6.1 Functional Results

Results from the simple SystemC examples show correct functionality and can be reproduced using the source included with the distribution. The more interesting result from these tests, however, is the promise of parallelism available in even the simplest model.

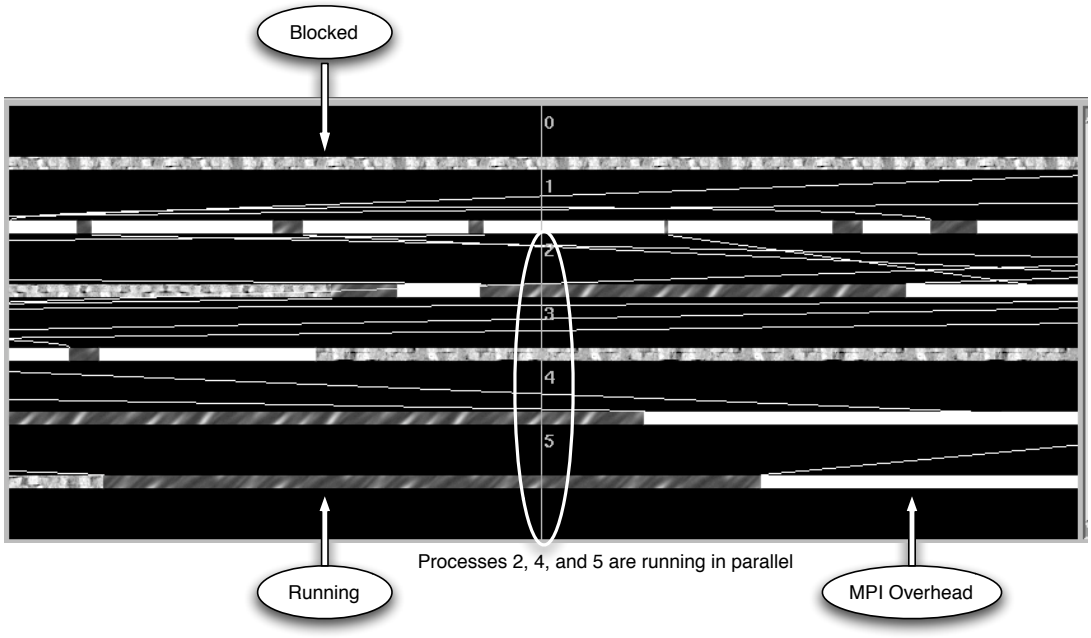

Figure 6.1: XMPI Trace for section for pipe example (modified for greyscale)

Figure 6.1 shows part of an XMPI visual trace of the distributed "pipe" example. The different patterns indicate running, MPI overhead, and blocked states for each LP. The white lines indicate messages that are sent from one process to another.

Though there is an abundance of overhead for such a simple model, there are clearly times during which multiple processes are executing in parallel. While most of this execution is actually in the SystemC kernel (not the modeled hardware thread), it shows potential for a more complex application to exploit parallelism. A pipeline that does real simulation work could prove to be a very successful application of SystemC PDES simulation.

6.2 CPU Model Performance Results

Using the metrics described above, we track the speedup over a sequential model for varying degrees of model complexity by scaling the dummy workload. Trends in speedup are plotted along with the percent of total execution time during which the module is blocked for communication. Figure 6.2 shows results for the testCPU model at 100 Mb/s, and Figure 6.3 is with 1000 Mb/s ethernet. Note that fewer data points were taken for 1000 Mb/s.

For small workloads, the synchronization overhead outweighs any parallel time savings. However, there is a break-even point at a workscale around 6300 for both 100 Mb/s and Gigabit ethernet. This is an estimate computed by a linear solve between workload 6000 and 6500 for speedup = 1. Beyond the break-even point, parallel execution provides increased performance as the percentage of blocked time decreases. Speedup asymptotically approaches the theoretical maximum of six, but actually levels off shortly after five as communication overheads becoming a limiting factor.

Even for the smallest workload, the initialization overhead is less than one ten-thousandth of one percent of total execution time. While this number would increase for models partitioned into more top-level modules, it is safe to conclude there is negligible initialization penalty for the RITSim application.

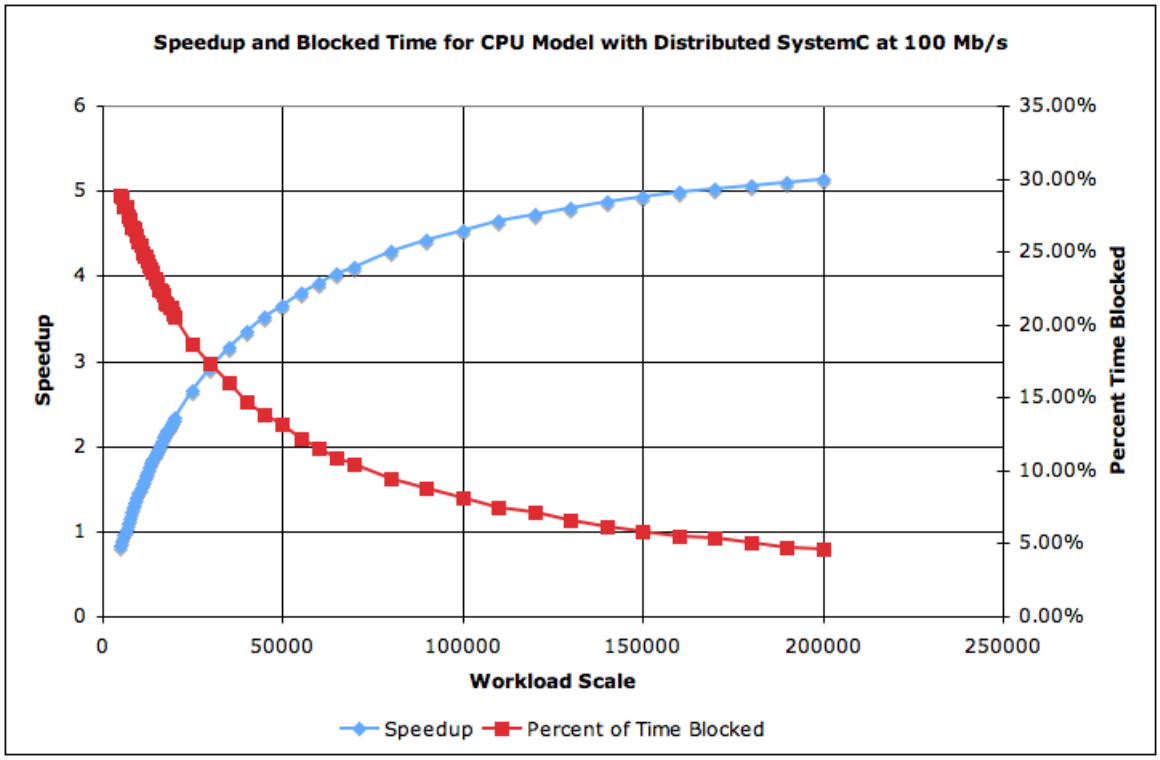

Figure 6.2: Results for 100 Mb/s

6.3 Analysis

For the type of application RITSim targets, the synchronous conservative method of this Distributed SystemC kernel shows promise for reasonable speedup. Even though lookahead is not provided or used in the synchronization, a speedup of around 5X is possible for high workloads. As a real microarchitecture model is implemented to perform detailed, accurate, and complex simulations, high workloads are a necessity.

One main reason that this particular application may perform well – and others may not – has to do with the model partitioning. The custom test application is well balanced. Each skeleton module has the same structure, and therefore similar execution within the SystemC kernel. Furthermore, well-balanced workloads allow for maximum exploitation of parallelism. This is one of the primary goals in any parallel programming exercise; if

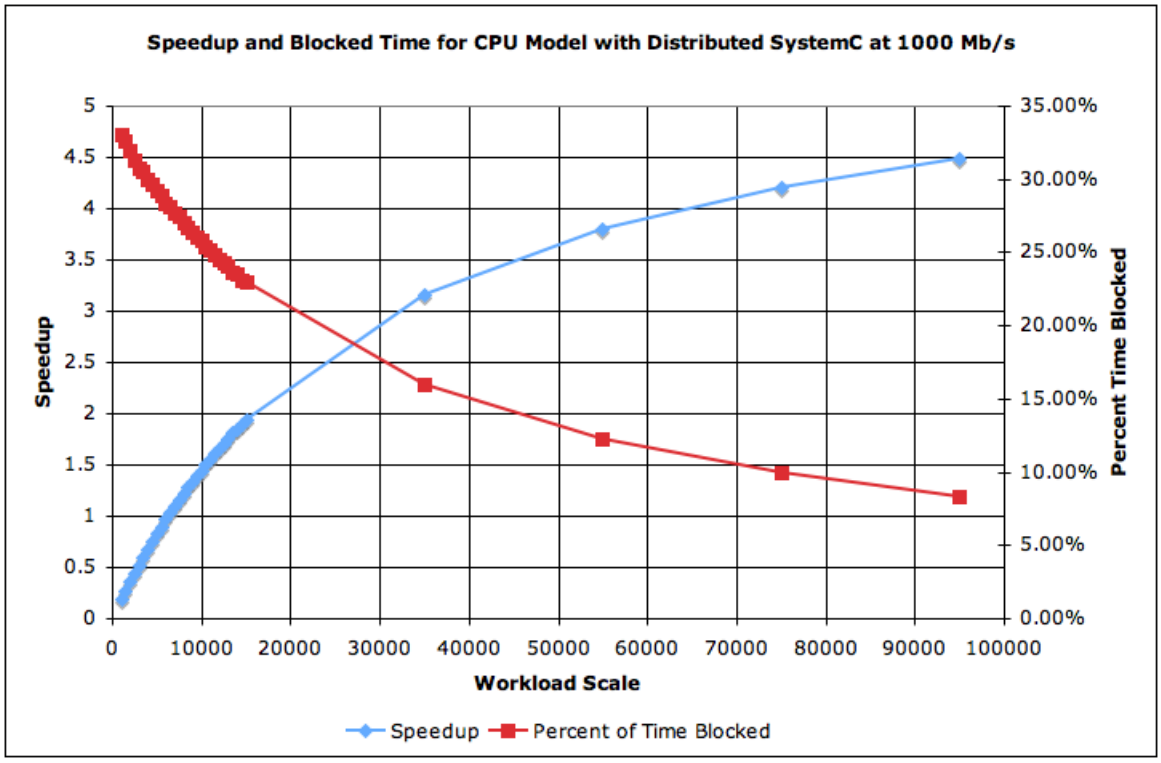

Figure 6.3: Results for 1000 Mb/s

the model designer partitions and balances the model appropriately, she may expect better performance.

One concern, however, is the large amount of synchronization required for this no-lookahead model to execute. Although the test model was only run for 500 simulated clock cycles, 2600 synchronization periods were needed. The simulation creeps along, blocking some processes and advancing one delta cycle at a time until modules are able to perform worthwhile work. More complex model hierarchies would result in even more synchronization, and worse performance. Adding lookahead could eliminate some of this traffic, allowing some distributed processes to continue unhindered.

The multitude of small MPI synchronization messages reveals network latency – not throughput – as the performance gate. This is noted by the unexpected result that stepping up to Gigabit ethernet has a negligible effect on the break-even point. Each small

synchronization message is packed individually and the MPI overhead code is called many times throughout the simulation. While this result seems at first quite negative, it also has a positive spin. For the type of synchronization currently implemented, an inexpensive 100 Mb/s ethernet network supports the application. If the application were instead dominated by message traffic across distributed channels sending large data streams, we would expect more dependence on the transmission medium. As it stands, messaging between distributed components does not saturate the network, even at low speeds.

Chapter 7

Conclusions

7.1 Summary

The goal of this work was to investigate and develop an extended SystemC simulation library for distributing parallel models. The need for this simulation infrastructure should now be apparent; tighter power and performance constraints demand increasingly complex models. SystemC is proving itself as a powerful hardware simulation tool expressed in the ubiquitous software terms of C/C++. Parallel discrete event simulation is a natural extension to SystemC's flexible and modular construction.

Overall, we have provided a viable solution for parallel simulation with SystemC. An analysis of the results shows a functionally accurate simulator with potential for reasonable speedup in a microarchitecture application like RITSim. The ideal goals of the library, discussed in Section 4.1 were met to varying degrees of success:

- *Speed* - Section 6.2 clearly shows decreased execution time for our CPU test model of a representative microarchitecture. However, large synchronization penalties will force a higher break-even point for poorly balanced models.

- *Transparency* - Following only a few simple restrictions for top-level modules, designers can use the library's tools to automatically extract, build, and run a parallel model from sequential code. This enables a good starting point, but customization

is available at every level to allow the expert model writer to tailor the parallel execution to her needs. Although the user must partition the model manually, the tools presented in this work try to make the rest of the process as simple as possible.

- *Accuracy* - Care has been taken to ensure accuracy to the delta cycle level. Numerous examples using a wide set of SystemC constructs were regressed to ensure deterministic operation compared to sequential versions.

- *Compatibility* - While not *any* sequential SystemC code can be automatically parallelized, models with well-defined modules should be fairly easy to adapt. Once the structure is in place, selecting between sequential and parallel execution can be performed with a runtime parameter, or just by changing the call to `sc_start`. Furthermore, the library installation procedure automatically determines whether the appropriate tools (MPI, perl, *etc.*) are installed on the target nodes. For example, if MPI is not available, no code for distribution will compile, leaving the user with a library identical to the original SystemC distribution.

- *Portability* - Only free, open source tools, and commodity hardware components were used in the development and testing of this software. However, it was only tested on the Linux 2.4.21 SMP kernel, and the automated configure and install process may not be appropriate for other architectures. This is a small inconvenience that can be patched with better study of the GNU autotools [23].

- *Maintainability* - Only one class (two files) of the original SystemC library was modified. Other classes are added but are kept within a new directory in the distribution for easier tracking. All code is well commented, and Chapter 4 should also serve as reference for future modifications.

7.2 Suggestions for Future Work

This Distributed SystemC endeavor provides a good starting point for the implementation of PDES into mainstream hardware simulation as envisioned through RITSim. While results look promising, synchronization overheads and partitioning will be the bottlenecks for truly fast and transparent parallel simulation. As model complexity continues to grow, some modifications and extensions of our method could make Distributed SystemC an even more viable option.

7.2.1 Including Lookahead

Though this work was designed to operate correctly without any lookahead, nearly every work mentioned in Chapter 3 stresses the importance of good lookahead to performance. The first recommended modification to this code would allow users to specify lookahead within their modules. Optional constructs could be added to SystemC to provide lookahead values for individual signals, threads, or entire modules. An advanced design would incorporate these calls to exploit any of the model's predictive behavior and reveal parallelism.

Once the structure for these additional SystemC constructs is in place, the current simulation kernel can handle much of the rest of this work. Modifications need only be made to add any available lookahead values to outbound messages. Null messages can also be sent at will, adding the Chandy-Misra approach [16] to the synchronous algorithm in a hybrid format like [7]. With more lookahead, the global algorithm becomes more like a backup to prevent deadlock.

7.2.2 Improved Global Synchronization

In parallel, a better global synchronization algorithm can be developed to reduce the performance dependency on well-balanced models. The global reduction is essentially an algorithm to compute global virtual time (GVT) - the lowest safe simulation time across the entire distributed model. Research into optimistic PDES reveals numerous algorithms

for computing GVT, each with its own tradeoffs [9]. The current calculation happens inline with the simulation kernel's handling of events. Perhaps a more appropriate approach would include a separate thread per LP to help provide a global snapshot of the simulation space without completely interrupting modules performing real work. Since our results indicate we are not saturating the network bandwidth with current messaging and synchronization, a more aggressive algorithm to advance simulation time may offer "free" speedup.

7.2.3 Automatic Partitioning and Mapping

Automatically partitioning and mapping a sequential problem into a parallel program is not always an easy task. In the case of PDES for SystemC, we have chosen to defer that task for the time being. However, the development of this library has led to an understanding of the hierarchy of the current open-source implementation of SystemC. The infrastructure already exists for understanding the topology of the simulation for a sequential model. An automatic partitioning algorithm would need to traverse that topology and have some way to analyze its complexity. Whether this information is obtained from the user, through code profiling, or by another method is a design decision for such a project. Once the automatic partitioning algorithm understands the intricacies of the model, it should use knowledge of the simulation host(s) to optimize balance and mapping. Different applications will find optimal speedup with varying granularity of parallelism. A good partitioning method will make intelligent tradeoffs transparently to ensure high performance.

7.2.4 Integration into SystemC

Finally, we hope PDES SystemC code can be integrated with existing SystemC simulators. This will require working with the Open SystemC Initiative (OSCI) [53] on future releases. Portability concerns need to be addressed but should not be a problem with the current implementation. Once the benefits of parallel discrete event simulation are merged

with SystemC's ease of use, researchers will hold a powerful tool for advancing microarchitecture design. Furthermore, the flexibility of SystemC guarantees that a wide variety of modeling and simulation applications can benefit from parallel execution. Given the appropriate model characteristics, Distributed SystemC has potential to spread well beyond microarchitecture research.

Bibliography

[1] RITSim Microarchitectural Simulator. http://www.ce.rit.edu/ gpseec/RitSimulator.pdf.

[2] Alexandre Amory, Fernando Moraes, Leandro Oliveira, Ney Calazans, and Fabiano Hessel. A Heterogeneous and Distributed Co-Simulation Environment. In *Proceedings of the 15th Symposium on Integrated Circuits and Systems Design*, page 115. IEEE Computer Society, 2002.

[3] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[4] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A Parallel Simulation Environment for Complex Systems. *Computer*, 31(10):77–85, 1998.

[5] Rajive L. Bagrodia. Language Support for Parallel Discrete-Event Simulations. In *Proceedings of the 26th Winter Simulation Conference*, pages 1324–1331. Society for Computer Simulation International, 1994.

[6] Rajive L. Bagrodia. Perils and Pitfalls of Parallel Discrete-Event Simulation. In *Proceedings of the 1996 Winter Simulation Conference*, pages 136–143. ACM Press, 1996.

[7] Rajive L. Bagrodia and Mineo Takai. Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages. *IEEE Transactions on Parallel and Distributed Systems*, 11(4):395–411, April 2000.

[8] William L. Bain and David S. Scott. An algorithm for time synchronization in distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 30–33. Society for Computer Simulation International, February 1988.

[9] Steven Bellenot. Global Virtual Time Algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 122–127. Society for Computer Simulation International, January 1990.

[10] T. D. Blanchard, T. W. Lake, and S. J. Turner. Cooperative Acceleration: robust conservative distributed discrete event simulation. In *PADS '94: Proceedings of the eighth workshop on Parallel and distributed simulation*, pages 58–64, New York, NY, USA, 1994. ACM Press.

[11] R. E. Bryant. Simulation of Packet Communication Architecture Computer Systems. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977.

[12] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[13] Wentong Cai and Stephen J. Turner. An Algorithm for Distributed Discrete-Event Simulation–the Carrier Null Message Approach. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 3–8. Society for Computer Simulation International, January 1990.

[14] Roger D. Chamberlain. Parallel Logic Simulation of VLSI Systems. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, pages 139–143. ACM Press, 1995.

[15] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(4):198–206, 1981.

[16] K.M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, September 1979.

[17] K.M. Chandy and B. Sherman. The conditional event approach to distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pages 93–99. Society for Computer Simulation International, March 1989.

[18] Matthew Chidester and Alan George. Parallel Simulation of Chip-Multiprocessor Architectures. *ACM Transactions on Modeling and Computer Simulation*, 12(3):176–200, 2002.

[19] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 55. IEEE Computer Society, 2003.

[20] Bruce A. Cota. A Framework for Automatic Lookahead Computation in Conservative Distributed Simulations. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 56–59. Society for Computer Simulation International, January 1990.

[21] Karsten Einwich, Christoph Grimm, Peter Schwarz, and Klaus Waldschmidt. Mixed-Signal Extensions for SystemC. In *Extended Papers: Best of FDL'02*. Kluwer Academic Press, April 2003.

[22] Harry Eisenbise. RITSim: Cache Modeling and Simulation. Master's thesis, Rochester Institute of Technology, April 2005.

[23] Inc. Free Software Foundation. Autoconf. Available at http://www.gnu.org/software/autoconf.

[24] Richard M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[25] Richard M. Fujimoto. Parallel and Distributed Discrete Event Simulation: Algorithms and Applications. In *Proceedings of the 1993 Winter Simulation Conference*, pages 106–114. ACM Press, 1993.

[26] Richard M. Fujimoto. Parallel and Distributed Simulation. In *Proceedings of the 1999 Winter Simulation Conference*, pages 122–131. ACM Press, 1999.

[27] Richard M. Fujimoto. Parallel and Distributed Simulation Systems. In *Proceedings of the 2001 Winter Simulation Conference*, pages 147–157. IEEE Computer Society, 2001.

[28] Richard M. Fujimoto. Distributed Simulation Systems. In *Proceedings of the 2003 Winter Simulation Conference*, pages 124–134. IEEE Computer Society, 2003.

[29] Alan D. George and Steven W. Cook, Jr. Distributed Simulation of Parallel DSP Architectures on Workstation Clusters. *Simulation*, 67(2):94–105, 1996.

[30] Sylvain Girbal, Gilles Mouchard, Albert Cohen, and Olivier Temam. DiST: A Simple, Reliable and Scalable Method to Significantly Reduce Processor Architecture Simulation Time. *SIGMETRICS Performance Evaluation Review*, 31(1):1–12, 2003.

[31] Bojan Groselj and Carl Tropper. The time-of-next-event algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 25–29. Society for Computer Simulation International, February 1988.

[32] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, Boston, 2002.

[33] Masachika Hamabe. SystemC Page. Available at http://www5a.biglobe.ne.jp/ hamabe/SystemC.

[34] Avery Design Systems Inc. SimCluster - Distributed Parallel Simulation. Available at http://www.avery-design.com, 2003.

[35] David R. Jefferson. Virtual Time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[36] Vikas Jha and Rajive L. Bagrodia. Transparent Implementation of Conservative Algorithms in Parallel Simulation Languages. In *WSC '93: Proceedings of the 25th Winter Simulation Conference*, pages 677–686, New York, NY, USA, 1993. ACM Press.

[37] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222. ACM Press, 2002.

[38] Argonne National Laboratory. MPI - The Message Passing Interface Standard. Available at http://www-unix.mcs.anl.gov/mpi.

[39] Lijun Li, Hai Huang, and Carl Tropper. DVS: An Object-Oriented Framework for Distributed Verilog Simulation. In *Proceedings of the Seventeenth Workshop on Parallel and Distributed Simulation*, page 173. IEEE Computer Society, 2003.

[40] Boris D. Lubachevsky. Efficient Distributed Event-Driven Simulation of Multiple-Loop Networks. *Communications of the ACM*, 32(1):111–123, 1989.

[41] Dragos Lungeanu and C.-J. Richard Shi. Distributed Simulation of VLSI Systems via Lookahead-Free Self-Adaptive Optimistic and Conservative Synchronization. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 500–504, Piscataway, NJ, USA, 1999. IEEE Press.

[42] Dragos Lungeanu and C.J. Richard Shi. Parallel and Distributed VHDL Simulation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 658–662. ACM Press, 2000.

[43] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hllberg, Johan Hgberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[44] Edward Mascarenhas, Felipe Knop, and Vernon Rego. Minimum Cost Adaptive Synchronization: Experiments with the ParaSol System. In *Proceedings of the 29th Winter Simulation Conference*, pages 389–396. ACM Press, 1997.

[45] Haruyuki Matsuo, Shigeru Imafuku, Kazuhiko Ohno, and Hiroshi Nakashima. Shaman: A Distributed Simulator for Shared Memory Multiprocessors. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, pages 347–355. IEEE Computer Society, 2002.

[46] Gokhan Memik, Mahmut T. Kandemir, and Ozcan Ozturk. Increasing Register File Immunity to Transient Errors. In *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 586–591, Washington, DC, USA, 2005. IEEE Computer Society.

[47] Jayadev Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[48] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Mark D. Hill, David A. Wood, Steven Huss-Lederman, and James R. Larus. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 8(4):12–20, 2000.

[49] E. Naroska. Parallel VHDL Simulation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 159–165. IEEE Computer Society, 1998.

[50] David M. Nicol. Principles of Conservative Parallel Simulation. In *WSC '96: Proceedings of the 28th Winter Simulation Conference*, pages 128–135, New York, NY, USA, 1996. ACM Press.

[51] Daniel Ortiz-Arroyo, Ben Lee, and Chansu Yu. EMSim: An Extensible Simulation Environment for Studying High Performance Microarchitectures. In *Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics*, volume 5, 2002.

[52] OSCI. Functional Specification for SystemC 2.0 (Update for SystemC 2.0.1). Available at http://www.systemc.org, 2002.

[53] Open SystemC Initiative (OSCI). SystemC Version 2.0 User's Guide. Available at http://www.systemc.org, 2002.

[54] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60. ACM Press, 1993.

[55] Paul F. Reynolds, Jr. A Spectrum of Options for Parallel Simulation. In *Proceedings of the 20th Winter Simulation Conference*, pages 325–332. ACM Press, 1988.

[56] Greg Semeraro. *Multiple Clock Domain Microarchitecture Design and Analysis*. PhD thesis, University of Rochester, August 2003.

[57] Kevin Skadron, Margaret Martonosi, David I. August, Mark D. Hill, David J. Lilja, and Vijay S. Pai. Challenges in Computer Architecture Evaluation. *IEEE Computer*, 36(8), August 2003.

[58] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-Aware Microarchitecture: Modeling and Implementation. *ACM Transactions on Architecture and Code Optimization*, 1(1):94–125, 2004.

[59] Mario Trams. Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead. Available at http://www.digital-force.net, February 2004.

[60] Indiana University. XMPI – A Run/Debug GUI for MPI. Available at http://www.lam-mpi.org/software/xmpi.

[61] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[62] Kenneth R. Wood and Stephen J. Turner. A generalized carrier-null method for conservative parallel simulation. *SIGSIM Simul. Dig.*, 24(1):50–57, 1994.

[63] David Wu, Eric Wu, Johnny Lai, Adras Varga, Y. Ahmet Sekercioglu, and Gergory K Egan. Implementing MPI Based Portable Parallel Discrete Event Simulation Support in the OMNeT++ Framework. In *Proceedings of the 14th European Simulation Symposium (ESS '02)*, pages 243–248. SCS Europe, October 2002.

[64] S. Xanthos, A. Chatzigeorgiou, and G. Stephanides. Energy Estimation with SystemC: A Programmer's Perspective. In *7th WSEAS International Conference on CIRCUITS*, July 2003.

[65] Edward D. Lazowska Yi-Bing Lin and Jean-Loup Baer. Conservative Parallel Simulation For Systems With No Lookahead Prediction. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 144–149. Society for Computer Simulation International, January 1990.