Algorithms for Parallel Simulation of Large-Scale DEVS and Cell-DEVS Models

By

Qi Liu, B. Eng., M. A. Sc.

A thesis submitted to the Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering (OCIECE)
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada

September 2010

The undersigned hereby recommends to the Faculty of Graduate Studies and Research acceptance of the thesis

Algorithms for Parallel Simulation of Large-Scale DEVS and Cell-DEVS Models

Submitted by Qi Liu

In partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering

Thesis Supervisor
Dr. Gabriel A. Wainer

External Examiner
Dr. Richard M. Fujimoto

Chair, Department of Systems and Computer Engineering
Dr. Howard M. Schwartz

Carleton University

September 2010

Dedicated with all my love

to

my loving wife, Sara,

and to

my little son, Jed.

"Let us … restrict ourselves to simple structures whenever possible and to avoid in all intellectual modesty 'clever constructions' like the plague."

Edsger W. Dijkstra,
Notes on Structured Programming,
In Pursuit of Simplicity,
1969 - 1970.

Abstract

The **Discrete Event System Specification (DEVS)** provides a general methodology for hierarchical construction of reusable models in a modular way and has been used to simulate sophisticated systems in a variety of domains. This dissertation addresses software design and performance issues that arise in parallel simulation of large-scale DEVS-based models on both multiprocessor clusters and chip-multiprocessor architectures.

The **Time Warp (TW)** mechanism is the most well-known optimistic synchronization protocol for **Parallel Discrete-Event Simulations (PDES)**. With the increasing scale and complexity, TW simulations face new challenges in terms of excessive memory consumption and operational overhead. In an effort to alleviate these problems, a novel **Lightweight Time Warp (LTW)** protocol is proposed for efficient optimistic parallel DEVS simulation on multiprocessor clusters. By exploring the intrinsic computational properties of DEVS-based simulations, the LTW protocol allows purely optimistic parallel simulation to be driven by only a few full-fledged TW **Logical Processes (LPs)**, while most of the LPs are set free from the burden of TW execution. The experimental results indicate that simulation performance can be improved significantly in various aspects, including shortened execution time, reduced memory footprint, lowered operational overhead, accelerated event queue operations, facilitated process migration, and enhanced system stability and scalability.

To address the limitations of microprocessor performance, the industry is moving towards multicore chip-multiprocessor designs. As a latest example of this trend, the IBM Cell processor has attracted a growing interest from the modeling and simulation community. However, general-purpose PDES on such platform requires innovative redesign of existing algorithms in return for better simulation performance. To this end, a new computing technique called **Multicore Acceleration of DEVS Systems (MADS)** is developed for high-performance parallel DEVS simulation on the Cell processor, combining multi-grained parallelism and various optimizations to overcome the major performance bottlenecks, while hiding, to a great extent, the technical details of multicore programming from general users. Through the concept of LP virtualization, the MADS technique explicitly exploits the massive data- and event-level parallelism inherent in the simulation, making the achievable

performance gain more deterministic and predictable than the traditional LP-oriented approaches. Promising results have been produced in the experiments, demonstrating that the MADS technique can be used to accelerate both memory-bound and compute-bound computational kernels in demanding parallel DEVS simulations. The proposed technique not only allows a broad community of DEVS users to tap the potential of the Cell processor with a minimal knowledge of the multicore execution environment, but also makes it possible to integrate cluster-based parallel simulation with multicore-accelerated parallel simulation on hybrid supercomputers.

Acknowledgements

This work would not have been possible without the ceaseless inspiration, continuous support, and unending encouragement of my supervisor Dr. Gabriel Wainer. I am especially grateful for his commitment to providing a stimulating learning and research environment and his investment of time and effort in my professional and personal development. I also greatly appreciate Dr. Richard Fujimoto, Dr. Emil Petriu, Dr. Frank Dehne, and Dr. James Green for their willingness to be on my defense committee and for their scientific insight and expertise.

This research was supported in part by Ontario Graduate Scholarship Program (OGS), Ontario Graduate Scholarship in Science and Technology (OGSST), MITACS Accelerate Ontario Program, Canada Foundation for Innovation (CFI), Koningstein Scholarship for Excellence in Science and Engineering, Carleton University, and by IBM T. J. Watson Research Center. I would like to express my gratitude to Dr. Michael Perrone, Dr. Ligang Lu, Dr. Lurng-Kou Liu, and Dr. Daniele Scarpazza from the IBM Watson Research Center for their helpful suggestions and constructive feedback.

I would also like to thank the members of the ARS Laboratory and the Department of Systems and Computer Engineering, both past and present, as well as all my friends and colleagues at Carleton University. Special thanks go to Narendra Mehta for all kinds of timely technical assistance and Tao Yue for always being an instant message away.

Finally, I am forever indebted to my dear parents and beloved wife for their endless love, enduring patience, and kind understanding throughout the course of my studies.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ALF	Accelerated Library Framework
API	Application Programming Interface
BIF	Broadband Interface
CA	Cellular Automata
CMP	Chip Multiprocessor
CT	Precondition Syntax Tree
DEDS	Discrete Event Dynamic System
DEVS	Discrete Event System Specification
DMA	Direct Memory Access
DT	Delay Syntax Tree
EIB	Element Interconnect Bus
EIC	External Input Coupling
EOC	External Output Coupling
FC	Flat Coordinator
FEL	Future Event List
FIFO	First In, First Out
FSK	FC Synchronization Kernel
GPU	Graphics Processing Units
GVT	Global Virtual Time
IA	Imminent ID Array
IC	Internal Coupling
ISA	Instruction Set Architecture
LBTS	Lower Bound on Time Stamp
LCT	Latest state Change Time
LP	Logical Process
LS	Local Storage
LTSF	Least Time Stamp First
LTW	Lightweight Time Warp

LVT	Local Virtual Time
M&S	Modeling and Simulation
MADS	Multicore Acceleration of DEVS Systems
MIC	Memory Interface Controller
MIMD	Multiple Instruction, Multiple Data
MPI	Message Passing Interface
MTSS	Message Type-based State Saving
NC	Node Coordinator
PDES	Parallel Discrete Event Simulation
P-DEVS	Parallel DEVS
PPE	Power Processor Element
PT	Postcondition Syntax Tree
RPC	Remote Procedure Call
SDK	Software Development Kit
SEK	Simulator Event-processing Kernel
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
SPE	Synergistic Processing Elements
STL	Standard Template Library
TA	Time Array
TW	Time Warp
TWLP	Time Warp Logical Process

Chapter 1. Introduction

Computer-based **Modeling and Simulation (M&S)** has long been used as a powerful tool for cost-effective analysis, design, control, and optimization of complex dynamic systems in a broad range of domains. One category of dynamic systems is known as **Discrete Event Dynamic Systems (DEDS)**, where changes in the system state can be represented by a collection of events occurred at discrete points in time [Fis73]. Although there is no shortage of formal methods for describing DEDS, the need for a universally applicable modeling framework has been well recognized (see, e.g., [Nan81 and Ho89]). Efforts towards this goal have led to the development of several methodologies that trace their origins to *general systems theory*, which postulates that different physical systems can obey the same laws and exhibit similar patterns of behavior [Roz93]. Among these methodologies, Zeigler's **Discrete Event System Specification (DEVS)** formalism [Zei76, Zei84, Zei90a, and Zei00] is regarded as one of the most developed general-purpose M&S frameworks for DEDS [Pag94]. Based on a solid system theoretic foundation, DEVS not only allows for hierarchical construction of reusable discrete-event models in a modular way, but also provides an *abstract simulation engine architecture* that can be realized on diverse computing platforms [Zei93]. The term *simulation engine architecture* refers to a hierarchy of simulation entities and their associated algorithms that can be used to execute DEVS-representable models correctly. It is considered as *abstract* in the sense that the conceptual simulation entities may not necessarily be mapped to physical processors in a one-to-one relation [Zei90b].

According to the DEVS theory, a model is defined as a mathematical entity that consists of a hierarchy of *atomic* (behavioral) and *coupled* (structural) components. Thanks to the *closure under coupling* property, a coupled model can be expressed as an equivalent basic DEVS model, which in turn can be used in a larger system as required for hierarchical model construction [Zei84]. After over forty years of research, many extensions to DEVS have been proposed in the literature. For instance, the **Parallel DEVS** (or **P-DEVS**) formalism [Cho94] extends DEVS to improve the mechanism for handling simultaneous events, eliminating the serialization constraints existed in the original DEVS definition. The

Cell-DEVS formalism [Wai02a] defines n-dimensional cell spaces as discrete-event models where each cell is a P-DEVS atomic model, allowing for specifying both temporal and spatial relations between model components. In parallel with these theoretical developments, various DEVS-based simulation tools have been implemented, such as DEVS-C++ [Zei96], RTDEVS/CORBA [Cho03], DEVSCluster [Kim04], and DEVS/SOA [Mit09], just to mention a few. In particular, the CD++ toolkit [Wai02b] is an open-source, object-oriented M&S environment that implements both P-DEVS and Cell-DEVS formalisms using different middleware technologies on varied platforms (see, e.g., [Tro03, Chi07, Liu07, Fen08, Har08, Wai08a, Wai08b, and Wai09a]).

In order to improve the performance of discrete-event simulations, **Parallel Discrete-Event Simulation (PDES)** techniques have been used to allow for executing a single discrete-event simulation program on a parallel computer with multiple processors (or nodes[1]). A PDES system is typically constructed as a set of **Logical Processes (LPs)**, each representing a different portion of the physical system and executing on a potentially different processor in an event-driven fashion. The execution of an event at a LP may modify the state of the LP and generate new events that will be sent to other LPs. During a simulation, the LPs interact with each other exclusively by exchanging time-stamped event messages. To ensure correct simulation results, the LPs must be synchronized properly to comply with the *local causality constraint* [Fuj90], which requires each LP to process events in nondecreasing time stamp order. Errors resulting from out-of-order event execution are referred to as *causality errors*. Synchronization techniques for PDES systems are broadly classified into two categories, namely *conservative* and *optimistic*. The conservative approaches, pioneered by Chandy and Misra [Cha79], strictly avoid the possibility of processing events out of time stamp order. In contrast, the optimistic approaches, exemplified by Jefferson's **Time Warp (TW)** protocol [Jef85], allow causality errors to happen temporarily, but provide mechanisms to recover from them during the execution. Both approaches have their own merits and are being used in different applications. An extensive survey of existing PDES techniques can be found in [Fuj00].

[1] The term *node* is used interchangeably with *processor* (or *chip*) hereafter, whereas the internal processing units included in a multicore processor are referred to as *processing elements* (or *cores*).

Traditionally, PDES systems have been implemented on distributed-memory and shared-memory multiprocessor clusters [Buy99]. Recent advent of multicore **Chip Multiprocessor (CMP)** architectures has attracted significant interest from the M&S community [Olu07]. The research described here is primarily concerned with issues related to software development and performance optimization in large-scale parallel simulation of P-DEVS and Cell-DEVS models on both distributed-memory multiprocessor clusters and *heterogeneous* multicore processors [Kum05]. Specifically, this dissertation investigates new ways of efficient parallel simulation on such platforms, taking advantage of the intrinsic computational properties and the inherent parallelism of the DEVS-based simulation process. To this end, novel computing techniques, algorithms, and protocols are proposed, and advanced parallel simulation engines are developed and integrated into the CD++ environment.

In the rest of this chapter, Section 1.1 presents the research motivations and objectives. Section 1.2 highlights the major contributions. Section 1.3 summarizes the publications derived from the research. And Section 1.4 outlines the organization of this dissertation.

1.1. Research Motivations and Objectives

This research is motivated by two complementary and interrelated objectives. The first one is to address the challenges of large-scale optimistic parallel simulation of P-DEVS and Cell-DEVS models on distributed-memory multiprocessor clusters based on the TW protocol [Jef85]. The second one is to achieve efficient parallel DEVS simulation on heterogeneous CMP architectures as exemplified by the IBM Cell Broadband Engine processor [Kha05 and Che07]. The fulfillment of these two objectives would help bridge the gap between PDES algorithms developed for traditional multiprocessor clusters and those for emerging CMP architectures, allowing for combining the advantages of parallel simulation at the cluster level with the benefits of accelerated parallel simulation on each multicore node.

1.1.1. DEVS Simulation with Time Warp

Originally introduced in [Jef85], Jefferson's TW mechanism remains the best known optimistic synchronization protocol that underlies many PDES systems (see, e.g., GTW [Das94], ROSS [Car02], WARPED [Mar03], μsik [Per05], and WarpIV [Ste05]). A TW

simulation is executed by several **Time Warp Logical Processes (TWLPs)**, each of which has its own **Local Virtual Time (LVT)** and processes events autonomously. They rely on a *rollback mechanism* to recover from potential causality errors based on stored historical event and state data. Numerous techniques have been proposed to improve the efficiency of TW simulations (see, e.g., state checkpointing [Pre94, Tay00, and Fen06], event cancellation [Lin91a, Nor02, and Che09a], GVT computation [Mat93, Kan96, Fuj97, and Che05], fossil collection [You99, Vee02, and Che06], memory management [Jef90, Lin91b, and Pre95], event set implementations [Bro88, Ron93, and Tan05a], optimism control [Ste93, Sri98, and Wan09], and dynamic load management [Rei90, Gla93, and Li04]). However, these techniques often do not relate themselves to the broader context of M&S methodology in general, and thus overlook the benefits that a formal modeling framework can offer.

The TW protocol has also been studied in the context of DEVS-based simulations (see, e.g., [Chr90, Kim98, Zei00, Nut04, Nut08, and Sun08]). In a recent effort, the CD++ toolkit has been extended to support TW simulation of P-DEVS and Cell-DEVS models on distributed-memory multiprocessors using the WARPED simulation kernel as a middleware layer [Liu07]. The resulting optimistic parallel simulator, referred to as **PCD++**, addresses several important issues raised in DEVS-based TW simulations, including asynchronous state transition, messaging anomaly, and rollback at virtual time zero. While performance is one of the main concerns, most of the foregoing studies put emphasis on the applicability and correctness of the TW approach to optimistic parallel DEVS simulations.

The increasing scale and complexity of DEVS systems poses new demands on the TW protocol. With many TWLPs allocated on each available processor in a typical large-scale simulation, saving historical data in the event and state queues not only consumes an excessive amount of memory, but also raises the cost of queue operation, fossil collection, and dynamic process migration. Moreover, the conventional rollback mechanism relies solely on propagation of anti-messages to undo the effect of incorrect computation at the TWLPs, imposing a heavy burden on the communication infrastructure and, as the number of TWLPs involved in a rollback increases, impairing the performance and scalability of the entire system. These problems become especially severe when a large number of *simultaneous events* (i.e., events with exactly the same time stamp) need to be executed at each virtual time, as commonly found in large-scale, densely-interconnected, and highly-

active DEVS-based models.

Although DEVS simulation performance could be improved by gradually incorporating different TW optimizations on a case-by-case basis, this approach suffers from two major disadvantages. First, the existing TW optimization strategies, which are developed without considering the specific properties of DEVS-based simulation, can usually produce only suboptimal performance results. Secondly, combining different optimizations of various types, many with their own special operational requirements that may not be compatible with those of others, makes the integration process time-consuming and error-prone.

One of the main objectives of this research is to address the above challenges *without* complicating the synchronization algorithms unnecessarily, sacrificing potential parallelism, or introducing a noticeable extra operational overhead. This is achieved by clearly identifying the intrinsic computational properties of DEVS-based simulations and directly exploiting them from the core of the TW mechanism. The result is a new variant of the TW protocol for efficient optimistic parallel simulation of P-DEVS and Cell-DEVS models.

1.1.2. DEVS Simulation on Cell Processor

As the monolithic approach to microprocessor design reaches a point of diminishing return due to physical and practical constraints such as heat dissipation, memory latency, and gate density, a clear trend has been observed in the industry moving towards multicore CMP architectures (see, e.g., [Kon05, Kot05, McN05, and Lin08]). Previous studies suggest that *heterogeneous* CMP designs, in which different types of cores of varying size and complexity are integrated on a single die, have the potential to meet the needs of a broad spectrum of applications [Kum05, Kum06, and Mor06]. One example of such designs is the **IBM Cell Broadband Engine**, also known as the **Cell processor** [Kha05 and Che07], which has demonstrated significant potential for scientific computing [Wil06] and been used in both high-end servers [Nan07] and next-generation supercomputers [Cra08 and Bar08].

The latest Cell processor consists of nine independent cores that employ two distinct **Instruction Set Architectures (ISAs)**, including a main two-way **SMT (Simultaneous Multithreading) Power Processor Element (PPE)** that uses the 64-bit PowerPC ISA, and eight specialized co-processors called **Synergistic Processing Elements (SPEs)** that use a 128-bit **SIMD (Single Instruction, Multiple Data)** ISA. The PPE adopts a conventional

two-level cache hierarchy (32KB L1, 512KB L2) to access system main memory and provides top-level thread control for a parallel application, whereas each SPE can directly access only a private on-chip **Local Storage (LS)** of 256KB that contains both code and data (including the call stack) of an SPE thread. Data sharing is achieved mainly through software-managed, explicitly-addressed, autonomous **Direct Memory Access (DMA)** transfers, which require proper address alignment and transfer size to attain peak performance. The cores can also communicate 32-bit short messages via the on-chip **Element Interconnect Bus (EIB)** channels (e.g., mailboxes and signals). Moreover, the SPEs support both scalar and 128-bit SIMD operations that can be applied at 2, 4, 8, and 16-way granularities. All these features make the Cell processor an attractive vehicle for studying new computing techniques on the emerging CMP architectures. On the flip side, the asymmetric design of heterogeneous cores with explicit memory control increases software complexity considerably and requires innovative redesign of existing algorithms to exploit parallelism at different system levels in return for better application performance.

While the Cell processor is rapidly gaining popularity in scientific and multimedia applications (see, e.g., [Bad07a, Ged07, Pet07, and Sai07]), its potential has yet to be realized in PDES systems due to several challenging issues. First of all, most of the existing PDES techniques, developed with traditional parallel computing systems in mind, adopt a LP-oriented approach to partitioning a simulation across multiple nodes of a cluster [Fuj00], while neglecting to integrate with other forms of parallelism (e.g., data-level parallelism, memory-level parallelism, and compute-transfer parallelism) that are made available on modern multicore platforms. As a result, developing efficient PDES algorithms on the Cell processor, and on CMP architectures in general, requires a holistic approach that takes into account all parallelization options provided by the processor microarchitecture. In addition, PDES programs typically involve highly irregular, control-intensive computation with complex data dependency and unpredictable memory access pattern [Fuj90], a class of workload that is generally regarded as not well-suited for parallelization on the Cell processor [Sca09a]. Furthermore, recent advances towards facilitating software development on the Cell processor, in the form of compiler-assisted vectorization (e.g., [Eic06 and Kni07]) and middleware frameworks (e.g., [McC06 and Per07]), offer little help in parallelizing PDES systems, mainly because these techniques, applied at a lower software layer, lack the

adequate knowledge to explore application-level parallelism effectively. Several proposed programming models and strategies attempt to provide a general guidance for porting legacy applications to the Cell processor (e.g., [Kha05, McC08, and Var07]). To make the most of the Cell potential, however, simulator developers still need to explicitly handle issues such as computational kernel analysis, data layout and movement, task synchronization, and performance optimization, among others, while at the same time satisfying the various requirements of PDES execution and the underlying hardware platform.

As multicore computing emerges as the primary way of scaling processor performance [Olu05 and Asa06], there is a growing need for new PDES techniques targeting CMP architectures. Towards this goal, the research presented in this dissertation takes a formalism-based performance-centric approach to efficient parallel DEVS simulation on the heterogeneous Cell processor, addressing the aforementioned challenges in a coherent way. In addition, this dissertation also attempts to take into account several other aspects such as enhancing modeler productivity, lowering user learning curve, and integrating with cluster-based PDES techniques for future expansion and development.

1.2. Contributions

The central themes of this dissertation are to improve the performance of DEVS-based TW simulation on distributed-memory multiprocessor clusters and to achieve high-performance parallel DEVS simulation on the Cell processor. This section summarizes the key contributions made in pursuit of each of these two research objectives.

1.2.1. Lightweight Time Warp

A new variant of the TW protocol, referred to as **Lightweight Time Warp (LTW)**, is proposed that takes advantage of the intrinsic computational properties of the DEVS-based simulation process to reduce the operational overhead of TW execution in a systematic way. The LTW protocol includes the following contributions.

- The complex message flow between the LPs is characterized by a well-structured multi-phased high-level abstraction, which, besides allowing for representing the DEVS simulation process in a compact form, assists in the development of new phase-based optimization and event-scheduling algorithms.

- Several key intrinsic (model-independent) computational properties of the DEVS simulation process are clearly identified and summarized, providing the basis for developing novel optimization strategies for efficient DEVS-based TW simulations.
- Corresponding to the DEVS computational properties, a set of assumptions regarding the control of the LPs in a TW simulation is generalized that not only forms the basis for the LTW protocol, but also serves as a guideline for applying the optimization strategies to other TW-based optimistic PDES systems.
- The simulation space on each node is divided into two *conceptual* domains, namely a *TW domain* and a *LTW domain*, in order to release most of the LPs from the burden associated with TW execution. The LPs from different domains interact through a mixed-mode *interface LP*. As a result, the overall purely optimistic TW simulation is driven by only a few TWLPs, while most of the LPs are turned into lightweight processes executing at a much lower operational cost.
- An event management scheme is proposed that classifies the events into *persistent* and *volatile* types. While the persistent events are still maintained in the input and output queues, the volatile events can be safely discarded right after execution, reducing memory consumption and accelerating event queue operations. Based on this concept, an event-scheduling algorithm is developed to schedule both types of events using a set of prioritized rules.
- An aggregate checkpointing scheme is introduced that allows the lightweight LPs to delegate the responsibility of state management to the interface LP. In addition, an enhanced risk-free infrequent state-saving mechanism is proposed to reduce state-saving overhead in DEVS-based TW simulations without increasing rollback cost.
- A lightweight rollback mechanism is provided to restrict the propagation of rollbacks to the TW domains only, whereas all the lightweight LPs are no longer required to perform rollback operations, reducing rollback overhead and enhancing system stability and scalability.
- The PCD++ simulator is extended to include the LTW protocol, and the performance is evaluated quantitatively using different models of varied characteristics based on a set of 14 key metrics that are of importance in TW simulations.

1.2.2. Multicore Acceleration of DEVS Systems

A novel computing technique, referred to as **Multicore Acceleration of DEVS Systems (MADS)**, is proposed that combines multi-grained parallelism and various optimization strategies to overcome the major performance bottlenecks in demanding DEVS-based simulations on the Cell processor. The development of the MADS technique consists of the following contributions.

- Two types of typical computational kernels are extracted from general-purpose DEVS-based simulations, reflecting the major performance bottlenecks in the simulation process, as illustrated by detailed simulation profiles.
- The concept of *LP virtualization* is introduced to support flexible and efficient mapping of LPs to different processing elements of the Cell processor dynamically at runtime, improving the utilization of the heterogeneous cores, minimizing the synchronization overhead, and allowing for fine-grained dynamic load balancing.
- Two forms of event-level parallelism are identified from a *data-flow* perspective, including the *event-embarrassing parallelism* and the *event-streaming parallelism*. Unlike the LP-oriented parallelization strategy adopted in most existing PDES systems, the MADS technique explicitly exploits the fine-grained event-level parallelism that is inherent in the DEVS simulation process, making the achievable parallelism more deterministic and predictable.
- To accelerate the computational kernels, new simulation algorithms are developed to combine multi-grained parallelism at different levels of the system in a coherent way, including thread-level parallelism, data-level parallelism, event-level parallelism, data-streaming parallelism, and compute-I/O parallelism.
- Various performance optimizations are considered in the design of the parallel simulator to further streamline the kernel computation, ranging from high-level minimization of thread orchestration and synchronization overhead to low-level data alignment and code implementation.
- By taking advantage of the built-in CD++ specification language, the proposed MADS technique hides the technical complexity of multicore programming from non-expert users in the M&S of Cell-DEVS models. In addition, it provides the necessary support to assist the development of P-DEVS models on the Cell processor (e.g., in terms of

memory control and kernel orchestration services), allowing a modeler to gain performance with minimal knowledge of the multicore execution environment.

- The mechanisms for integrating the MADS technique with other cluster-based PDES techniques (both conservative and optimistic approaches) are discussed, demonstrating the feasibility and potential of combining cluster-level parallelization with multicore-accelerated DEVS systems on hybrid supercomputers.
- The methods employed in this research, such as LP virtualization and data-flow orientated exploitation of event parallelism, can also be applied to other CMP architectures and shared-memory multiprocessors. Moreover, this dissertation discusses several practical issues that of interest to other application developers who intend to port legacy software to the Cell processor.
- A new parallel simulation engine, referred to as CD++/Cell, is implemented on the Cell processor using the proposed MADS technique, and the experiments show that a significant level of performance can be achieved in simulating different Cell-DEVS environmental models.

1.3. Research Publications

Some of the results derived from this research have been published thus far, including those directly related to the two central research themes and those relevant to DEVS-based M&S in general. Following is a list of manuscripts that pertain to the LTW protocol.

- Liu, Q., and G. Wainer, "Parallel Environment for DEVS and Cell-DEVS Models". *SIMULATION*, 83(6), pp. 449-471, 2007. This paper, referred to as [Liu07] in the list of references, proposes a multi-phased high-level abstraction for describing DEVS-based simulations. The intrinsic computational properties of the DEVS-based simulation process, briefly analyzed in this paper and later refined, are summarized in Chapter 2 of this dissertation, providing the basis for the LTW protocol. Based on previous research [Liu06], this paper also includes algorithms to ensure the correctness of TW simulation of P-DEVS and Cell-DEVS models with the PCD++ simulator.
- Liu, Q., and G. Wainer, "Lightweight Time Warp – A Novel Protocol for Parallel Optimistic Simulation of Large-Scale DEVS and Cell-DEVS Models". *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time*

Applications, Vancouver, Canada, pp. 131-138, 2008. This paper, referred to as [Liu08] in the list of references, proposes the algorithms included in the LTW protocol, which are discussed in Chapter 4 of this dissertation.

- Liu, Q., and G. Wainer, "A Performance Evaluation of the Lightweight Time Warp Protocol in Optimistic Parallel Simulation of DEVS-based Environmental Models". *Proceedings of the 23rd IEEE Workshop on Principles of Advanced and Distributed Simulation*, Lake Placid, NY, pp. 27-34, 2009. Based on a set of 14 key metrics, this paper, referred to as [Liu09] in the list of references, evaluates the performance of the LTW protocol using several Cell-DEVS environmental models as benchmarks. The performance evaluation is presented in Chapter 6 of this dissertation.

The manuscripts relevant to the MADS technique are summarized as follows.

- Liu, Q., and G. Wainer, "Accelerating Large-scale DEVS-based Simulation on the Cell Processor". *Proceedings of the 2010 Symposium on Theory of Modeling and Simulation – DEVS Integrative M&S Symposium*, Orlando, FL, pp. 191-198, 2010. This paper, referred to as [Liu10a] in the list of references, proposes strategies for accelerating the synchronization task in DEVS-based simulation on the Cell processor. It also analyzes the event-level parallelism available in the simulation process. These preliminary results, later refined, are included in the MADS technique that appears in Chapter 5 of this dissertation.
- Liu, Q., G. Wainer, L. Lu, and M. Perrone, "Novel Performance Optimization of Large-Scale Discrete-Event Simulation on the Cell Broadband Engine". *Proceedings of the 2010 International Conference on High Performance Computing & Simulation*, Caen, France, pp. 108-114, 2010. This paper, referred to as [Liu10b] in the list of references, proposes various optimization strategies for improving the performance of DEVS-based simulation on the Cell processor. It also presents preliminary experimental results using a wildfire propagation model as a case study. The optimization strategies and the preliminary results, later refined, are discussed in Chapter 5 and Chapter 6 of this dissertation respectively.
- Liu, Q., and G. Wainer, "Exploring Multi-Grained Parallelism in Compute-Intensive DEVS Simulations". *Proceedings of the 24th IEEE Workshop on Principles of Advanced and Distributed Simulation*, Atlanta, GA, pp. 65-72, 2010. This paper,

referred to as [Liu10c] in the list of references, proposes the core algorithms included in the MADS technique. These algorithms, later extended, are presented in Chapter 5 of this dissertation.

The following manuscripts are related to other aspects of DEVS-based M&S practice.

- Harzallah, Y., V. Michel, Q. Liu, and G. Wainer. "Distributed Simulation and Web Map Mash-Up for Forest Fire Spread". *Proceedings of the 2008 IEEE Congress on Services – Part I, Honolulu*, HI, pp. 176-183, 2008. This paper, referred to as [Har08] in the list of references, presents a Web service based mash-up application for wildfire emergency management, including a distributed CD++ simulation service, a global weather service, and the Google Maps service. In particular, the CD++ simulator is integrated with the `fireLib` library [Bev96] to compute the fire spread rates based on a set of real-time environmental parameters. This wildfire propagation model is used as a benchmark for performance evaluation in Chapter 6 of this dissertation.
- Feng, B., Q. Liu, and G. Wainer. "Parallel Simulation of DEVS and Cell-DEVS Models on Windows-based PC Cluster Systems". *Proceedings of the 2008 Spring Simulation Multiconference: High Performance Computing Symposium*, Ottawa, Canada, pp. 439-446, 2008. This paper, referred to as [Fen08] in the list of references, presents a technique for constructing ad hoc clusters from Microsoft Windows-based commodity PCs to carry out conservative parallel simulation of P-DEVS and Cell-DEVS models, allowing users to unleash the relatively untapped computing power of desktop workstations.
- Wainer, G., Q. Liu, J. Chazal, L. Quinet, and M. K. Traore, "Performance Analysis of Web-based Distributed Simulation in DCD++: A Case Study across the Atlantic Ocean". *Proceedings of the 2008 Spring Simulation Multiconference: High Performance Computing Symposium*, Ottawa, Canada, pp. 413-420, 2008. This paper, referred to as [Wai08a] in the list of references, presents a case study of Web-based distributed DEVS simulation between Canada and France using the DCD++ simulation engine [Wai08b], analyzing the performance bottlenecks found in the system and identifying the areas for further investigation.
- Wainer, G., and Q. Liu, "Tools for Graphical Specification and Visualization of DEVS Models". *SIMULATION*, 85(3), pp. 131-158, 2009. This paper, referred to as [Wai09b]

in the list of references, presents the design and implementation of various graphical modeling and visualization facilities that have been incorporated into the CD++ environment. With the MADS technique proposed in this dissertation, some of these facilities can be tightly coupled with the parallel simulation engine on multicore platforms to provide an integrated high-performance modeling, simulation, and visualization environment, a future research direction that will be discussed in Chapter 7 of this dissertation.

- Wainer, G., Q. Liu, O. Dalle, and B. P. Zeigler, "Applying DEVS and Cellular Automata Methodologies to Serious Games", *Simulation & Gaming: An Interdisciplinary Journal of Theory, Practice and Research*, 2010 (in press). This paper, referred to as [Wai10] in the list of references, describes how P-DEVS and Cell-DEVS formalisms can be used to support simulation-based serious game applications for a variety of non-entertainment purposes. As will be discussed in Chapter 7, these applications could also benefit from the parallel simulation techniques presented in this dissertation to improve performance.

1.4. Organization

The following of this dissertation is organized as follows. Chapter 2 presents the concepts of the DEVS M&S framework and the P-DEVS and Cell-DEVS formalisms, which provide the theoretical foundation for the research. It also introduces the CD++ simulation algorithms and generalizes several key intrinsic computational properties of the DEVS-based simulation process. Chapter 3 reviews the literature most relevant to the research objectives of this dissertation. Chapter 4 presents the LTW protocol, including its assumptions, algorithms, and implications. Chapter 5 proposes the MADS technique for efficient parallel DEVS simulation on the Cell processor, illustrating the design methodologies, simulation algorithms, and the underlying software architecture. The possible approaches to integrating the MADS technique with other cluster-based PDES algorithms are also discussed. Chapter 6 analyzes the performance of the LTW protocol and the MADS technique quantitatively using different benchmark models of varied characteristics. Chapter 7 concludes the dissertation by summarizing the research objectives, contributions, and main qualitative results. It also suggests a number of future research directions.

Chapter 2. DEVS Framework and Its Implementation

This chapter presents the DEVS M&S framework and its implementation in the CD++ environment. Section 2.1 introduces the basic concepts and the software architecture of the DEVS M&S framework. Section 2.2 reviews the classical DEVS formalism. Section 2.3 covers the Parallel DEVS (or P-DEVS) formalism. The Timed Cell-DEVS and Parallel Cell-DEVS formalisms are presented in Section 2.4 and 2.5 respectively. Section 2.6 describes the simulation algorithms and computational properties in the context of the CD++ environment.

2.1. Conceptual Modeling and Simulation Framework

A conceptual M&S framework defines the basic entities and their relationships that are central to the M&S practice. To allow for a strict modular separation of model and simulator concepts, Zeigler *et al.* proposed a conceptual M&S framework that includes four basic entities and two types of relationships [Zei00], as illustrated in Figure 1.

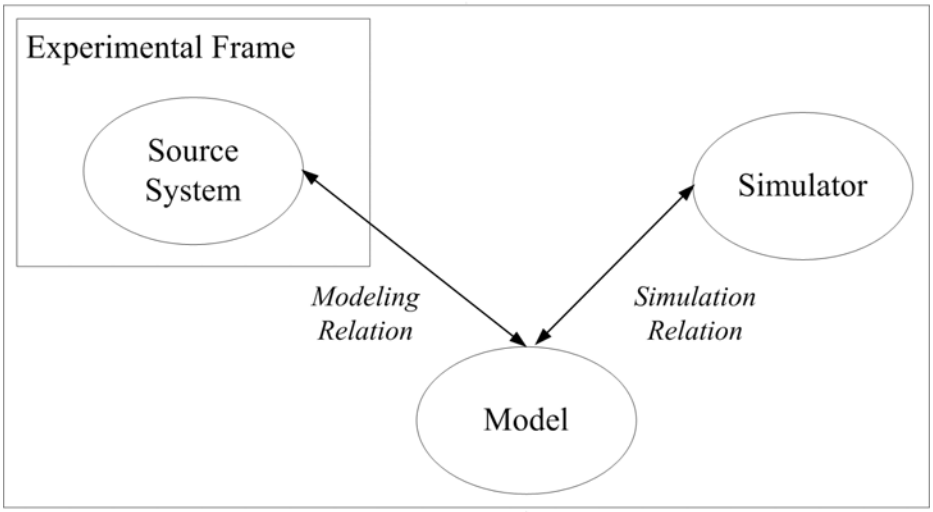


Figure 1. Entities and Relationships of a System M&S Framework [Zei00]

The entities include *source system*, *experimental frame*, *model*, and *simulator*. The source system is the real or virtual environment under study. It serves as the source of data that are collected based on an experimental frame that is of interest to the modeler. The experimental frame specifies the conditions under which the source system is observed or experimented with. A model is an abstract representation of the construction and behavior of

the source system. In general, a model is a set of instructions, rules, mathematical equations, or constraints that are used to approximate the I/O trajectories of the source system. A simulator is any computation system that defines the operational semantics required to execute a class of models in order to faithfully generate their behavior.

The two fundamental relationships among the entities are the *modeling relation* (or *validity*) and the *simulation relation* (or *simulator correctness*) [Zei00]. The modeling relation exists between a source system of interest, an experimental frame in use, and a model defined for that source system. This relation is concerned with the accuracy of the model-generated behavior when compared to the system behavior observed under the experimental frame. On the other hand, the simulation relation lies between a model and a simulator, determining whether the model is executed correctly by the simulator.

The clear separation of model and simulator concepts in the M&S framework offers a number of advantages [Zei00]. First, the same class of models expressed in a specific formalism can be executed by different simulators that are built to support such formalism, allowing for portability and interoperability at a high level of abstraction. Secondly, the models and simulators can be validated and verified independently and reused in later combinations with minimal re-verification effort. In addition, the models and simulators can evolve separately as needed without loss of compatibility as long as they adhere to the semantics of a common formalism.

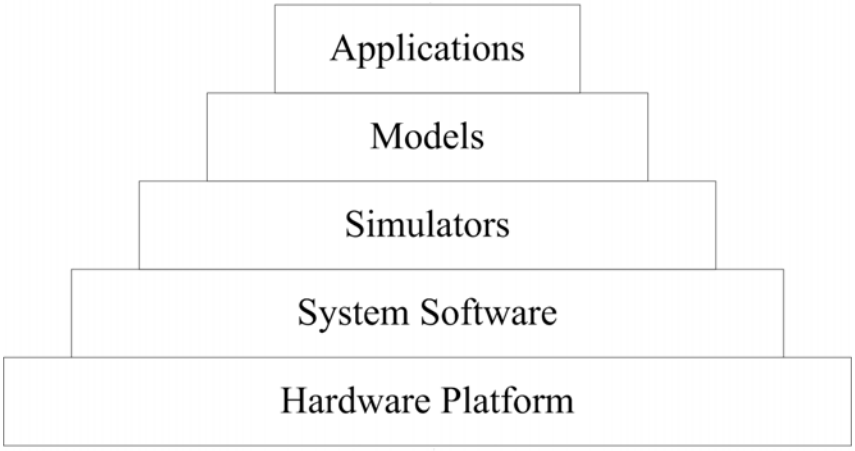


Figure 2. Layered Software Architecture of the M&S Framework

Figure 2 gives an architectural view of the M&S framework with five distinct layers. The bottom layer of the architecture includes diverse hardware platforms on which M&S is based. Examples are multiprocessor parallel computing systems, commodity PC

workstations, and single-board computers commonly used in real-time applications. The layer above the hardware platform is the system software layer, which includes various operating systems, programming languages, standard libraries, and simulation middleware solutions. The system software layer provides the necessary services for developing simulators, which implement different formalisms to support the execution of certain classes of models. The applications layer addresses issues related to standardization, reusability, and interoperability of heterogeneous models to realize transparent sharing of computing power, data, models, and experiments in confederated systems. With this layered architecture, technology changes in any layer would have a minimal impact on the other layers, allowing for the flexibility required to meet the needs of evolving challenges in M&S. This dissertation is mainly focused on the system software and the simulators layers to achieve efficient parallel simulation on different hardware platforms.

2.2. Classical DEVS Formalism

Based on the above M&S framework concepts, the **Discrete Event System Specification (DEVS)** formalism supports hierarchical construction of reusable discrete-event models in a modular way [Zei00]. A DEVS model is defined as a mathematical entity that is composed of a hierarchy of *atomic* (behavioral) and *coupled* (structural) components. An atomic model is a basic building block that possesses a set of input and output ports through which all interactions with the outside environment are mediated. There are two types of events that can occur at an atomic model, namely *internal events* and *external events*. Internal events arise from within the atomic model, manifest themselves as events to be sent to other model components through the output ports, and change the state of the model. When external events, arising in the outside environment, are received from the input ports, the atomic model determines how to respond to them. This dynamic behavior can be defined rigorously using set theory notations as follows [Zei00].

$$M = <X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta>,$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

S is the set of states;

δ_{int}: S→S is the *internal state transition function*;

δ_{ext}: Q×X→S is the *external state transition function*, where

Q = {(s,e) | s∈S, 0 ≤ e ≤ ta(s)} is the set of total states, and

e is the time elapsed since the last state transition;

λ: S→Y is the *output function*;

ta: S→ $R^{+}_{0,\infty}$ is the *time advance function*.

Figure 3 illustrates the behavior of a DEVS atomic model. At any time, an atomic model is in some state s ∈ S. Without the influence of external events, it will remain in state s for a period of ta(s), which is also called the lifetime of state s. When ta(s) expires with e = ta(s), the atomic model outputs the value given by λ(s) and changes to a new state δ_{int}(s). An atomic model that has a due internal state transition at the current simulation time is referred to as an ***imminent*** model component. Notice that output is only possible in an imminent model and occurs right before the scheduled internal state transition.

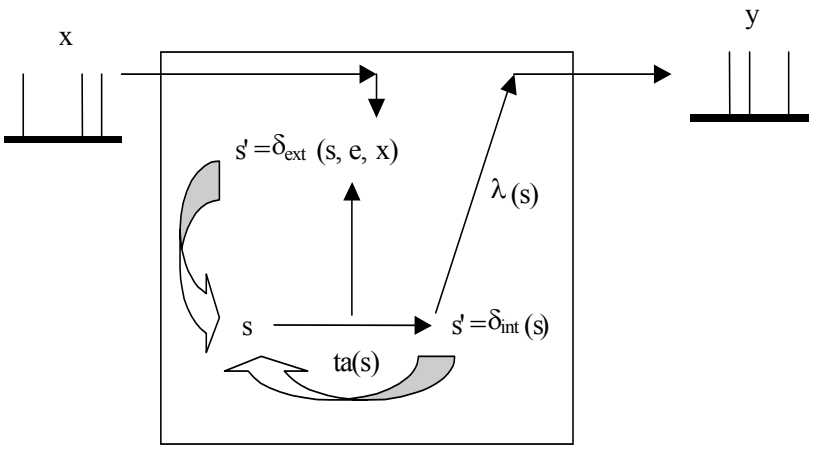


Figure 3. Informal Illustration of a DEVS Atomic Model [Zei00]

If an external event x ∈ X occurs before the expiration of the current state lifetime, i.e., when the atomic model is in total state (s,e) with e < ta(s), an external state transition is performed that changes the model state to a new one given by δ_{ext}(s,e,x). In other words, the internal state transition function dictates the model's new state when no events have occurred since the last transition, whereas the external state transition function determines the model's new state under the influence of an external event. Note that the lifetime of a state can take on any nonnegative values, including *zero* and *infinity*. An atomic model is said to be in a *transitory state* s if ta(s) has a value of 0; and in a *passive state* if ta(s) is equal to infinity.

The DEVS formalism has a well-defined concept of system modularity and component coupling that gives rise to hierarchical model construction. A DEVS coupled model specifies how its subordinate model components are connected with each other and with the external environment, as shown in the following formal definition [Zei00].

$$N = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, \textit{Select} \rangle,$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

D is the set of the component names;

The following requirements are imposed on each component d that is included in D.

$M_d = \langle X_d, Y_d, S_d, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ is a DEVS model with

$X_d = \{(p,v) \mid p \in IPorts_d, v \in X_p\}$ and $Y_d = \{(p,v) \mid p \in OPorts_d, v \in Y_p\}$.

The component coupling relationships are subject to the following requirements.

External Input Coupling (EIC) connects external inputs to component inputs,

$$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\};$$

External Output Coupling (EOC) connects component outputs to external outputs,

$$EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\};$$

Internal Coupling (IC) connects component outputs to component inputs,

$$IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\};$$

Select: $2^D - \{\} \rightarrow D$ is the *tie-breaking function*.

Direct feedback loops are not allowed in the formalism. That is, no output port of a component may be connected to an input port of the same component, which can be formally specified as $((d, op_d), (e, ip_d)) \in IC$ implies $d \neq e$. In addition, the values sent from a source port must be within the range of acceptable values of a destination port, as expressed by:

$$\forall ((N, ip_N), (d, ip_d)) \in EIC : X_{ipN} \subseteq X_{ipd};$$

$$\forall ((a, op_a), (N, op_N)) \in EOC : Y_{opa} \subseteq Y_{opN};$$

$$\forall ((a, op_a), (b, ip_b)) \in IC : Y_{opa} \subseteq X_{ipb}.$$

Thanks to the *closure under coupling* property, which states that a valid coupling of DEVS-representable model components yields a DEVS-representable model component [Pra94], a coupled DEVS model can be treated as an equivalent basic model in the DEVS

formalism. The resulting basic model can then be used in a larger coupled model as required for hierarchical model construction.

As multiple imminent components can coexist in a coupled model at the same simulation time, ambiguity may arise. If an imminent component executes its internal state transition and produces an output that is received by another imminent component as an external event, it is not clear which state transition should happen at the receiving component. To solve such potential ambiguities, the DEVS formalism introduces a *Select* function that provides a tie-breaking mechanism at the coupled level. A modeler must use the *Select* function to specify an order over all the subordinate components of a coupled model so that only one component is chosen from the imminent set to execute internal transition with e = 0. The other imminent components are divided into two groups: receivers of the external events generated from the chosen imminent component, and the rest. Components in the former group will execute their external transitions with e = ta(s), and those in the latter group will remain imminent in the next simulation cycle and may need to use the *Select* function again to decide the execution sequence.

This tie-breaking mechanism, however, suffers from two drawbacks. First, an artificial ordering of imminent components might not properly reflect the consequence of the occurrence of multiple concurrent events in the real system. Secondly, the serialized execution could prohibit the exploitation of potential parallelism in the simulation system, especially in parallel and distributed simulation environment. These problems have been addressed by Chow and Zeigler in a DEVS extension, known as the Parallel DEVS formalism [Cho94], which will be presented in the next section.

In addition to the specification for hierarchical model composition, the DEVS framework also includes an *abstract simulation engine architecture* that consists of different simulation entities organized in a hierarchy that mimics the hierarchical structure of a model. These simulation entities (also known as ***abstract DEVS processors*** [Zei00]) are specialized into *Simulators* and *Coordinators*, which are used to control the execution of the corresponding *atomic* and *coupled* model components respectively. During a simulation, the interaction between different model components is achieved through event messages exchanged between the Simulators and Coordinators. In order words, the simulation process is managed by the abstract DEVS processors in an event-driven fashion. Note that, although

it appears intuitive to simply consider an abstract DEVS processor as a LP, this one-to-one correspondence may not always hold. Specifically, a single LP can be used to implement *one or more* abstract DEVS processors in a simulation, though it is uncommon to implement an abstract DEVS processor using multiple LPs.

2.3. Parallel DEVS Formalism

The **Parallel DEVS** (or **P-DEVS)** formalism is intended to eliminate the serialization constraint imposed by the tie-breaking *Select* function as required in the classical DEVS definition [Cho94]. Instead of resolving transition collisions at the coupled level, P-DEVS addresses the tie-breaking problem within the specification of each atomic model. This is achieved by introducing an additional transition function, referred to as *confluent transition function*, which allows a modeler to explicitly define the collision behavior for individual atomic models. In addition, each atomic model uses a *bag* structure to collect all external events received from other model components at a given simulation time so that these events can be processed as a group in the state transition, combining the execution of multiple external transitions into a single one. As a result, many imminent components can be activated *simultaneously* to send output to other components all at the same simulation time [Zei03]. The receiver is responsible for examining the input external events and interpreting them properly. It has been suggested that the P-DEVS formalism allows for increased parallelism to be exploited in a simulation [Cho94].

The formal specification of a P-DEVS atomic model is given as follows [Cho94].

$$M = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >,$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

S is the set of sequential states;

δ_{int}: $S \rightarrow S$ is the *internal state transition function*;

δ_{ext}: $Q \times X^b \rightarrow S$ is the *external state transition function*, where

X^b is a set of bags over elements in X,

$Q = \{(s,e) \mid s \in S, \ 0 \leq e \leq ta(s)\}$ is the set of total states,

e is the time elapsed since the last state transition;

δ_{con}: $Q \times X^b \rightarrow S$ is the *confluent transition function*;

λ: $S \rightarrow Y^b$ is the *output function*;

ta: $S \rightarrow R^+_{0,\infty}$ is the *time advance function*.

Consequently, the *Select* function is removed in the P-DEVS formalism, resulting in the following revised definition for P-DEVS coupled models [Cho94].

$$DN = <X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC>.$$

The sets of input/output ports and values (X and Y) as well as the couplings (EIC, EOC, and IC) are defined similarly as in the classical DEVS coupled model specification. The basic components (D and M_d) are required to be P-DEVS structures.

2.4. Timed Cell-DEVS Formalism

The **Cellular automata (CA)** theory was pioneered by John von Neumann in his study of self-replicating systems [Neu66]. The goal was to design an artificial system that consists of a two-dimensional mesh of finite state machines interconnected locally with each other. The state machines, referred to as *cells*, change their states *synchronously* and *in parallel* at discrete time steps based on the states of a finite set of neighboring cells, referred to as the *neighborhood*, by evaluating a common local update rule. Such CA systems bear several intrinsic properties commonly found in the physical world, including homogeneous, massively parallel, and local interconnectivity [Kar05]. As a result, CA models have been widely used to simulate complex physical, biological, and social systems (see, e.g., [Wol02, Yan03, Rot04, Cor06, and Sli09]). Despite its widespread application, the CA approach is regarded as computationally inefficient for two main reasons [Wai01]. First, it uses a discrete time base that constrains simulation precision and execution efficiency. Secondly, all the cells are evaluated synchronously, incurring an unnecessarily high computational cost when only a small fraction of the cells needs to be updated at each time step.

The **Timed Cell-DEVS** formalism [Wai02a] attempts to solve these problems by defining n-dimensional cell spaces as discrete-event models, allowing for more efficient asynchronous execution using a continuous time base without losing simulation accuracy. Each cell is represented as a DEVS atomic model that changes state in response to the occurrence of events in an event-driven fashion. The temporal behavior of a cell is specified with explicit timing delay constructions, which find their origins in logic circuit simulations

[Chi76]. Two types of delay constructions that are most commonly used in hardware design applications are known as *transport delay* and *inertial delay*. Transport delay models a straightforward propagation of signals over links of infinite bandwidth with anticipatory semantics, whereas inertial delay uses preemptive semantics to model the timing behavior of a retarded system that changes state only when a certain level of energy is reached. These delay constructions can be used to model the behavior of other physical phenomena [Wai01].

A Timed Cell-DEVS atomic model is defined formally as follows [Wai02a].

$$CM = <X, Y, I, S, \theta, N, delay, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D>,$$

where

X is the set of external input events;

Y is the set of external output events;

$I = <\eta, \mu, P^x, P^y>$ represents the cell's modular interface, where

η is the cell's neighborhood size, μ is the number of input/output ports,

and P^x and P^y are the definition of the cell's input and output ports;

S is the set of sequential states for the cell;

θ is the definition of the cell's state;

N is the set of values for input events;

delay $\in$ {transport, inertial} is the type of the delay construction;

d is the delay period;

δ_{int}: $\theta \rightarrow \theta$ is the *internal state transition function*;

δ_{ext}: $Q \times X \rightarrow \theta$ is the *external state transition function*, where

$Q = \{(s,e) \mid s \in \theta \times N \times d;\ e \in [0, D(s)]\}$ is the cell's state values;

τ: $N \rightarrow S$ is the *local transition function*;

λ: $S \rightarrow Y$ is the *output function*;

D: $\theta \times N \times d \rightarrow R^+_{0,\infty}$ is the *state's duration function*.

Figure 4 illustrates a Timed Cell-DEVS atomic model. The modular interface (I) of a cell consists of a fixed number of ports that are connected to other cells in the neighborhood. Through the input and output ports, a cell can exchange data with other neighboring cells as well as those model components outside of the cell space. A cell's future state is determined by the *local transition function* (τ) based on the cell's current state and the values arrived at

the input ports. If the calculated future state is different from the current one, a state change is scheduled and the value of the future state will be revealed to all the neighboring cells after a delay period (d). Otherwise, the cell remains in its current state and therefore no output will be sent. As in a DEVS atomic model, the output and state transition behavior of each cell is defined by the output function (λ), internal state transition function (δ_{int}) and external state transition function (δ_{ext}).

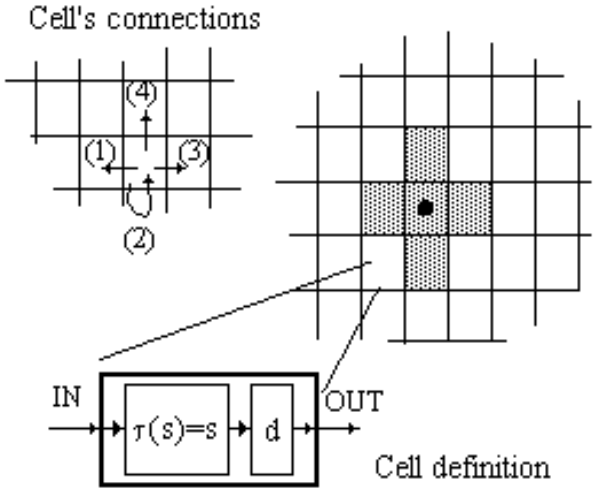


Figure 4. Informal Illustration of a Timed Cell-DEVS Atomic Model [Wai02a]

The cells are coupled with each other, through the neighborhood relation, to form a cell space, which can then be integrated with other DEVS or Cell-DEVS models. A cell space is defined as a Timed Cell-DEVS coupled model, as follows [Wai02a].

$$GCC = <X_{list}, Y_{list}, I, X, Y, \eta, \{t_1, \ldots, t_n\}, N, C, B, Z, \textit{Select}>,$$

where

X_{list} is the list of input coupling;

Y_{list} is the list of output coupling;

I represents the modular model interface;

X is the set of external input events;

Y is the set of external output events;

η is the dimension of the cell space;

$\{t_1, \ldots, t_n\}$ is the number of cells in each of the dimensions;

N is the neighborhood set;

C defines the cell space;

B is the set of border cells;

Z is the translation function;

Select is the tie-breaking function for simultaneous events.

The cell space (C) is a coupled model composed of an array of Timed Cell-DEVS atomic models with a fixed size ($t_1 \times \ldots \times t_n$). The neighborhood set (N) is a set of n-tuples that specifies the positions of the neighboring cells relative to a given central cell. The cells on the border of the cell space are included in set B. If B is empty, every cell in the cell space has the same behavior (i.e., homogeneous cell space) and the border is *wrapped* (i.e., the cells on one side of the border are viewed as being connected to those on the opposite side). Otherwise, the border cells will have different behavior from those inside the cell space (i.e. heterogeneous cell space). The interface (I) of the cell space itself includes the lists of input and output coupling (X_{list} and Y_{list}). The Z function is used to specify the coupling between cells inside the cell space. It translates the values on the i^{th} output port of a cell C_a into values for the i^{th} input port of a neighboring cell C_b. Finally, as in the classical DEVS formalism, potential transition collisions are handled at the cell space level using the *Select* function, with the same drawbacks as discussed earlier in Section 2.2.

2.5. Parallel Cell-DEVS Formalism

In order to resolve transition collisions without using the *Select* function, a new version of the Timed Cell-DEVS formalism, referred to as **Parallel Cell-DEVS** [Wai00], has been proposed based on the P-DEVS concepts. The formal definition of a Parallel Cell-DEVS atomic model is given as follows [Wai00].

$$PCM = < X^b, Y^b, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \delta_{con}, \tau, \tau_{con}, \lambda, D >.$$

Most of the components in the definition remain unchanged as in the Timed Cell-DEVS specification, with two exceptions. First, the external state transition function and the output function are improved to handle *bags* of inputs and outputs (X^b and Y^b) for each cell. Secondly, two additional confluent state transition functions (δ_{con} and τ_{con}) are introduced in the definition. When collisions between internal and external events happen at a cell, the confluent function δ_{con} is triggered as in the P-DEVS formalism. Instead of handling the collisions by itself, function δ_{con} activates the *confluent local transition function* τ_{con}, which in turn analyzes the current values in the input bags and presents a unique set of inputs for the cell to compute the next state. Hence, a modeler can precisely control the behavior of

each cell under collision situations by implementing the confluent local transition function.

As a result, the *Select* function is eliminated in the Parallel Cell-DEVS coupled model definition, which is given as follows [Wai00].

$$GCC = <X_{list}, Y_{list}, I, X, Y, \eta, \{t_1, \ldots, t_n\}, N, C, B, Z >.$$

Each cell in the cell space (C) is a Parallel Cell-DEVS atomic model, while all the other components are defined in the same way as presented in the previous section.

In [Wai00], Wainer proved that a Parallel Cell-DEVS model can be used as an equivalent P-DEVS model and the property of closure under coupling still holds, allowing for seamless integration of Parallel Cell-DEVS models with other P-DEVS models to construct hierarchical systems.

The Parallel DEVS and Cell-DEVS [2] formalisms not only afford a unified M&S framework under the DEVS paradigm, but also facilitate the exploitation of increased parallelism in parallel and distributed simulations. Together, they serve as the theoretical foundation for the research described in the following chapters.

2.6. Parallel DEVS and Cell-DEVS Simulation in CD++

The **CD++** toolkit, originally developed by Wainer [Wai02b], is an open-source, object-oriented M&S environment that implements both P-DEVS and Cell-DEVS formalisms in C++. Over years, CD++ has been augmented with a family of DEVS-based simulation engines using different middleware technologies for simulation on varied platforms [Tro03, Gli06, Chi07, Liu07, Fen08, Har08, Wai08a, Wai08b, and Wai09a].

Among these simulation engines, the *optimistic parallel CD++ simulator* (or **PCD++** for short) allows for TW simulation of P-DEVS and Cell-DEVS models on Linux-based distributed-memory multiprocessor cluster systems [Liu06 and Liu07]. It is built on top of the WARPED simulation kernel [Rad98 and Mar03], which in turn relies on the **Message Passing Interface (MPI)** libraries [Gro99] for inter-node communication. In PCD++, a model is partitioned at the *atomic* level, and each abstract DEVS processor is implemented as a LP[3]. The resulting LPs are then mapped to a set of physical processors (or nodes) for

[2] For the sake of simplicity, the term *Cell-DEVS* is used hereafter to stand for *Parallel Cell-DEVS*.
[3] Since a LP is simply an implementation of a specific abstract DEVS processor in PCD++, *LPs* and *abstract DEVS processors* are used interchangeably hereafter in this dissertation.

parallel execution. The PCD++ simulator incorporates various optimization techniques and addresses several important issues arising in DEVS-based TW simulations, providing a testbed for the research presented in this dissertation.

Following previous studies [Kim04 and Gli06], which suggest that flattening the abstract DEVS processor structure can reduce communication overhead significantly due to the elimination of intermediate coordinators in the hierarchy, PCD++ adopts a *flat LP structure* in which the sequential simulation on each node consists of three types of LPs, including one ***Node Coordinator* (NC)**, one ***Flat Coordinator* (FC)**, and a group of ***Simulators***. One can consider that all of the intermediate coordinators previously existed in the abstract DEVS processor hierarchy are collapsed into a single FC, which aggregates the functionality of these intermediate coordinators in a more efficient way.

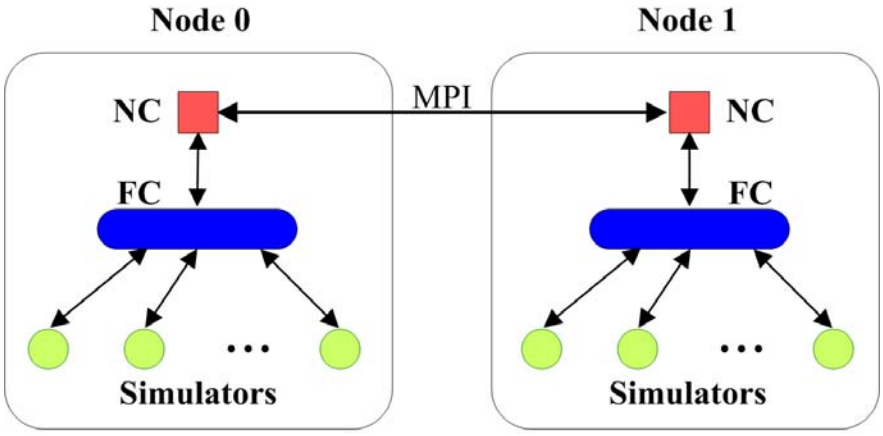


Figure 5. Flat LP Structure in PCD++

Figure 5 shows the flat LP structure established across two physical processors. A parallel simulation is managed by a set of NCs running asynchronously on different nodes in a decentralized manner. On each node, the NC is responsible for sending and receiving inter-node MPI messages as well as advancing the local simulation time, while the FC is in charge of synchronizing the child Simulators underneath and routing messages between the local LPs using user-defined model coupling information. Messages exchanged between Simulators residing on the same node are directly routed to their destinations by the local FC. In the case when a Simulator sends a message to a remote Simulator located on another node, the message is forwarded by the sender's FC to the local NC, which then relays the message to the corresponding remote NC. On the receiver side, the NC will route the message to the destination Simulator through its child FC.

It is worthwhile to point out that, thanks to the clear separation of model and simulator concepts in the DEVS framework, the flat LP structure is implemented *transparently* to modelers, who still define modular P-DEVS model components and construct hierarchical systems as usual [Gli06]. Moreover, as will be discussed in Chapter 5, this flat LP structure also facilitates the exploitation of fine-grained event-level and data-level parallelism in parallel DEVS simulations on the Cell processor.

The LPs exchange messages that fall into two categories: ***content messages*** include the *external* (X) and *output* (Y) events that represent the actual model input and output data, while ***control messages*** include the *initialization* (I), *collect* (@), *internal* (*), and *done* (D) events that control the execution of *simultaneous* events scheduled at each virtual time in line with the P-DEVS formalism.

2.6.1. Event-Processing Algorithms

Based on the flat LP structure, this section briefly describes the PCD++ event-processing algorithms defined for the Simulators, the FC, and the NC respectively, while more details can be found in [Liu06 and Liu07]. In the following, an event is denoted as (*event_type*, *receive_time_stamp*). The send time stamp of an event is by default the current virtual time t.

- **Simulator event-processing algorithms**

As shown in Figure 6, a Simulator triggers the P-DEVS functions defined in its associated atomic model by processing the (I), (@), (X), and (*) events received from the parent FC.

When an (I, 0) is received at the beginning of a simulation, a Simulator initializes the state variables and returns a (D, t) to the FC (line 1.4). The Simulator then waits for another event from the FC. The output function (λ) of an *imminent* atomic model is called during the processing of a (@, t) (line 2.4). As a result, a (Y, t) is sent to the FC, followed by a (D, t). The imminent atomic model enters into a *transitory* state since the internal state transition (δ_{int}) needs to be performed immediately after the output. A received (X, t) is simply cached in the message bag of the associated atomic model (line 3.2). All these cached (X, t) events will be consumed by the atomic model as a whole when its state transition function (δ_{ext} or δ_{con}) is triggered by an ensuing (*, t) received from the FC (line 4.4 and 4.9). Upon the arrival of a (*, t), the atomic model's state transition functions are invoked based on the

current state of the model and the status of the message bag (line 4.2 to 4.11). The Simulator then returns a (D, t) event to the FC (line 4.13), carrying the timing information (ta) of the next state change scheduled in the atomic model.

```
tL : current virtual time;                ta : remaining time to the next state change;
tN : absolute next state change time;     e  : time elapsed since last state change;

1.1 when an (I, 0) is received from the parent FC
1.2         tL = 0; ta = infinity                       //passive state
1.3         initialize state variables in the Simulator and the associated atomic model
1.4         send (D, 0) with ta to the parent FC
1.5 end when

2.1 when a (@, t) is received from the parent FC
2.2         if t = tN then
2.3                     tL = t; ta = 0                  //transitory state
2.4                     Y = λ(s)                        //model output
2.5                     send (Y, t) to the parent FC
2.6                     send (D, t) with ta to the parent FC
2.7         end if
2.8 end when

3.1 when a (X, t) is received from the parent FC
3.2         insert (X, t) into the bag associated with the atomic model
3.3 end when

4.1  when a (*, t) is received from the parent FC
4.2         if tL ≤ t < tN then
4.3                     e = t – tL; ta = tN - t
4.4                     s = δext(s, e, bag)             //external state transition
4.5                     empty bag
4.6         else if t = tN and bag is empty then
4.7                     s = δint(s)                     //internal state transition
4.8         else if t = tN and bag is not empty then
4.9                     s = δcon(s, bag)                //confluent state transition
4.10                    empty bag
4.11        end if
4.12        tL = t
4.13        send (D, t) with ta to the parent FC
4.14 end when
```

Figure 6. Simulator Event-Processing Algorithms

- **FC event-processing algorithms**

The FC synchronizes its child Simulators, routes input/output messages between them, and forwards output messages to the NC when necessary. The FC event-processing algorithms are given in Figure 7.

```
1.1 when an (I, 0) is received from the parent NC
1.2         t_L = 0;  doneCount = number of child Simulators
1.3         send an (I, 0) to each child Simulator
1.4 end when

2.1 when a (@, t) is received from the parent NC
2.2         t_L = t; ta = 0
2.3         call findImminents() to find the IDs of all imminent child Simulators with t_N = t
2.4         for each imminent child C_i with ID = i
2.5                 cache i in the synchronizeSet;  doneCount++
2.6                 send (@, t) to C_i
2.7         end for each
2.8 end when

3.1 when a (X, t) is received from the parent NC
3.2         insert (X, t) into the bag
3.3 end when

4.1   when a (Y, t) is received from a child Simulator C_i
4.2         if (Y, t) eventually influences a remote Simulator or the environment then
4.3                 forward the (Y, t) to the parent NC
4.4         end if
4.5         for each child C_j that is influenced by (Y, t)
4.6                 (X, t) = Z_i,j (Y, t)              //translate the output into an input
4.7                 cache j in the synchronizeSet
4.8                 send (X, t) to C_j
4.9         end for each
4.10 end when

5.1   when a (*, t) is received from the parent NC
5.2         t_L = t
5.3         for each (X, t) in bag
5.4                 for each local receiver C_i of (X, t)
5.5                         send (X, t) to C_i
5.6                         cache i in the synchronizeSet
5.7                 end for each
5.8         end for each
5.9         empty bag
5.10        for each i in the synchronizeSet
5.11                send (*, t) to i
5.12                doneCount++
5.13        end for each
5.14        clear the synchronizeSet
5.15 end when

6.1 when a (D, t) is received from a child Simulator C_i
6.2         doneCount--
6.3         update t_N = t + (D, t).ta in entry i of the timeOfNextStateChange MAP
6.4         if doneCount = 0 then
6.5                 t_L = t
6.6                 min_ta = findMinTime(timeOfNextStateChange) - t
6.7                 send (D, t) with min_ta to the parent NC
6.8         end if
6.9 end when
```

Figure 7. FC Event-Processing Algorithms

The FC uses a *synchronizeSet* to record the IDs of those child Simulators with a state transition (δ_{int}, δ_{ext}, or δ_{con}) scheduled at the current virtual time. A variable called *doneCount* is used to keep track of the expected number of (D, t) events that will be returned from the child Simulators. The FC also uses a map structure called *timesOfNextStateChange* (<*SimulatorID*, *absoluteNextStateChangeTime*>) to store the timing information encoded in the (D, t) events received from the child Simulators (line 6.3).

The synchronization task is performed by the FC in two functions, referred to as `findImminents` and `findMinTime` respectively, as follows.

(1) Function `findImminents` is called when a (@, t) is received from the NC (line 2.3). This function retrieves the IDs of all *imminent* child Simulators from the *timesOfNextStateChange* map by comparing the stored Simulator timing information with the current simulation time t, which is determined by the NC;

(2) Function `findMinTime` is invoked when all of the expected (D, t) events have already been received from the child Simulators (line 6.6). This function searches the *timesOfNextStateChange* map to find the *minimum* absolute time of the next state change scheduled by the child Simulators. The resulting minimum remaining time to the next state change (*min_ta*) is then wrapped in a (D, t) to be sent to the NC (line 6.7), which will use this timing information to determine the next simulation time on the host node.

The FC routes a (Y, t) received from a child Simulator to its local destination Simulators as one or more (X, t) events based on the model coupling relation (line 4.6 to 4.8). If the (Y, t) also needs to be sent to other remote Simulators or to the external environment, the FC just forwards the (Y, t) to the NC (line 4.3). Moreover, the FC caches the (X, t) events received from the NC in its *bag* so that these events can be flushed to the destination child Simulators during the processing of (*, t) events (line 5.5).

- **NC event-processing algorithms**

As shown in Figure 8, the NC uses an *inter-node message buffer* to hold incoming (X, t) events from the other remote NCs (line 1.2). If a (Y, t) needs to be sent to remote NCs, it is converted to an equivalent (X, t) before transfer (line 2.6). That is, only (X, t) events are sent as inter-node MPI messages.

```
1.1 when a (X, t) is received from a remote NC
1.2         insert (X, t) to this NC's inter-node message buffer
1.3 end when

2.1 when a (Y, t) is received from the child FC
2.2         if (Y, t) needs to be sent to the environment then
2.3                 perform I/O on the output ports of the TOP model
2.4         end if
2.5         for each remote node i that hosts at least one destination Simulator
2.6                 (X, t) = Z_{i,j} (Y, t)          //translate the output into an external event
2.7                 send (X, t) to NC_i
2.8         end for each
2.9 end when

3.1   when a (D, t) is received from the child FC
3.2         t_L = t; t_N = t_L + (D, t).ta            //t_N is the min next state change time at the Simulators
3.3         if next-message-type = * then
3.4                 send (*, t) to the child FC
3.5                 next-message-type = @
3.6         else
3.7                 min-time = MIN( min time stamp among the events from the environment,
3.8                                 min time stamp among the events in the inter-node message buffer,
3.9                                 t_N )
3.10                if min-time > stop-time then
3.11                        suspend the simulation on this node (waiting for potential remote messages)
3.12                else
3.13                        if min-time = time stamp of the events from the environment then
3.14                                for each environmental event with min-time do
3.15                                        send (X, t) to the child FC
3.16                                end for each
3.17                        end if
3.18                        if min-time = time stamp of the events in inter-node message buffer then
3.19                                for each inter-node (X, t) event with min-time do
3.20                                        forward (X, t) to the child FC
3.21                                end for each
3.22                        end if
3.23                        remove all (X, t) events with min-time from the inter-node message buffer
3.24                        if t_N = min-time then
3.25                                send (@, t) to the child FC
3.26                                next-message-type = *
3.27                        else
3.28                                send (*, t) to the child FC
3.29                                next-message-type = @
3.30                        end if
3.31                end if
3.32        end if
3.33 end when
```

Figure 8. NC Event-Processing Algorithms

The NC determines the next simulation time (*min-time*) on the host node by calculating the minimum among three factors: (i) the minimum time stamp of the events received from the external environment (line 3.7); (ii) the minimum time stamp of the events currently available in the inter-node message buffer (line 3.8); and (iii) the minimum absolute Simulator state change time derived from the (D, t) event received from the FC (line 3.9).

The NC controls the message flow by virtue of a flag called *next-message-type* (either @ or *), which is initialized to @ at the beginning of a simulation. The value of this flag determines the type of the control message to be sent to the FC in the next simulation round. The NC starts the simulation on the host node by sending an (I, 0) to the FC.

2.6.2. Structural Representation

Like many other discrete-event simulators, PCD++ defines a set of event-processing algorithms for each type of LPs, specifying how to update state data and generate output in response to a given type of input events. Although this *event-oriented view* is adequate to describe the simulation execution in great detail, an *abstract structural representation* of the simulation process may be able to reveal certain high-level patterns or computational properties that are not immediately obvious in the complex message flow.

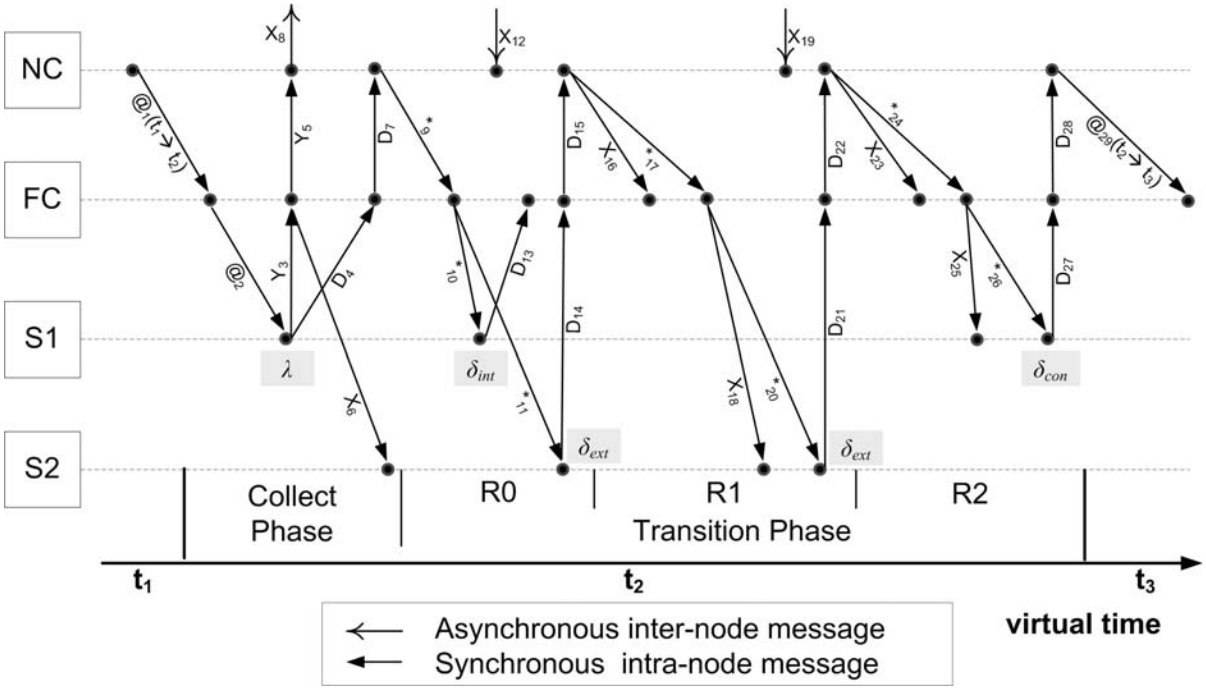


Figure 9. PCD++ Message-Passing Scenario

Based on the event-processing algorithms presented in the previous section, Figure 9 gives an example PCD++ message-passing scenario, demonstrating that the simulation process can be represented concisely as a ***multi-phased structure***. In this scenario, four LPs are created on the node, namely a NC, a FC, and two Simulators (S1 and S2). The order of event execution is marked by the sequence numbers beside the event types. An event whose send time stamp is different from receive time stamp, referred to as a ***time-changing event***, is

shown so explicitly (e.g., $@_1(t_1 \rightarrow t_2)$ denotes a collect message with send time stamp t_1 and receive time stamp t_2), whereas all the other events have the same send and receive time stamps that are equal to the current virtual time. The locations where the P-DEVS functions are invoked in an atomic model are also illustrated in the figure.

Note that, due to the asynchronous nature of inter-node messaging in a TW simulation, a Simulator may need to perform *multiple rounds of state transitions* at a given virtual time (shown as $[R_0 \ldots R_n]$) to incorporate additional (X, t) events received from remote senders. This is exemplified by S1, which first executes the atomic model's internal state transition (δ_{int}) assuming the absence of (X, t) events, but later finds that the assumption is wrong upon the arrival of X_{25} (which is derived from X_{19} and X_{23}). To correct the false δ_{int}, S1 triggers an additional confluent state transition (δ_{con}) at the atomic model to combine the scheduled internal state transition with the simultaneous external state transition at virtual time t_2. More details on the asynchronous state transition algorithms can be found in [Liu06 and Liu07].

At any virtual time *t*, the *simultaneous events* exchanged between the LPs can be organized into an optional ***collect phase*** (which only occurs if some Simulators are imminent at *t*) and a mandatory ***transition phase*** (which consists of at least one round of state transition at *t*). The simulation process begins with an ***initialization phase*** at virtual time 0. A *simulation phase* (or *round*) is initiated by a control message, either (I, t) or (@, t) or (*, t), sent from the NC to the FC (e.g., $@_1$, $*_9$, $*_{17}$, $*_{24}$, and $@_{29}$ in Figure 9), and ended by a (D, t) message returned back to the NC (e.g., D_7, D_{15}, D_{22}, and D_{28} in Figure 9). Therefore, the control messages exchanged between the NC and the FC are also called ***phase-changing events***, which include the time-changing events mentioned earlier. At the end of a transition phase, the NC determines the next virtual time and advances the simulation on the node.

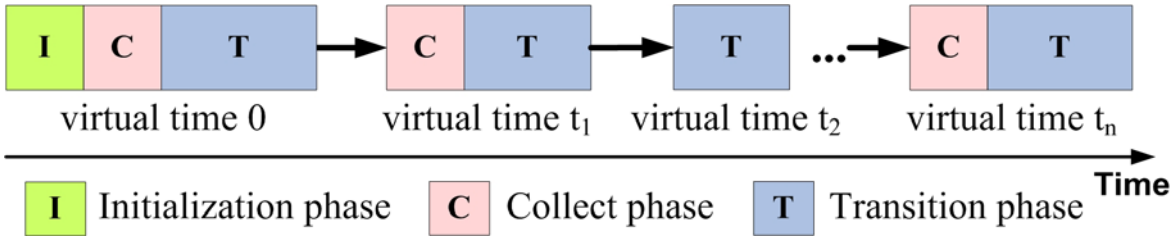


Figure 10. Multi-Phased Simulation Process on a Node

Following this *phase-oriented view*, Figure 10 shows a high-level structured abstraction of the simulation process on a node. In the diagram, the complex message flow between the LPs disappears. Instead, the simulation is depicted as a sequence of computation units, each

of which stands for the execution of simultaneous events occurred at the LPs at a distinct virtual time. These computation units are linked together by time-changing events, which transform the simulation from one virtual time to the next. In addition, each computation unit is composed of one or more ***simulation phases*** (separated by phase-changing events) that execute the simulation according to the P-DEVS formalism. Although not explicitly defined in the PCD++ event-processing algorithms, this *multi-phased structure* of simulation process emerges spontaneously from the underlying event execution and can be exploited directly in the development of novel phase-based optimization and scheduling algorithms, as will be discussed in Chapter 4 and 5.

2.6.3. Computational Properties

The above-described P-DEVS simulation process has several key intrinsic computational properties that are particularly relevant to the research of this dissertation. These computational properties are generalized as follows.

Property 1. Time advance property.

On each node, the virtual simulation time is advanced only by the NC. This property is due to the fact that all time-changing events are always sent from the NC to the other local LPs, as guaranteed by the PCD++ event-processing algorithms. On the other hand, all the events sent by the FC and the Simulators have the same send and receive time stamps that are both equal to the current virtual time. In other words, the FC and the Simulators change their LVTs passively, in response to the requests from the NC.

Property 2. Communication property.

There is no *direct* communication (i.e., exchange of event messages) between the Simulators, even though the atomic models can be coupled with each other in the model definition. All events exchanged between the Simulators go through the FC, which translates and routes the messages to their (local and remote) destinations based on user-defined model-coupling information. In addition, inter-node messaging is handled by the NCs only.

Property 3. Simultaneous event property.

At any virtual time, the simulation process involves a relatively large number of simultaneous events, which include not only the content messages that carry model input and

output data, but also the control messages that enforce a partially-ordered event execution following the P-DEVS formalism. This property is particularly pronounced in *large-scale*, *densely-interconnected*, and *highly-active* P-DEVS models where many imminent atomic models need to send output events (each of which may have multiple destinations) simultaneously at each virtual time. Besides, Jha and Bagrodia also pointed out several other general causes that can lead to the occurrence of simultaneous events in a discrete-event simulation [Jha00], as will be discussed further in Section 3.2.3.

Property 4. Pending event property.

During a simulation, the number of pending (input) events fluctuates over time. At the *end* of each simulation phase (or round), all of the events scheduled for the FC and the Simulators have already been processed, as guaranteed by the event-processing algorithms. Hence, only a relatively small number of pending events remain to be executed by the NC at these points in the simulation, including the (D, t) event returned from the FC as well as the (X, t) events received from remote NCs. On the contrary, the number of pending events tends to increase when the simulation enters into a new simulation phase (or round) as output events are generated continuously by the FC and the Simulators. Furthermore, the pending events are distributed disproportionately among the LPs during a simulation. Only a relatively small fraction of events are sent between the NC and the FC, while most of the events are exchanged between the FC and the Simulators, especially in *large-scale* simulations with many Simulators.

These computational properties rely in no way on the actual P-DEVS or Cell-DEVS models to be simulated, although some of them (i.e., Property 3 and Property 4) are most prominent in models with certain characteristics. Rather, they are the natural generalization of the event-processing algorithms and the flat LP structure employed in the PCD++ simulation engine that operates below the model layer (refer to Figure 2). As will be discussed later in Chapter 4 and Chapter 5, these intrinsic computational properties can be exploited for efficient parallel simulation of P-DEVS or Cell-DEVS models.

Chapter 3. Literature Review

This chapter presents a non-comprehensive review of previous contributions in the literature that are most relevant to the research presented in this dissertation. Section 3.1 introduces parallel discrete-event simulation and the two main synchronization approaches. Section 3.2 surveys different techniques for improving the performance of TW simulations. Section 3.3 summarizes the techniques for high-performance computing on the Cell processor.

3.1. Parallel Discrete-Event Simulation

A **Parallel Discrete-Event Simulation (PDES)** is usually composed of a set of simulation entities, referred to as **Logical Processes (LPs)**, which do *not* share any state variables, but instead interact with each other exclusively through the exchange of time-stamped event messages [Fuj00]. These LPs are mapped to a collection of physical processors of a parallel computing system. When the number of LPs exceeds the number of available processors, multiple LPs may coexist on a physical processor. The LPs allocated on the same processor typically employ a single **Future Event List (FEL)** to schedule event execution in a *sequential* manner. As event messages are also exchanged between LPs residing on different processors, the major challenge in PDES is thus to synchronize all the LPs in the system so that the parallel simulation produces exactly the same results as a sequential execution of the simulation program. This synchronization requirement is expressed as a necessary and sufficient condition, termed as the *Local Causality Constraint*, by Fujimoto [Fuj00].

> ***Local Causality Constraint**: A discrete-event simulation, consisting of LPs that interact exclusively by exchanging time stamped messages obeys the local causality constraint if and only if each LP processes events in nondecreasing time stamp order.*

To satisfy the local causality constraint, two major schools of thought have emerged that are commonly known as *conservative approaches* and *optimistic approaches*. Conservative approaches strictly avoid any possibility of *causality errors* (i.e., events are processed out of time stamp order) in a simulation. On the other hand, optimistic approaches

assume that the events are processed in time stamp order, but rely on certain detection and recovery mechanisms to dynamically correct potential causality errors when they actually occur. Conservative and optimistic synchronization techniques form the core of a large body of research in the field of PDES (see, e.g., [Fuj00, Tro02, and Per06] for surveys on both conservative and optimistic techniques). The following subsections give a brief introduction to the basic concepts behind these two approaches.

3.1.1. Conservative Synchronization Algorithms

In order to avoid causality errors, conservative synchronization algorithms need to determine when it is safe to process an event for all the LPs in the system. A LP is not allowed to process an event until it is certain that no other events with a smaller time stamp can ever occur. To this end, the LPs are explicitly synchronized using some blocking mechanisms. In general, conservative synchronization algorithms can be broadly classified into two categories, namely *synchronous* and *asynchronous*.

Synchronous conservative algorithms use global barrier synchronization and reduction at specific points in the simulation process to iteratively determine which events are safe to process (see, e.g., [Lub89, Nic93, Nic95, and Leg96]), making them best suited for shared-memory computers where the overhead of global synchronization can be minimized.

On the contrary, asynchronous conservative algorithms obviate the need for global barrier computation. Instead, a LP is blocked only when it does not have enough information to find an event to process safely. Nevertheless, deadlocks can occur if the blocked LPs form a cycle [Cha83], requiring the use of either *deadlock-avoidance* or *deadlock-recovery* techniques to ensure the progress of the simulation.

The null message algorithm (also known as the Chandy-Misra-Bryant or CMB algorithm), developed independently by Chandy and Misra [Cha79] and Bryant [Bry77], is among the first asynchronous *deadlock avoidance algorithms* used in conservative PDES. In the CMB algorithm, potential deadlocks are avoided by requiring the LPs to broadcast control messages, referred to as *null messages*, notifying the other LPs of the earliest time stamps when events can arrive at their input channels. These null messages provide the receiving LPs with additional information to distinguish between safe and unsafe events. The flooding of null messages, however, can dramatically degrade simulation performance. To

improve performance, different variants and optimizations of the CMB algorithm have been proposed (see, e.g., [DeV90, Woo94, Xia99, Bag00, and Par04]). A critical concept behind all of these CMB-based algorithms (and virtually all conservative synchronization algorithms in general) is the notion of *lookahead*, which is used to generate the time stamps of null messages. A LP has a lookahead of L if and only if it is guaranteed that any future event sent from the LP will have a time stamp of at least T + L, where T is the LP's current simulation time. It has been shown that the larger the lookahead, the better the performance of CMB-based conservative algorithms [Lin96, Nke01, and Per06].

In [Cha81], Chandy and Misra proposed *deadlock detection and recovery algorithms* for conservative PDES. Other techniques have also been developed to detect deadlock situations (see, e.g., [Mis86, Gro91, and Bou95]). Once detected, a deadlock can be broken by allowing the LPs involved in the deadlock cycle to process the events with the smallest time stamp. In order to enlarge the set of safe events, a global reduction computation can be used to derive a **Lower Bound on the Time Stamp (LBTS)** among the events that can be received by a LP in the future (i.e., the minimum time stamp of the next future event in the entire simulation system) [Fuj01, Per01, and McL03]. With such information, each LP can safely process any pending events with time stamps less than the LBTS value.

The success of conservative synchronization algorithms largely depends upon the ability to predict the future, in terms of the lookahead or LBTS, in order to achieve acceptable performance [Fuj00 and Tro02]. This in turn requires an effective use of *application-specific* information such as the topological structure of the network of LPs, the characteristics of the communication network, and the underlying model behavior. A side effect of this requirement is that a seemingly minor change to the model could affect the simulation performance dramatically, hindering the robustness of the application [Fuj90]. Perhaps the most prominent drawback of conservative approaches is that they often cannot fully exploit the potential parallelism available in a simulation [Fuj90], especially when the estimated lookahead or LBTS values are overly pessimistic and when global synchronizations are performed too frequently in the synchronous execution mode. Nonetheless, when the application characteristics are favorable with predictable lookahead, conservative approaches can reduce execution time significantly with moderate memory consumption (see, e.g., [Mey99, Bou00, Liu02, and Chu06]).

3.1.2. Optimistic Synchronization Algorithms

Jefferson's **Time Warp (TW)** mechanism [Jef85] is one of the first and remains the best known optimistic synchronization protocol that uses *virtual time* to model the passage of time in a simulation. A TW simulation is driven by a set of **Time Warp Logical Processes (TWLPs)**, each of which has its own **Local Virtual Time (LVT)** and processes events autonomously without explicit synchronization. TWLPs differ from ordinary LPs, such as those used in sequential and conservative simulations, in the way how the states and events are managed. Specifically, an ordinary LP maintains only one copy of its state (i.e., its *current state*), which is updated repeatedly during event execution. Furthermore, an ordinary LP does not need to keep a record of past input and output events, allowing the events to be reclaimed immediately after execution. In contrast, each TWLP needs to manage a history of its past events (both input and output) and states in three structures: ***an input queue*** that contains recently arrived input events sorted in receive time stamp order, ***an output queue*** that holds negative copies of recently sent output events (i.e., *anti-messages*) sorted in send time stamp order, and ***a state queue*** that stores the recent states of the TWLP. As will be discussed shortly, the historical data saved in these queues cannot be discarded until it is guaranteed that no event with a smaller time stamp can ever be received by any TWLP in the system. Hence, in a TW simulation, the events and states are considered as ***persistent*** in the sense that they continue to exist in the queues for a while after having been processed or updated by the TWLPs.

A causality error is manifested by the arrival of an event with a time stamp[4] that is less than the LVT of the receiving TWLP. Such an event is called a *straggler* event. Consequently, the TWLP recovers from the causality error by undoing the effects caused by those events processed speculatively during previous computation. This recovery operation is known as *rollback*. As a result, the state of the TWLP is restored to the one that was saved just prior to the virtual time as indicated by the straggler's time stamp. Since *false messages* (i.e., those generated during speculative event processing) may have spread to other TWLPs, they must be cancelled as well. Cancellation of false messages is achieved by sending *anti-messages* previously stored in the output queues. An anti-message is a copy of the original

[4] The *receive time* of an event is used interchangeably with the *time stamp* of the event, while the *send time* of an event is referred to explicitly hereafter.

positive output message with a negative flag. When an anti-message encounters its positive counterpart in a TWLP's input queue, they annihilate each other immediately, thus cancelling the positive one. If the false messages have already been processed before the arrival of anti-messages, the destination TWLPs are also rolled back, leading to further propagation of rollbacks in the system. The rollbacks caused by straggler events are referred to as *primary rollbacks*, while those triggered by anti-messages are known as *secondary rollbacks*. The time stamp of the straggler or anti-message is commonly called *rollback time*. The rollback-handling algorithms constitute the *local control mechanism* of the TW protocol.

The TW protocol also includes a *global control mechanism* that requires a distributed computation involving all of the TWLPs in the simulation system to handle such global issues as memory management, I/O operations, and termination detection. A key concept behind the global control mechanism is the **Global Virtual Time (GVT)**, which is defined as follows [Fuj00].

> ***Definition:*** *Global Virtual Time at wall clock time T (GVT_T) during the execution of a TW simulation is defined as the minimum time stamp among all unprocessed and partially processed messages and anti-messages in the system at wall clock time T.*

It has been shown that GVT never decreases, even though the LVTs can be reset frequently during rollbacks [Jef85]. That is, any TWLP will never receive an event with a *smaller* time stamp than the current GVT. Therefore, all the events and states[5] saved before the GVT can be discarded safely through a procedure known as *fossil collection* to free up the memory occupied by these historical data. In addition, I/O operations scheduled before the GVT can also be committed irrevocably. Note that the global control mechanism must estimate GVT and perform fossil collection every so often to reduce the possibility of *memory stalls*, where the simulation cannot complete because of memory exhaustion. Since GVT estimation incurs a significant overhead in terms of processor time and network bandwidth, a trade-off between TW execution efficiency and memory space usage needs to be sought in choosing the frequency of GVT computation [Jef85].

---

[5] In fact, the *last* state saved just prior to the current GVT in each state queue cannot be discarded in case that a TWLP might receive a straggler or anti-message with a time stamp that is exactly equal to the GVT.

Fujimoto summarized a few advantages of optimistic synchronization algorithms over conservative techniques [Fuj03]. First, optimistic algorithms can generally exploit higher degrees of parallelism, while conservative techniques tend to force a sequential execution when it is not absolutely necessary. Secondly, unlike conservative techniques, optimistic algorithms are less reliant on application-specific information for correct execution, even though execution efficiency can be improved if such information is available. Thirdly, since events are processed in a non-blocking manner, optimistic algorithms do not suffer from the deadlock problem as asynchronous conservative algorithms do. On the other hand, the persistent storage of historical data and the possibility of cascaded rollbacks in optimistic simulations could result in performance degradation to a certain extent.

The TW protocol has been employed in many real-world applications, achieving significant speedups in simulations of communication networks [Car95 and Ber98], battlefield scenarios [Bae92 and Wie06], biological phenomena [Pre90], and computer systems [Yau03].

3.2. Challenges of Optimistic PDES with Time Warp

In order to reduce operational overhead and improve performance of TW-based optimistic simulations, a wide variety of techniques and optimization strategies have been proposed in the literature. While these results are very encouraging, several challenging issues remain to be resolved to meet the ever-increasing demands and performance requirements in large-scale TW simulation of complex systems. This section evaluates some of the most relevant previous contributions made towards this goal.

3.2.1. Memory Management

It is well-known that TW-based optimistic parallel simulation requires more memory space to execute efficiently than an equivalent sequential simulation [Jef90, Gro91, Lin91b, Pre95, and Fuj00]. This additional memory space requirement stems mainly from the need for saving historical input/output events and states in the persistent queues during a simulation. Different memory-conserving techniques have been proposed to reduce memory consumption in TW simulations. One challenge, though, is to maintain high-performance TW execution *without undue overhead* even when the system memory is tight.

- **Fossil collection algorithms**

Fossil collection is one way to reduce the possibility of memory stalls in TW simulations. To achieve efficient fossil collection, different approaches have been attempted in the literature. For instance, Young *et al.* proposed an optimistic fossil collection technique that allows each TWLP to make its own fossil collection decisions based on locally predicted information without estimating GVT globally [You96 and You98]. Introducing optimism to fossil collection comes with a risk in that a TWLP may later be rolled back to a previous state that has already been reclaimed in the speculative fossil collection. To make this technique feasible, a recovery mechanism is used to reconstruct the erroneously reclaimed historical data. The recovery mechanism employs a distributed algorithm to create *consistent* checkpoints during a simulation, saving the states of all processes and inter-process communication channels. While this technique can reduce the cost of GVT computation, it entails extra overhead for risk prediction and fossil recovery. Young *et al.* demonstrated that the optimistic fossil collection can achieve a comparable performance with the GVT-based algorithms [You99].

A fossil identification mechanism, proposed by Chetlur and Wilsey [Che06], also attempts to reclaim fossil data without the need for GVT estimation. This mechanism uses an extended time stamp structure, known as *plausible total clock*, to record the causal relation between events. Based on this causal information, a TWLP may be able to reclaim certain historical data beyond the current GVT value, thus releasing more memory. However, this approach incurs an additional communication overhead for transferring extra causal information associated with events. Moreover, it assumes a static TWLP interconnection topology with certain special characteristics.

Vee and Hsu proposed an enhanced fossil collector, referred to as PAL, which can reduce the cost of fossil collection by prioritizing the TWLPs based on the amount of fossil data they have (thus allowing for more efficient retrieval of fossil data from the TWLPs), and by concentrating all committed events in a shared data structure so that the memory buffers can be reused later in event-saving operations (thus reducing memory allocation and deallocation overhead) [Vee02].

Although these efforts are intellectually stimulating, memory management using fossil collection alone has two main drawbacks. First, fossil collection still constitutes a significant

overhead in large-scale simulations because the operation needs to handle a large number of TWLPs and a great amount of fossil data that are usually scattered across the entire simulation system. Secondly, even frequent fossil collection cannot guarantee the absence of memory stalls if the GVT does not advance sufficiently fast, especially when the simulation needs to execute a large number of simultaneous events at each virtual time (some examples can be found in Table 3 and Table 6 of Section 5.2). This research aims to accelerate fossil collection in DEVS-based TW simulations by reducing the amount of fossil data kept in the system and by managing them in a concentrated manner for efficient batch operations.

- **Memory stall recovery algorithms**

Different approaches that aim to recover from memory stalls in TW simulations have been explored, resulting in the development of techniques such as *cancelback* [Jef90, Das93, and Aky93], *artificial rollback* [Lin91b and Lin94], and *pruneback* [Pre95]. Both cancelback and artificial rollback require a *global* pool of memory to be shared by all of the TWLPs in the system. All memory requests are granted from the pool, and released memory is returned to the pool. They differ in how to deal with situations when the pool runs out of memory. Under the cancelback mechanism, some future pending events are returned to their original senders, thus forcing the sender TWLPs to roll back and release memory. With artificial rollback, the TWLPs with the greatest LVTs (i.e., the most aggressive ones) are rolled back artificially, releasing memory in the process. The need for a common memory pool, nonetheless, makes these two approaches best-suited for shared-memory architectures.

On the other hand, the pruneback mechanism does allow for recovery from memory stalls on distributed-memory multiprocessors. Instead of using rollback as the means to memory reclamation, the pruneback mechanism releases the memory occupied by past states in the state queues, producing an effect that is similar to infrequent state-saving strategies. This technique thus incurs a cost similar to infrequent state saving as well. That is, a TWLP may have to roll back further in the past than is really necessary, and resume forward event processing from there (an action called *coast forward* in the PDES literature). In addition, the pruneback mechanism targets only past states, while the memory used by past input/output events remains unaffected.

All these techniques attempt to recover from memory stalls at the expense of a time penalty, which can be very high in certain circumstances. To address this problem, this

research attempts to reduce the amount of memory used for saving historical events and states during forward execution, without introducing an additional overhead during rollbacks.

- **Checkpointing algorithms**

Instead of trying to *recover* from memory stalls, an alternative approach is to save fewer historical data in the first place. To this end, different checkpointing algorithms have been proposed to reduce state-saving overhead. Some of them, known as *infrequent state-saving* or *periodic state-saving* techniques, focus on reducing the number of states saved in a simulation (see, e.g., [Lin93, Pre94, Ron94, Fle95, Sko96, Qua98, and Qua01a]), while others, known as *incremental state-saving* techniques, try to reduce the amount of data that need to be saved in each state (see, e.g., [Bau93, Wes96, Ron96, and Fen06]). There are also techniques that multiplex different state-saving mechanisms to improve performance (see, e.g., [Gom97 and Tay00]).

While these techniques can reduce state-saving overhead, they inevitably increase the computational cost in one form or another. For instance, infrequent state saving requires an extra coast-forward operation with a higher rollback overhead, while incremental state saving needs to keep track of the changes made to individual state variables with an increased event-processing overhead. Moreover, incremental state saving can improve performance only when a small portion of the state data is subject to modification during each event execution, making it mainly suitable for simulations such as digital logic circuits [Bau93].

- **Other memory conserving techniques**

A relatively new technique for conserving memory in optimistic parallel simulation is called *reverse computation* [Car99, Car02, Tan05b, Tan06, and Nab07], which allows the LPs to restore their states by computing the inverse operations for each event being rolled back. It has been shown that reverse computation can achieve a significant performance improvement in such simulations as queuing network models [Car99], personal communication service networks [Car02], and physical systems [Tan06]. This technique, however, usually requires annotation and manipulation of source code at the individual statement level, making it difficult to modify model logic. Although this issue can be alleviated by using advanced code transformation and compilation tools to generate reverse code automatically, certain *destructive* operations (which result in loss of data) might not be

perfectly reversible (e.g., certain bit-wise and floating-point operations) [Bau07], thus limiting the applicability of the technique. Furthermore, reverse computation potentially increases the computational cost during rollbacks, especially when a large number of events need to be processed reversely at the LPs.

In [Liu07], Liu and Wainer proposed a *user-controlled state-saving* mechanism that allows for efficient and flexible checkpointing at runtime. With this mechanism, a TWLP can make its own state-saving decisions on an event-by-event basis using application-level knowledge. A specific type of user-controlled state-saving, referred to as the **Message Type-based State-Saving (MTSS)** strategy, was implemented in the PCD++ simulator so that a TWLP can save its state only for a certain type of events in DEVS-based TW simulations, reducing memory consumption and state-saving overhead while avoiding the need for coast-forward operations during rollbacks. In other words, it is *risk-free* in the sense that, unlike other infrequent state-saving techniques, no performance penalty is incurred as the result of saving fewer states. However, as will be discussed in Chapter 4, the MTSS strategy can be enhanced to further reduce the number of states saved in DEVS-based TW simulations.

Other techniques have also been investigated to reduce memory consumption. One example is the *event reconstruction* technique proposed by Li and Tropper [Li04]. The motivation is to reduce the overhead associated with *event saving*. Based on the observation that the size of events can be very large in some cases, they suggested a method for reconstruction of both input and output events by comparing the differences between adjacent states saved in the state queues. Significant performance gain has been obtained in VLSI logic simulations. Yet it is recognized that this method works well only in a certain class of simulations with fine event granularity and small state size.

3.2.2. Cascaded Rollback

Rollback propagation can have a significant impact on the performance of optimistic PDES systems. As mentioned earlier, two types of rollbacks may occur in a TW simulation, namely *primary rollbacks* and *secondary rollbacks*. An optimistic synchronization protocol is said to be *aggressive* if primary rollbacks are allowed, while the protocol is considered to allow *risk* if secondary rollbacks are possible. The TW protocol is aggressive and allows risk. In general, both the *rollback width* (i.e., how many TWLPs are involved in a rollback

propagation) and *rollback depth* (i.e., how many events are unprocessed by a TWLP during a rollback) cannot be bounded easily. Without care, this can lead to uncontrolled rollback behavior commonly known as *domino effect* that jeopardizes the stability and scalability of the entire system. A major cause of the domino effect lies in the need for cancellation of many false messages during secondary rollbacks, which give rise to *cascaded rollback* where a large number of TWLPs are involved in the propagation [Fuj00].

- **Optimism control mechanisms**

One approach to rollback reduction is through *optimism control*, which tries to regulate inappropriate (overly) optimistic execution in TW simulations. The Moving Time Window (MTW) algorithm, which puts a bound on the difference in the LVTs of the TWLPs, is an early example of this approach [Sok91]. Many other techniques for optimism control have been developed over the last two decades, including the Breathing Time Warp protocol [Ste93], the Global Progress Window algorithm [Tay97 and Tay01], the Elastic Time algorithm [Sri98 and Qua01b], the Switch Time Warp mechanism [Sup00], and varied flow-control and learning based algorithms (see, e.g., [Sac04, Wan07 and Wan09]).

In addition, various *adaptive algorithms* have been used to improve simulation performance by adjusting specific control parameters dynamically at runtime to influence the degree of optimism (see, e.g., [Fer95, Das96, and Pan97]).

The basic idea behind all these optimism-limiting techniques is to improve event *temporal locality* in a TW simulation so that most of the events processed concurrently have relatively close time stamps. In doing so, these techniques sacrifice the degree of parallelism to a certain extent. Moreover, these techniques often rely on knowledge of certain aspects of the global simulation state in order to tune the control parameters, incurring an extra overhead for information collection and analysis during the simulation.

- **Event cancellation algorithms**

Rollback efficiency can also be improved using diverse cancellation mechanisms. The original TW protocol adopts an *aggressive cancellation* scheme that sends anti-messages immediately when a TWLP is rolled back. The *lazy cancellation* mechanism, originally proposed by Gafni [Gaf88] and subsequently analyzed by Lin and Lazowska [Lin91a], is one of the first attempts to reduce the communication overhead of event cancellation. With this

mechanism, the sending of anti-messages at a TWLP is suspended until its necessity has been verified when events are reprocessed after a rollback. However, cancellation of false messages may be delayed as a result, and additional computation and memory space is required to realize the lazy cancellation algorithms.

A *throttled lazy cancellation* scheme has been proposed recently to slow down the spread of potentially incorrect computation when events are re-evaluated during lazy cancellation operations [Sol08], but only at the expense of increased communication cost for broadcasting special control messages to block and unblock the TWLPs (which is not unlike the mechanism used in the Wolf Calls protocol originally proposed by Madisetti *et al.* to contain error propagation in TW simulations [Mad88]). Furthermore, broadcasting control messages is not without its limitations. For one thing, it may block some TWLPs unnecessarily. For another, the effectiveness of this mechanism depends on the relative speeds at which errorneous computation and control messages may spread.

To further reduce the communication overhead of event cancellation, an optimization strategy called *early cancellation* can be used to cancel false messages *in place* in the buffer of a programmable network interface controller [Nor02], which demonstrates the potential of using specialized hardware to improve TW performance, but also limits the utility of the strategy in TW simulations on general-purpose computing platforms.

Other studies have shown that cancellation performance can be improved by capturing the causal relationship between events [Che01]. By exploiting event causal dependency, a *proactive cancellation* mechanism was developed that can be used to prevent cascaded rollbacks [Che09a]. Using a similar strategy, a *batch-based cancellation* algorithm was proposed that allows a TWLP to recover from a causality error with at most one rollback [Zen04]. However, this algorithm introduces extra communication overhead for exchanging causal inforamtion and rollback histories between TWLPs as well as additional computation overhead for reclaiming these data during fossil collection.

- **LP aggregation techniques**

Besides, different *LP aggregation* techniques have been investigated to mitigate the overhead of event cancellation. One example is the *Local Time Warp protocol* proposed by Rajaei *et al.* [Raj93 and Raj07]. In this protocol, the global simulation space is divided into several sub-regions referred to as clusters, each of which contains a set of TWLPs. While the

TWLPs *within* a cluster are executed optimistically based on the TW mechanism, the clusters themselves are synchronized in a conservative fashion, thus preventing false messages from propagating beyond cluster boundaries. As a result, the problems of cascaded rollbacks and memory stalls need to be handled only locally. However, the Local Time Warp protocol, like many other hybrid approaches that combine both conservative and optimistic algorithms in a simulation, may suffer from reduced parallelism. Furthermore, it requires careful control to balance the optimistic local simulation of individual clusters with the global virtual time horizon established by the conservative algorithm [Bou05].

The *clustered adaptive-risk* technique, proposed by Soliman and Elmaghraby [Sol96 and Sol97], is another example that aims to control the degree of risk in TW simulations. Similar to the Local Time Warp protocol, the TWLPs are grouped into clusters. Nonetheless, instead of applying a conservative synchronization protocol at the global level, the technique uses an adaptive algorithm to keep the probability of cross-cluster rollback propagation below a certain user-defined threshold. This is achieved by tuning the intervals between the release times of buffered inter-cluster messages based on observed simulation behavior at runtime, at the cost of additional computation and memory space overhead.

While the above techniques can be used to improve rollback performance, every TWLP in the system is still subject to rollback operations that are triggered either locally or globally, thus requiring each TWLP to maintain its persistent event and state queues, just like in the original TW mechanism.

To enhance the performance of TW-based digital logic simulations, Avril and Tropper introduced a *Clustered Time Warp protocol* that uses the TW protocol to synchronize clusters of LPs globally, whereas the execution of LPs in each cluster is scheduled sequentially by a *cluster environment* with the aid of a *time zone table* (to detect changes in virtual time), a *cluster input queue* (to receive events from other clusters), and a *cluster output queue* (to hold anti-messages that might be sent to other clusters during rollbacks) [Avr95]. The rationale behind this approach is that a logic circuit can be partitioned naturally into different functional units, each of which will then be simulated on a distinct processor. Two types of rollback mechanisms are defined in the protocol, referred to as *clustered rollback* and *local rollback*. Under the former mechanism, rollbacks are handled at the cluster level. When a straggler or anti-message is received by a cluster, all of the LPs

included in the cluster are rolled back together if they have executed an event with a time stamp greater than the rollback time. Although this mechanism can reduce memory consumption (since individual LPs do not need to keep anti-messages in their output queues, and all input events with time stamps greater than the rollback time are discarded during rollbacks), some LPs may be rolled back unnecessarily. Under the latter mechanism, the cluster environment simply forwards the received straggler or anti-messages to the destination LPs so that they can make rollback decisions individually. In this case, the LPs must maintain anti-messages in their output queues; and rollbacks are carried out in the same way as in the original TW protocol. In the same vein, two checkpointing mechanisms are defined, including the *clustered checkpointing* mechanism that saves states for the LPs only when remote events (from other clusters) are received, and the *local checkpointing* mechanism that saves the state for a LP whenever the simulation time is changed in the time zone table, regardless of whether the time change is caused by a local event or a remote one. Since both mechanisms use infrequent state saving, coast-forward operations are required during rollbacks with the associated overhead. Using several digital circuit models as benchmarks, different combinations of these rollback and checkpointing mechanisms have been evaluated quantitatively in [Avr01]. The experiments showed that, while memory usage can be reduced by up to 40% in some cases, the execution time is comparable or even worse than obtained with the original TW protocol, indicating that a trade-off must be made between execution efficiency and memory conservation.

In this research, cascaded rollback is controlled by virtue of a lightweight rollback mechanism, which enables most of the LPs to recover from causality errors without the need for event cancellation, allowing for purely optimistic TW simulation with bounded rollback propagation on each node, as will be discussed in Chapter 4.

3.2.3. Event Management

A central issue that needs to be addressed in any discrete-event simulation system is event management, which has been studied extensively in the literature from different aspects (e.g., event scheduling, event ordering, and event set implementation, just to name a few). The following discussion briefly reviews two of these event management problems that are most relevant to the research presented in this dissertation.

- **Event set implementation**

The relative performance of different event set data structures and algorithms has been a topic of research since the early days of discrete-event simulation [Com79 and McC81]. The need for handling potential rollbacks in optimistic parallel simulations makes event management more complex than in a sequential simulation, mainly because past events that have already been processed remain in the event queues. As a result, efficient insertion and retrieval of both *historical* and *future* events become necessary for the overall simulation performance [Ron93].

Numerous non-trivial data structures have been investigated in the context of TW-based optimistic simulations. Most of these data structures can be characterized into three broad categories: *list structures*, *tree structures* and *multi-list structures*. Examples of list structures include the Indexed Lists [Nik93] and the SPEEDES Queue [Ste96]. Tree structures are exemplified by Binary Heaps, Skew Heap, and Splay Trees [Jon86 and Jon89], while the Lazy Queue [Ron91], the Ladder Queue [Tan05a], and the Calendar Queues [Bro88, Oh97, Oh99, and Tan00]) are based on multi-list structures. A primary motivation behind these efforts is to achieve efficient event queue operations as the number of events stored in the event queues increases in large-scale and fine-grained simulations. While these techniques have proven to be quite useful in improving performance, an attractive alternative solution is to keep the event queues relatively short throughout a simulation, an approach that is adopted by the technique presented in Chapter 4 of this dissertation.

- **Simultaneous events**

The way how *simultaneous* events are managed in discrete-event simulation can have serious implications on simulation correctness, reproducibility, and performance [Wie97, Ron99, and Fuj00]. As noted by Jha and Bagrodia [Jha00], simultaneous events may occur in a discrete-event simulation for three general reasons. First, the physical system may include many independent activities. As a result, when the system is decomposed into a set of LPs, it is convenient to represent the interactions between different portions of the system as simultaneous events. Secondly, the limited resolution of simulation time can also lead to events with the same time stamp even though these events are not truly parallel in the physical system. Finally, the need for modeling activities with zero delay time (or a delay

that is negligible compared to the duration of other activities being modeled) often necessitates the use of zero-delay LPs that generate output events with the same time stamp as the received input events.

Many previous studies have been devoted to *ordering* of simultaneous events (see, e.g., [Agr91, Meh92, Ron99, and Jha00]). Two types of tie-breaking mechanisms are commonly used in discrete-event simulation [Ron99 and Jha00]. One of them, referred to as *user-consistent and deterministic*, bundles all of the simultaneous events received by a LP and executes them based on a set of protocol-independent, user-specified rules. The other, referred to as *arbitrary and deterministic*, relies on the simulation protocol to choose a well-defined implicit ordering of the events. Still, some researchers argue that the appropriate way of handling simultaneous events is to take *all* possible orderings into account when evaluating the simulation results, rather than forcing the users or the protocols to choose an ordering that may not always serve well the intention of the simulation [Wie97 and Pes07a]. Various approaches have been taken to implement tie-breaking mechanisms in the context of PDES. Some of them extend the time stamps of event messages to impose a *ranking* on the simultaneous events for deterministic execution [Agr91, Meh92, and Wan06], while others employ the concept of *aging* for the same purpose [Meh92].

The simultaneous event problem has also been tackled in DEVS simulations. As discussed in Chapter 2, the classical DEVS formalism achieves tie-breaking using the *Select* function. Kim *et al.* proposed a protocol-transparent scheme to implement the *Select* function in distributed simulation environment [Kim97]. The P-DEVS formalism realizes user-consistent and deterministic tie-breaking by virtue of the *bag* structure to bundle the external events scheduled for a LP at each virtual time, and by the *confluent state transition* function to define the rules for resolving transition collisions. A P-DEVS simulator is responsible for collecting all of the simultaneous external events in the bag of an atomic model so that these events can be processed *as a whole*, necessitating the use of a control flow to ensure proper event delivery at each virtual time, as illustrated by the PCD++ event-processing algorithms presented in Section 2.6.

Although these techniques provide a well-founded basis for handling simultaneous events in PDES, the performance consequence of processing a large number of simultaneous events at each virtual time in a simulation has not yet attracted enough attention from the

research community. This performance issue is especially important in large-scale TW simulations. Without careful design and proper control, the expanded execution of simultaneous events could have a detrimental effect on TW performance in terms of increased overhead for state-saving, rollback, fossil collection, and dynamic load migration. This research addresses the above performance issue by proposing an event management scheme that is especially adept at efficient execution of simultaneous events in DEVS-based TW simulations, a topic that will be discussed in Chapter 4 of this dissertation.

3.2.4. Dynamic Process Migration

Dynamic load-balancing algorithms typically rely on the runtime system state information to make decisions regarding the movement of workload from one processor to another during execution. According to Willebeek-LeMair and Reeves [Wil93], dynamic load balancing can be organized as a process consisting of four major components, including (1) processor load evaluation; (2) load balancing profitability determination; (3) load migration strategy, and (4) load selection strategy. All of these components have been studied extensively in the PDES literature, leading to the development of a large number of dynamic load-balancing algorithms. An exhaustive review of these load-balancing algorithms is beyond the scope of this dissertation (but, see, [Rei90, Gla93, Jia94, Sch95, Bou97, Wil98, El-K99, Car00, Iko00, Low02, Li04, and Pes07b] for related works on this topic). Instead, the following discussion summarizes some of the efforts that aim to facilitate *load migration* (also known as *process migration*) in TW simulations. While dynamic load balancing is concerned primarily with distributing the workload as evenly as possible among the processors, process migration focuses on the operation of load transfer to achieve a certain load-balancing objective.

To minimize the communication overhead and the interference with normal system execution, agile process migration is recognized as crucial to efficient dynamic load balancing in parallel and distributed systems [Art89]. Even more so in large-scale TW simulations where a potentially unbounded amount of event and state data associated with a TWLP must be transferred between processors. An early work, presented by Reiher and Jefferson [Rei90], employed a *phase-based* computation model to reduce process migration cost in the Time Warp Operating System. In this model, the life span of a TWLP is divided into multiple phases, each representing a portion of the execution history of the TWLP

during a specific interval of virtual time. These phases can be transferred individually across processors as needed in order to implement a finer-grained load balancing scheme below the LP granularity. Yet this computation model suffers from increased overhead for message routing and scheduling during both forward execution and rollbacks because each TWLP can consist of many small fragments scattered all over the system.

Different dynamic load balancing and process migration algorithms have been investigated in the SPEEDES simulation framework [Wil98]. These algorithms use a central coordinator to make global load redistribution decisions regularly during a simulation. If load migration is warranted, the actual load selection and transfer operations are then handled by each pair of chosen nodes at the LP level. With the Breathing Time Warp protocol [Ste93], a parallel simulation is executed in a cyclic fashion. Each simulation cycle starts with a purely optimistic TW execution, but then switches to a risk-free synchronization mode using the Breathing Time Buckets algorithm [Ste91]. After the risk-free execution stage, a new GVT value is computed, followed by the reclamation of historical state and event data. Therefore, the amount of data to be transferred can be minimized if load migration is carried out at the end of a simulation cycle. This data minimization strategy, however, has two main drawbacks. First, the conservative risk-free execution might reduce the degree of achievable parallelism in a simulation. Secondly, the success of this strategy still depends on GVT computation and fossil collection, which, if performed too frequently, could adversely affect simulation performance.

Based on the Clustered Time Warp concepts, Avril and Tropper developed a dynamic load-balancing algorithm that transfers all of the TWLPs included in a cluster as a group [Avr96]. Although this approach makes it easier to implement the load-balancing algorithm since migration decisions are made only for clusters (instead of for individual TWLPs), it reduces the flexibility of the mechanism and shifts some of the responsibilities to the users, who also need to consider load-balancing issues when partitioning the model.

In a more recent study, Li and Tropper argued that the event reconstruction technique, originally intended for memory conservation in Clustered Time Warp, can also be used to facilitate process migration because only the state data associated with a TWLP need to be transferred [Li04]. Nonetheless, the study did not indicate whether the proposed migration scheme would be realized at the cluster level or at the individual LP level. In addition, this

migration scheme can be applied only to models with certain event and state characteristics that motivated the development of the Clustered Time Warp in the first place.

As will be discussed later, the technique presented in Chapter 4 of this dissertation can be used to facilitate dynamic process migration in DEVS-based TW simulations by reducing the amount of data that need to be transferred across cluster nodes.

3.2.5. DEVS-based Time Warp Simulation

A number of studies have been devoted to DEVS-based TW simulations. The DEVS-Ada simulator, developed by Christensen [Chr90], is one of the first efforts towards optimistic parallel DEVS simulation using the TW protocol. In DEVS-Ada, a DEVS model is partitioned into several major *coupled* models (each can have a hierarchy of atomic and coupled subcomponents). A major coupled model is then wrapped into a TWLP that executes on a distinct node. Although this coarse-grained partition scheme may reduce event-scheduling complexity as the messages sent between the subcomponents of a major coupled model are treated as internal events of the corresponding TWLP, the enlarged process granularity not only makes it difficult to realize dynamic load balancing, but also increases checkpointing and rollback overhead since some of the subcomponents may be required to perform state-saving and rollback operations unnecessarily.

Seong *et al.* developed another optimistic parallel DEVS simulator called P-DEVSIM++ [Seo95], which employs the TW protocol to synchronize the execution of both external and internal events in DEVS simulations. The key idea is to use the TW rollback mechanism to ensure that transition collisions are handled properly according to a given DEVS *Select* function.

In [Kim96], an event-scheduling algorithm was proposed that schedules the execution of DEVS models hierarchically on each node, while using the TW protocol for global synchronization. The model-partitioning issue raised in such parallel simulation has been discussed in [Kim98], resulting in a hierarchical partitioning algorithm for improved load balancing in the simulation. Nonetheless, all of the above-mentioned studies discuss TW-enabled optimistic parallel simulaiton in the context of the *classical* DEVS formalism, which, as discussed in Chapter 2, hinders the exploitation of potential parallelism in a simulation because of the serialization constraint imposed by the tie-breaking *Select* function.

Based on the P-DEVS formalism, Nutaro proposed a risk-free optimistic simulation mechanism to reduce the possibility of rollback thrashing in simulating P-DEVS approximation of continuous systems [Nut04]. With this risk-free algorithm, a TWLP must withhold the sending of speculative inter-process output events until it is certain that doing so will not cause rollbacks at the destination processes. This is achieved by allowing a TWLP to emit only those output events with time stamps right after the current GVT. While this approach can contain rollback propagation and improve performance so long as inter-process communication is infrequent, it limits the achievable parallelism and increases operational overhead as GVT must be computed every so often in a simulation.

Nutaro also proposed an abstract simulation algorithm for correct TW execution of a *flat* P-DEVS coupled model that consists of interacting atomic models [Nut08]. The correctness proof focuses on the high-level operations performed by each TWLP, which encapsulates the logic of a corresponding P-DEVS atomic model. Nutaro proved that, in theory, the proposed algorithm produces the same input-output behavior (up to the GVT) as would be exhibited by a sequential abstract DEVS processor. It is suggested that this abstract simulation algorithm can thus provide a basis for correct TW simulation of DEVS-based models. However, implementation and performance issues are not considered in the work.

In [Sun08], Sun and Nutaro implemented a modified TW algorithm to improve the performance of P-DEVS simulations on *shared-memory* multiprocessor computers. The improvement is obtained by executing the P-DEVS output (λ) and state transition functions (δ_{int}, δ_{ext}, and δ_{con}) at the due atomic models synchronously and in parallel at each virtual time. The experiments demonstrated that the simulation of a compute-intensive forest fire propagation model can run up to 4 times faster on 8 processors over the equivalent sequential implementation. However, the proposed algorithm cannot deliver much performance gain in models with lower computation intensity (in some cases, the parallel simulation runs even slower than the sequential one). Moreover, the algorithm does not attempt to address the various issues involved in TW simulations, such as those mentioned in the previous sections.

By taking advantage of the intrinsic computational properties of the DEVS-based simulation process, a **Lightwight Time Warp (LTW)** protocol is proposed in Chapter 4 to addresses the above-mentioned challenges in a systematic way. The performance impact of the LTW protocol is evaluated quantitatively in Chapter 6 and summarized in Chapter 7.

3.3. Challenges of Efficient PDES on Cell Processor

The IBM Cell processor [Kha05 and Che07] represents a departure from the traditional design of microprocessor architectures, and offers new opportunities for effective control of hardware resources in return for optimized performance. The architectural features of the Cell processor are also attractive for studying computing paradigms on *heterogeneous* CMP platforms. As will be discussed in the following subsections, the irregular computation and complex data dependency commonly found in PDES pose significant challenges to software development on heterogeneous CMP architectures, requiring innovative redesign of existing simulation algorithms to realize the full performance potential on such platforms.

3.3.1. Asymmetric Architecture with Explicit Memory Control

The latest Cell processor adopts a heterogeneous CMP architecture with nine independent processing elements (or cores): a main dual-threaded **Power Processor Element (PPE)** and eight specialized co-processors called **Synergistic Processing Elements (SPEs)**. These cores are tightly coupled by a high-bandwidth, low-latency, on-chip **Element Interconnect Bus (EIB)** [Kis06]. While the PPE uses a *hardware-controlled* conventional cache hierarchy (32KB L1, 512KB L2) to access the system main memory, each SPE can directly access only a private, non-coherent, on-chip **Local Storage (LS)** of 256KB that contains both code and data (including the runtime call stack) of an SPE thread at any time. Data sharing between the main memory and the LS of an SPE is achieved mainly through *software-managed*, *explicitly-addressed* **Direct Memory Access (DMA)** transfers, which are handled autonomously by a **Memory Flow Controller (MFC)** associated with each SPE. An SPE can also communicate 32-bit short messages with the other cores over the EIB channels, including *mailboxes* (one 4-entry inbound channel, and two single-entry outbound channels) and *signals* (two single-entry inbound channels)[6]. Moreover, the SPEs support both scalar and 128-bit SIMD operations that can be applied at 2, 4, 8, and 16-way granularities. An architectural overview of the Cell processor is given in Figure 11 [Are08].

[6] An *inbound* channel is used to send messages to the local SPE from the PPE or other SPEs, while an *outbound* channel is used to send messages in the opposite direction.

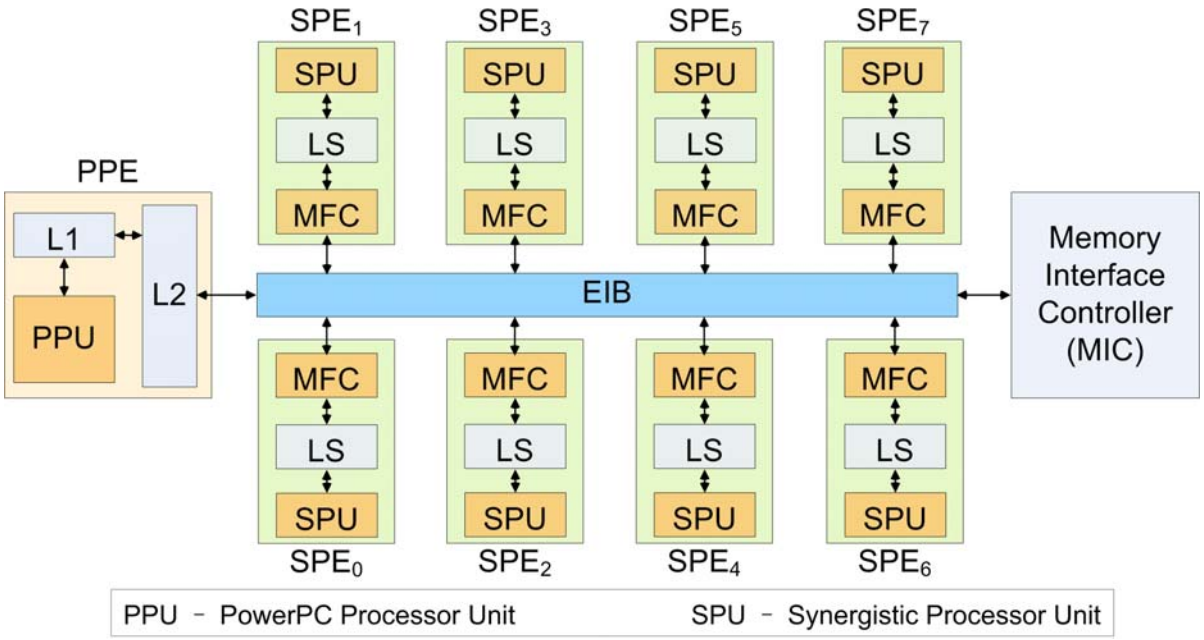


Figure 11. Cell Processor Block Diagram

The design of the Cell processor presents a set of features that must be taken into consideration when developing high-performance algorithms. These architectural features are outlined as follows, while an extensive documentation on the Cell platform can be found at [IBM10a].

- **Heterogeneous processing elements**

The two types of cores integrated on a Cell processor are designed to provide different functionalities. As a general-purpose core based on the 64-bit PowerPC ISA, the PPE is intended to execute *control-intensive* code. It is used to host the operating system and to orchestrate the execution of different threads involved in a parallel application. On the other hand, the SPEs are optimized to execute *compute-intensive* code based on a 128-bit SIMD ISA. As a result, it is necessary to partition the computation of a Cell application according to the functional specialization of the cores. To attain good performance, the most compute-intensive portion of the application code (organized as *computational kernels*) must be offloaded to the SPEs. In addition, an efficient SPE implementation should also take advantage of the SIMD instructions to streamline the kernel computation. However, the workload of a general-purpose PDES system depends not only on the computational properties of the simulator, but also on the model behavior. Hence, an in-depth analysis and generalization of the simulation workload characteristics are required to match the hardware

capabilities. The irregular PDES computation also complicates the SIMDization effort when porting simulation code to the SPEs.

- **Limited LS size with explicit memory control**

An SPE thread relies on its LS to execute a designated computational kernel, which in many cases has a memory footprint that is much larger than the size of the LS, necessitating the division of the kernel workload into a sequence of *working sets* small enough to fit into the on-chip local memory. Moreover, an application is responsible for scheduling DMA data transfers across memory domains (i.e., main memory and LS) explicitly during the computation, requiring a careful balancing of the computation and communication granularity to hide memory latency and ensure timely data availability [Lev07 and Var07]. Furthermore, a DMA transfer also has its own requirements with respect to *data address alignment* and *transfer size*. In particular, when the transfer size is 16 bytes or larger (up to 16KB), the addresses of the data in both memory domains must be aligned to at least 16-byte boundaries (peak performance is obtainable when the addresses are aligned on 128-byte cache-line boundaries) [Are08]. These DMA transfer requirements can lead to increased software complexity for handling such issues as data padding, address alignment, and macroscopic data prefetching (e.g., double-buffering).

- **Multiple levels of parallelism**

In order to harness the Cell potential, an application needs to *simultaneously* exploit parallelism at multiple levels of the system. As noted by Gschwind [Gsc06 and Gsc07], the Cell processor offers at least five distinct forms of parallelism that can be utilized in an application, including (1) **thread-level parallelism** with hardware multithreading on the PPE and across multiple SPEs; (2) **data-level parallelism** by virtue of extensive SIMD instruction support on the SPEs; (3) **compute-transfer parallelism** using autonomous, programmable MFC engines; (4) **memory-level parallelism** by overlapping and interleaving multiple DMA transfer requests from each core as well as from multiple cores; and (5) a certain degree of **instruction-level parallelism** enabled by a statically-scheduled, power-aware, dual-issue microarchitecture. Based on application-specific knowledge, a software developer thus needs to determine the appropriate form and the right amount of parallelism that should be exploited at each system level in an application.

- **SPE programming considerations**

Several practical programming considerations also need to be taken into account when developing algorithms on the SPEs [IBM09]. First of all, only one thread can be executed by an SPE at any time, and context switch on these co-processors is considered as very expensive in terms of both time and system resources. For this reason, an SPE is usually assigned with a *reusable* task that operates on a stream of data. Secondly, the SPEs do not support hardware dynamic branch prediction, assuming that all branches will *not* be taken. If a branch is taken, however, a significant performance penalty is incurred. Hence, the SPE code should be designed in a way that either avoids (or at least minimizes) branching instructions or uses explicit branch hints to assist the compiler to improve instruction prefetching. Thirdly, C++ exception handling and I/O streams are not supported on the SPEs, and the standard C++ **STL (Standard Template Library)** container classes should be used with caution since the generated binary image can easily exceed the size of the LS. Therefore, it is a common practice to develop SPE programs in C and organize LS data in array-based buffers. Finally, all SPE memory accesses operate on 128-bit *quadwords*, which must be aligned on 16-byte boundaries in the LS. A sub-quadword scalar value is kept in a specific *preferred slot* of a SIMD register, thus requiring the compiler to generate extra rotation instructions to align the data, if necessary, at the cost of inflated code size and reduced performance. To minimize this extra rotation cost, scalar values should be either replaced by quadword vectors, or aligned and padded properly whenever possible to meet the SPE data layout requirement.

3.3.2. Multi-Grained Parallelization Strategies

As mentioned earlier, the performance of a Cell application depends on effective exploitation of *multi-grained* parallelism (in terms of both form and amount) at different system levels. Exploring multi-grained parallelism on the Cell processor has been investigated recently in a number of studies. For instance, Williams *et al.* demonstrated the potential of the Cell processor for scientific computing by combining three forms of parallelism (i.e., *task* parallelism, *pipelined* parallelism, and *data* parallelism) in several scientific computational kernels [Wil06]. Blagojevic *et al.* proposed an adaptive algorithm for dynamically choosing the form and degree of *task*-, *data*-, and *loop-level* parallelism in bioinformatics applications

[Bla07a, Bla07b, and Bla08]. Petrini *et al.* optimized the performance of a radiation transport application by exploiting several dimensions of parallelism such as *thread-level* parallelism, *data-streaming* parallelism, *vector* parallelism, and *pipeline* parallelism [Pet07].

While these studies provide valuable insight, it remains a challenging task to combine multi-grained parallelism coherently in general-purpose PDES on the heterogeneous Cell processor. To begin with, a PDES system typically includes different types of computational kernels such as LP synchronization and event execution, each with its own specific data access pattern and workload characteristics (e.g., compute-bound or memory-bound), requiring different parallelization strategies to be developed for different types of kernels. Moreover, most of the existing PDES techniques take a coarse-grained parallelization strategy at the LP level to exploit parallelism across multiple nodes of a parallel computing system, without paying much attention to other forms of parallelism available on multicore processors. In some cases, this *LP-oriented* strategy might actually make the task of kernel parallelization more difficult to achieve because a computational kernel could be implemented collectively by several LPs and a LP could also encapsulate different types of kernel computation. In addition, the complex causal and data dependency involved in a PDES program further increases the complexity of the parallelization task.

3.3.3. Abstract Programming Models

A programming model is an abstract description about how a computation is executed. A variety of programming models have been developed for different parallel computing platforms [Ski98]. Broadly speaking, most of the existing parallel programming models are based on either *distributed-memory message passing* or *shared-memory multiprocessing* paradigms. Although these paradigms are often applied to program conventional multiprocessor systems, there has been a growing interest in extending their use to emerging CMP architectures like the Cell processor.

In a recent work [Sta08], Stamatakis and Ott studied the performance of a bioinformatics application using **MPI** [Gro99], **Pthreads** [Nic96], and **OpenMP** [Cha07] based parallelization strategies on different parallel architectures, including AMD Barcelona system (2-way, quad-core), Intel Clovertown system (2-way, quad-core), Sun x4600 system (8-way, dual-core), SGI Altix 4700 system (2-way, dual-core), and a cluster of **SMP**

(Symmetric Multiprocessing) AMD Opteron processors (4-way, single-core). Based on the test results, the authors concluded that there is *no* universally best-suited programming model with respect to execution efficiency and code portability for the studied application. Instead, they suggested that the selection of programming models should rest on software engineering criteria and promoting data locality in the application.

McCool presented a general discussion of the scalability and portability issues associated with different parallel programming models for a range of CMP architectures, with a focus on **Graphics Processing Units (GPUs)** and the Cell processor [McC08]. Some of the programming models, exemplified by **MIMD (Multiple Instruction, Multiple Data)**, exploit *task-level parallelism* by decomposing a program into separate tasks that run concurrently on different processing elements. Communication and synchronization of these tasks can be achieved by either message passing or shared memory regions. Other programming models expose *data-level parallelism* following a data decomposition strategy. One simple example in this category is the **SIMD (Single Instruction, Multiple Data)** model, which can be generalized as the **SPMD (Single Program, Multiple Data)** model where the computational kernel may also include control flow with branching conditions instead of just a sequence of in-order data manipulation instructions. A widely used data-level parallel programming model on CMP architectures is known as the **stream programming** paradigm, which constructs a parallel computation from a *data-flow* perspective where compute-intensive kernels operate on streams of input and output data that are typically organized in contiguous arrays [Thi02, Gum08, and Kud08]. Data locality is enhanced both spatially (as *contiguous* data streams) and temporally (with a *producer-consumer* style of data processing between kernels).

To improve programmability and accessibility of the architectural features of the Cell processor, Khale *et al.* proposed six programming models [Kha05], including (1) **function offload model** where the SPEs are used to accelerate certain compute-intensive functions dispatched by a main application running on the PPE; (2) **device extension model** in which the SPEs serve as intelligent front ends to external devices using direct memory mapping; (3) **SPE-centric computational acceleration model** where the SPEs execute the bulk of the application code under the control of the PPE; (4) **streaming model** in which the SPEs are organized as a pipeline to process the data that stream through it and the PPE is used as a

stream controller; (5) **shared-memory multiprocessor model** where a Cell processor is viewed as an asymmetric shared-memory multiprocessor with two distinct ISAs; and (6) **asymmetric thread runtime model** in which an application is constructed as a pool of threads, each of which can be scheduled on either the PPE or the SPEs at runtime based on the architectural characteristics of the cores.

Although these abstract parallel programming models provide general guidelines for developing new computing techniques for PDES, significant efforts are still required to customize and integrate them effectively to address the specific needs of different computational kernels when porting a PDES system to the Cell platform.

3.3.4. Automated Compilation Techniques

One way to facilitate software development on CMP architectures with explicit memory control is to use advanced compilation techniques. For instance, the IBM XL C/C++ Compiler for Multicore Acceleration [IBM08a] includes several automated techniques that allow for compiler-assisted branch prediction, instruction prefetching, data and code partitioning, and SPE code vectorization on the Cell processor [Eic05 and Eic06]. Knight *et al.* developed an optimizing compiler for processors with a hierarchy of memories managed explicitly by software [Kni07]. The proposed compiler addresses issues such as data padding, software pipelining, and memory space allocation for programs written in the *Sequoia* programming language [Fat06], which provides an abstraction of the memory hierarchy as private address spaces allocated to different concurrent tasks.

Without a full understanding of high-level application logic, these compilation techniques usually support performance optimization only at data or function levels. While this approach works well for certain types of applications, it is inadequate on its own to handle the highly irregular computation in PDES systems whose complex data dependency is revealed only at runtime. On the other hand, compiler-assisted code optimization can help further improve PDES performance when application-level parallelism has already been properly analyzed and extracted.

3.3.5. Middleware Frameworks

Efforts towards providing a layer of abstraction on top of the Cell programming primitives have led to the development of several middleware frameworks. Some of these frameworks are briefly reviewed as follows.

The **RapidMind** framework [McC06] is an embedded programming language inside C++ that employs a simple *data-parallel* programming model built around a few C++ container types, including `Value<N, T>` that holds N values of type T, `Array<D, T>` that represents a multi-dimensional (1, 2, or 3) array of data elements with type T, and `Program` objects that encapsulate a sequence of operations. A parallel computation is executed by applying programs to arrays, and consequently generating new arrays. RapidMind also includes runtime components for handling certain low-level execution details such as task queuing, data streaming, task synchronization, and load balancing. The framework can be used to develop a class of high-performance applications on both GPUs and the Cell processor.

The **Cell Superscalar (CellSs)** framework [Bel06 and Per07] allows for exploiting *functional parallelism* of a sequential application on the Cell processor based on user-supplied annotation (similar to those used in OpenMP) in the source code. An annotated code segment represents a task that *can* be executed on the SPEs. To exploit parallelism, CellSs uses a runtime component that builds a data dependency graph, reflecting the dynamic relationship between the isolated tasks. Independent tasks are then scheduled on different SPEs for concurrent execution, while multiple dependent tasks may be co-scheduled on the same SPE to facilitate data reuse. However, CellSs does not support code SIMDization and other low-level code optimization, but instead relies on either the programmer or a native Cell compiler (e.g., IBM XL Compiler) to improve application performance manually or semi-automatically.

Currently, MPI is considered as the *de facto* standard for parallel programming on conventional distributed-memory multiprocessor architectures. To enable certain existing MPI-based parallel applications to be ported to the Cell processor, Kumar *et al.* conducted a feasibility study in which *certain core MPI functionalities* (e.g., blocking point-to-point and collective messaging) are implemented using Cell programming primitives [Kum07]. The SPEs are used as a group of nodes for hosting the MPI processes, while the application data

are actually stored in the main memory to circumvent the LS size limitation. The **MPI Microtask** framework is another effort towards enabling MPI applications on the Cell processor [Oha06]. With MPI Microtask, programmers do not need to explicitly manage the on-chip LS as long as they can partition a MPI application into a set of small chunks called *microtasks* that can fit into the LS. The framework includes a preprocessor that transforms a MPI program defined in microtasks into a Cell program based on the streaming model. A *resident* runtime environment (which occupies the lowest 16KB memory in the LS) is then used to handle task synchronization, context switch, and message buffer management.

The IBM **Software Development Kit (SDK)** for Multicore Acceleration [IBM08b] also includes an **Accelerated Library Framework (ALF)** that attempts to facilitate software development on the Cell processor by providing an **Application Programming Interface (API)** for *data-* and *task-parallel* libraries and applications [IBM08c]. An application is decomposed into two types of tasks: *control tasks* are executed on the PPE, whereas *compute tasks* are mapped to the SPEs. The framework has two runtime libraries, referred to as *host runtime* and *accelerator runtime*, which provide a set of key services such as task management, double-buffered data transfer, and certain error handling capabilities. The ALF framework is part of an on-going development project that is being actively pursued by IBM. Some early studies indicate that the current ALF implementation may shy of a high enough level of abstraction and a full support for varied SPE communication patterns [Cra08].

To summarize, these middleware frameworks have exhibited notable success in a range of applications, such as image processing, list ranking and sorting, matrix multiplication, and fast Fourier transformation. Yet some of them assume a strict data-parallel programming model or adhere to a pure C programming language, while others provide only a minimal set of functionality of a standard library for specific applications. These limitations greatly hinder their applicability to complex object-oriented PDES systems. Besides, they usually rely on some runtime components to support parallel execution, introducing a nonnegligible overhead in terms of both time and memory space.

3.3.6. Application Development on Cell

Various applications have been developed on the Cell platform to study performance characteristics and to serve as examples of porting legacy software to the heterogeneous Cell

architecture. Some examples include key scientific computational kernels [Wil07], radiation transport algorithms [Pet07], wavefront algorithms [Aji08], Fourier transform [Bad07a and Che09b], regular expression scanning [Sca09b], list ranking [Bad07b], image processing [Sai07 and Ara09], and molecular dynamics [Oli07], among many others. Most of these Cell applications optimize performance by exploiting *application-specific* perallelism (typically at the task and data levels) based on careful analysis of the underlying algorithms and *a priori* knowledge of the workload characteristics.

The Cell processor has also been used to host M&S applications. For instance, Meredith *et al.* improved the performance of bio-molecular simulations by farming out the most compute-intensive molecular dynamics calculation to the SPEs of a Cell processor [Mer07]. In a similar vein, De Fabritiis optimized the performance of bio-molecular simulations by parallelizing the code for computing non-bonded force field using SPE SIMD intrinsics, while the majority of the simulation code remains unchanged on the PPE [DeF07]. Yet another example of Cell-accelerated bio-molecular simulation was presented by Shi and Kindratenko [Shi08], in which the authors implemented the NAMD SPEC 2006 CPU benchmark [SPEC10] on the Cell processor following the function offload programming model. Porting these bio-molecular simulations to the Cell processor is relatively straightforward due to the fact that there is only *one* dominant *numerically-intensive* computational kernel in the simulation and the kernel computation is rather compact (less than 500 lines of C code), making it easy to fit into the on-chip LS.

Several studies have focused on lattice Boltzmann simulation on the Cell processor. Peng *et al.* proposed a parallel lattice Boltzmann algorithm that divides the simulation into several sub-domains, each of which is represented by a three-dimensional floating-point array mapped to a distinct Cell processor [Pen08]. The values within each sub-domain are processed by multi-threaded collision and streaming functions executed concurrently across the SPEs of a Cell processor, while inter-domain communication is handled by the PPE using MPI messaging. However, the proposed algorithm does not utilize SIMD programming to further parallelize the SPE code, and the performance gain is mainly achieved by allowing the SPE threads to process independent data in parallel. In [Wil08], Williams *et al.* developed an application-specific auto-tuning environemnt for performance optimization of lattice Boltzmann simulations on different multicore architectures, including the Cell

processor, Intel Clovertown, AMD Opteron X2, and Sun Niagara2. The optimization strategies include loop unrolling, code reordering, software-controlled data prefetching, and SIMD vectorization. The experimental results indicate that the Cell processor can achieve high raw performance gain and power efficiency for demanding *numerical simulations* such as lattice Boltzmann computations.

Financial modeling is another field that has attracted attention from the multicore high-performance computing community. Agarwal *et al.* developed efficient parallel pseudo- and quasi-random number generators as well as normalization techniques, which are then used to accelerate Monte Carlo simulations in different financial pricing systems on the Cell processor [Aga08]. Bader *et al.* also used the Cell processor to speed up random number generation in financial risk assessment with Monte Carlo simulations [Bad08]. Docan *et al.* presented a parallelization strategy for financial risk analysis on the Cell processor [Doc09]. Instead of focusing on random number generation, the proposed strategy uses the PPE to generate a large set of Monte Carlo simulation scenarios for predicting future returns from different investment portfolios, whereas the SPEs are used to execute the option pricing algorithms concurrently based on independent simulation scenarios. However, none of these techniques attempts to parallelize the *Monte Carlo simulation* itself on the Cell platform.

While the capability of the Cell processor has been demonstrated in a broad array of applications, its potential has yet to be realized in discrete-event simulations [Per06]. In particular, formalism-based general-purpose PDES on the Cell platform remains an interesting and challenging research problem. To this end, a **Multicore Acceleration of DEVS Systems (MADS)** technique is proposed in Chapter 5 to achieve efficient parallel DEVS simulation on the Cell processor. The performance impact of the MADS technique is evaluated quantitatively in Chapter 6 and summarized in Chapter 7.

Chapter 4. The Lightweight Time Warp Protocol

This chapter proposes a novel Lightweight Time Warp protocol for efficient TW simulation of P-DEVS and Cell-DEVS models on distributed-memory multiprocessor clusters. Section 4.1 outlines the research problem and the underlying design rationales. Section 4.2 introduces the basic concepts and a set of generalized assumptions that underpin the proposed algorithms. Section 4.3 presents a rule-based event-scheduling mechanism that makes use of two types of event queues. Section 4.4 describes an aggregate checkpointing technique and an enhanced risk-free infrequent state-saving strategy. Section 4.5 covers a lightweight mechanism for efficient rollback operations, while Section 4.6 discusses several implications of the proposed algorithms. The performance impact of the Lightweight Time Warp protocol will be analyzed quantitatively in Chapter 6 and summarized in Chapter 7.

4.1. Problem Statement and Design Methodologies

This chapter aims to tackle the various challenges of DEVS-based TW simulation on distributed-memory multiprocessors by exploiting the intrinsic computational properties of the underlying simulation process presented in Section 2.6. Specifically, the research attempts to systematically address the issues discussed in Section 3.2, improving simulation performance (in terms of both execution time and memory consumption) *without* complicating the synchronization mechanism unnecessarily, sacrificing parallelism, or introducing a noticeable extra operational overhead.

The proposed algorithms, collectively referred to as the **Lightweight Time Warp (LTW)** protocol, are developed based on the following rationales.

- **Purely optimistic synchronization**.

The LTW protocol takes a purely optimistic approach to simulation synchronization, preserving the dynamics of the TW mechanism and allowing for the exploitation of an increased degree of parallelism in the simulation system. Doing so would also make the system less reliant on the knowledge of model behavior for efficient execution, a criterion that is particularly important in *general-purpose* DEVS-based simulations.

- **Simulator-centered optimization**.

As discussed in Section 2.6, PCD++ employs a flat LP structure that consists of only two coordinators (i.e., one NC and one FC) on each node, whereas many Simulators are created in a typical large-scale simulation. Hence, a substantial reduction in the operational overhead at the Simulators would result in a significant improvement in the overall simulation performance, which justifies the rationale for the adoption of a Simulator-centered optimization strategy that accelerates both forward execution and rollback recovery at the Simulators in the LTW protocol.

- **Simultaneous reduction of execution time and memory usage**.

Existing TW memory conservation techniques usually need to make a trade-off between execution time and memory usage. In contrast, the LTW protocol seeks to achieve both objectives *simultaneously* by maintaining fewer state and event (both input and output) data in the persistent queues during forward execution, while at the same time, incurring no additional overhead during rollbacks. Moreover, the LTW protocol also tries to speed up fossil collection by managing most of the persistent state data in a more concentrated manner on each node, thus allowing for more frequent memory reclamation with a relatively small impact on the overall simulation performance.

- **Facilitated event management**.

Instead of using advanced data structures and algorithms to facilitate event queue operations, the LTW protocol attempts to keep the event queues relatively short throughout a simulation, even though simulation performance can be further improved if such data structures are used. In addition, the LTW protocol is designed specifically for efficient execution of a large number of simultaneous events at each virtual time, directly addressing the computational property of large-scale, densely-interconnected, and highly-active DEVS-based models (refer to Property 3 in Section 2.6.3).

- **Bounded rollback propagation**.

The LTW protocol is intended to bound *rollback width* by preventing most of the LPs from being involved in rollback propagations, while limiting *rollback depth* by reducing the number of events that need to be unprocessed during rollbacks. As a result, the possibility of uncontrolled cascaded rollback would decrease considerably, enhancing the stability and

scalability of the simulation system. Furthermore, this objective is achieved naturally using the native TW mechanisms without additional dynamic control at runtime.

4.2. Concepts and Assumptions

The novelty of the LTW protocol lies in the division of the local simulation space on each node into two domains, referred to as the ***TW domain*** and the ***LTW domain*** respectively. The former contains *full-fledged TWLPs* that maintain past event and state data in the persistent queues following the standard TW protocol (or an optimized version). On the other hand, the latter contains *lightweight LPs* that are released from the operational overhead associated with TW execution. Figure 12 illustrates the division of simulation domains with the LTW protocol in the context of the PCD++ simulator.

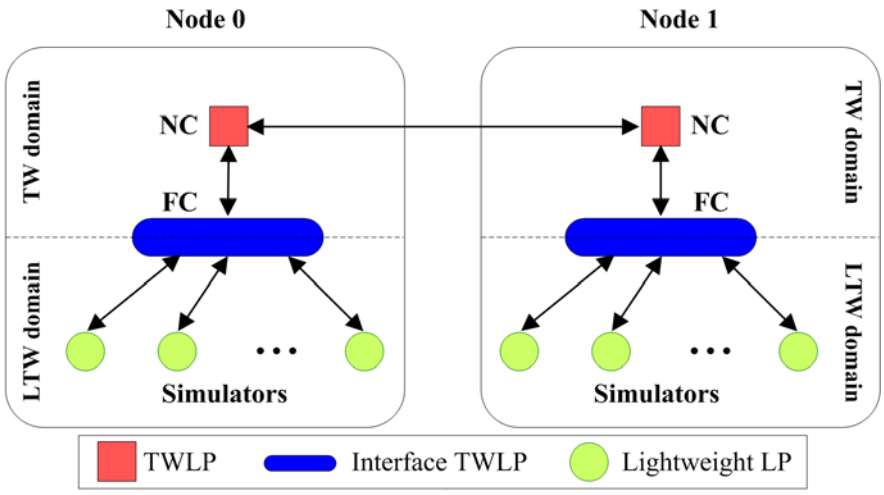


Figure 12. Division of Simulation Domains in the LTW Protocol

In PCD++, the **NC** is the only full-fledged TWLP created in the TW domain on a node, whereas *all* of the **Simulators** reside in the LTW domains. On each node, the LPs from different domains interact through a mixed-mode *interface TWLP*, which acts as a gatekeeper for the local lightweight LPs. This interface TWLP is realized by the **FC** in PCD++.

There are two points that need to be clarified here. First, all of the LPs in the system still execute *optimistically* with potentially different LVT values, just like in the standard TW protocol. However, as will be explained later, the lightweight LPs no longer rely on the persistent event and state queues to recover from causality errors under the LTW protocol. Secondly, unlike many other LP aggregation techniques that require the use of *additional concrete* data structures and simulation entities for the purpose of grouping LPs into different

clusters (e.g., the input and output gateway processes created in the Local Time Warp protocol [Raj93 and Raj07], and the cluster environment and cluster input/output queues used in the Clustered Time Warp protocol [Avr95 and Avr01]), the LTW protocol divides a local simulation space into two domains only *conceptually* based on the functional differentiation of the LPs. Such a conceptual division of local simulation space, however, is useful for better understanding the LTW algorithms presented in the following sections.

The LTW protocol makes several generalized assumptions regarding the control of LPs in a TW simulation. Note that these assumptions not only specify the conditions under which the LTW protocol can be applied to TW simulation of P-DEVS and Cell-DEVS models, but also provide a general guideline for utilizing the LTW protocol to improve performance in a wide range of TW-based PDES systems. The following is a summary of these LTW assumptions.

Assumption 1. Prior knowledge about timing of state changes at lightweight LPs.

The interface TWLP is assumed to possess the full knowledge about the timing of state changes at the local lightweight LPs. This timing information does not need to be known prior to the start of a simulation. Instead, it is obtained by the interface TWLP at runtime, just before the actual occurrence of state changes at the local lightweight LPs. In PCD++, this assumption is guaranteed by the fact that, on each node, all events to be executed by the Simulators come from the parent FC (refer to *Property 2 – Communication property* in Section 2.6.3). Since state changes can happen only as a result of processing input events in a discrete-event simulation, the FC knows the exact timing of state changes at the local Simulators whenever it schedules events for them.

Assumption 2. Advancing simulation time from TW domains.

The full-fledged TWLPs in the TW domain is supposed to advance the simulation time on the host node, usually by sending time-changing events (whose receive time stamps are greater than their send time stamps) to the other local LPs. On the other hand, the interface TWLP and the lightweight LPs do not advance their LVTs voluntarily, nor do they send messages across virtual time boundaries (i.e., they only send messages with the same send and receive time stamps that are both equal to the current simulation time). In PCD++, this assumption is a direct consequence of the event-processing algorithms since the NC is the

only TWLP that is in charge of simulation time advancement on each node (refer to *Property 1 – Time advance property* in Section 2.6.3).

Assumption 3. Performing inter-node communication from TW domains.

The full-fledged TWLPs in the TW domains are assumed to be responsible for inter-node messaging, whereas the interface TWLP and all of the lightweight LPs on each node send event messages to local receivers only. This does not prevent the interface TWLP or lightweight LPs from delivering events to remote destinations. It just requires that any message to be sent to other nodes must be forwarded to the local full-fledged TWLPs so that the message can be routed to remote destinations. In PCD++, this assumption is a direct match to the computational properties of the DEVS-based simulation process (refer to *Property 2 – Communication property* in Section 2.6.3).

Assumption 4. Prior knowledge about timing of rollbacks at lightweight LPs.

The interface TWLP is assumed to know the timing of rollbacks that will happen at the local lightweight LPs. This assumption can be inferred from other two assumptions. Since any speculative computation (in terms of virtual time advancement) is initiated in the TW domain (Assumption 2), and any straggler or anti-messages from remote nodes are received by full-fledged TWLPs first (Assumption 3), potential rollbacks on each node will always propagate from the TW domain to the LTW domain through the interface TWLP. As a result, the interface TWLP has enough information to figure out when rollbacks will occur at the local lightweight LPs. In PCD++, the FC thus knows when rollbacks will happen at the child Simulators at runtime.

These assumptions might seem a bit restrictive, but in practice many TW-based PDES systems can be converted, at least partially, into this computing model by imposing an appropriate control over the TWLPs.

4.3. Rule-Based Dual-Queue Event Management

As discussed in Section 3.2, keeping historical input and output events in the persistent event queues is one of the major sources of operational overhead in TW simulations, consuming memory space and increasing the cost of event queue operations. To address this problem, the LTW protocol classifies the input events into two types, referred to as *persistent events*

and *volatile events*. While persistent events are preserved in the input queues after being processed (i.e., they can be reclaimed only during fossil collection), volatile events are discarded right after execution, just like in a sequential or conservative simulation. An event-scheduling algorithm is then proposed to schedule both types of input events using a set of prioritized rules. The following subsections present the LTW event management scheme.

4.3.1. Introducing a Volatile Input Queue

On each node, an additional *volatile input queue* is introduced to manage the volatile events sent between the local LPs. Specifically, it is used to hold temporarily the *simultaneous* events exchanged between the FC and its child Simulators *within* a simulation phase (or round) at any given virtual time. It is safe to reclaim these events immediately after execution because, as simultaneous events with the same send time stamp, they are either committed together in the absence of causality errors or annihilated with each other during rollbacks under the TW protocol. Hence, defining these events as volatile captures the overall behavior (or net effect) of the TW simulation. On the other hand, the original *persistent input queue* is used only by the NC and the FC to store the events sent between them.

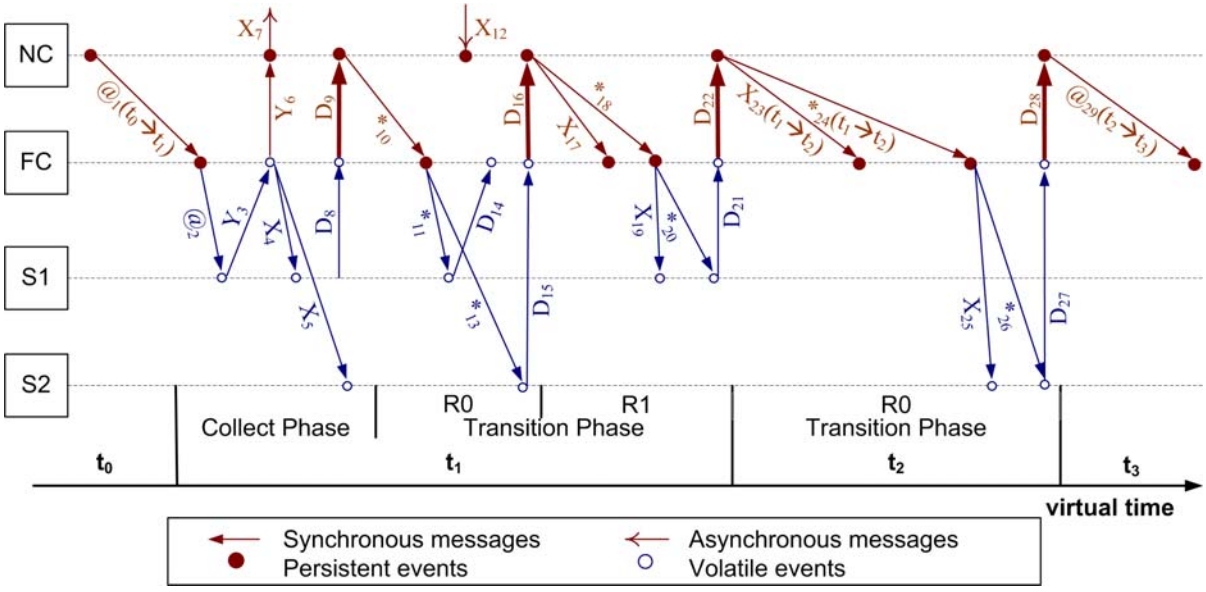


Figure 13. A Message-Passing Scenario with LTW Event Classification

Figure 13 illustrates this dual-queue event management scheme, where a NC, a FC, and two Simulators (denoted as S1 and S2) are created in the local simulation on a node. As shown in the diagram, events scheduled for the NC are still inserted into the persistent input

queue. However, events received by the FC are put into the persistent input queue only if they come from the parent NC. In addition, all events exchanged between the FC and the child Simulators are inserted into the volatile input queue.

Figure 14 depicts the message-passing scenario from the TW perspective, in which all of the Simulators appear to be *nonexistent* in the simulation. As a consequence, the TW simulation involves only *a small fraction of* the total events executed by the LPs, whereas most of the input events are no longer under the control of the TW mechanism (refer to *Property 4 – Pending event property* in Section 2.6.3).

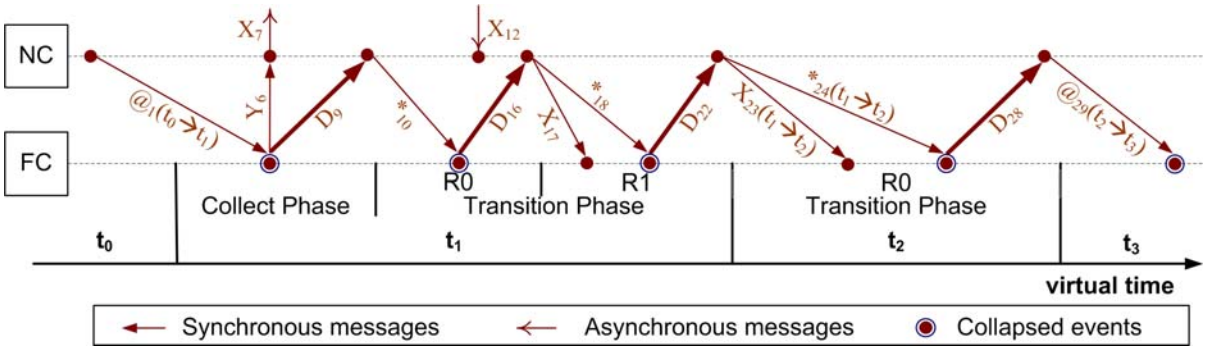


Figure 14. A Message-Passing Scenario from the TW Perspective

Comparing Figure 13 with Figure 14, the simultaneous events executed by the FC and the Simulators within a simulation phase (or round) can be viewed as being collapsed into a single aggregate event. This aggregate view is just a *conceptual* aid for demonstrating the effect of introducing the volatile input queue. It does not change the actual event granularity in the simulation. Note that the time-changing events sent from the NC to the FC (e.g., X_{23} and $*_{24}$) are still kept in the persistent input queue, allowing the simulation to resume forward execution after rollbacks. For instance, events X_{25}, $*_{26}$, D_{27}, D_{28}, and $@_{29}$ will be cancelled should a straggler or anti-message with a time stamp of t_2 arrive, and the simulation resumes from the unprocessed X_{23} and $*_{24}$ thereafter, essentially reinitiating the simulation phase(s) at virtual time t_2 with the received straggler or anti-message being taken into account.

Introducing the volatile input queue comes with several appealing features, as follows.

- **Reduced memory consumption with minor extra overhead**.

Events in the volatile input queue are discarded safely and their memory reclaimed immediately after execution. Since most of the input events are turned into volatile in a simulation, the system memory footprint would be reduced considerably. Furthermore, doing

so will not incur a significant extra operational overhead. As will be discussed shortly, the only extra operation required by this new event management scheme is to compare the time stamps of the next persistent and volatile events available in the queues for proper event scheduling during forward execution, at a minor computational cost.

- **Facilitated event queue operation using simple data structures**.

Events in the volatile input queue always have the same time stamp. At any virtual time, simultaneous events are added to the volatile input queue as the simulation enters into each phase (or round), and deleted as the execution proceeds. By the end of a simulation phase (or round), i.e., when the FC sends a (D, t) event back to the NC, the volatile input queue becomes empty, allowing the queue to be kept relatively short throughout the simulation. Moreover, it is sufficient to implement the volatile input queue using a simple **FIFO (First In, First Out)** data structure, allowing for efficient event queue operation in constant time. Consequently, the persistent input queue also becomes much shorter than it would be in the standard TW protocol, with accelerated event queue operation as well. Note that advanced data structures (e.g., Calendar Queues [Bro88, Oh97, Oh99, and Tan00]) can be used to further speed up operations in the persistent input queue, but their operational efficiency would be improved with this new event management scheme thanks to the reduced problem size (i.e., number of persistent events).

- **Accelerated forward execution with reduced rollback cost**.

For those volatile input events, their counterpart anti-messages need not to be saved in the output queues of the sending LPs, essentially eliminating all of the output queues previously created for the Simulators, and shortening the output queue associated with the FC to a great extent. This not only further decreases memory usage for saving output events and accelerates simulation during forward execution, but also reduces rollback overhead since message annihilation is no longer required to cancel incorrect volatile input events (as they have already been deleted during forward execution), which in turn enhances overall simulation performance and system stability.

- **Reduced overhead for fossil collection**.

Using the volatile input queue also reduces the overhead of fossil collection due to a significant reduction in the total number of input events and anti-messages saved in the

persistent input and output queues. Therefore, more frequent GVT estimation and fossil collection could be performed in a simulation without incurring an overwhelming operational cost, releasing more memory and resulting in even shorter persistent queues.

4.3.2. A Rule-Based Event-Scheduling Algorithm

With the proposed event management scheme, input events are managed collectively by both the volatile input queue and the persistent input queue on each node. With two input queues at hand, an event-scheduling algorithm is required to ensure a **Least-Time-Stamp-First (LTSF)** event execution, while at the same time enhancing execution efficiency and lowering the possibility of potential performance degradation. The following discussion assumes that an event scheduler is located on each node to determine the next event to be executed in each simulation cycle. Figure 15 illustrates the dual-queue event-scheduling mechanism in the LTW protocol.

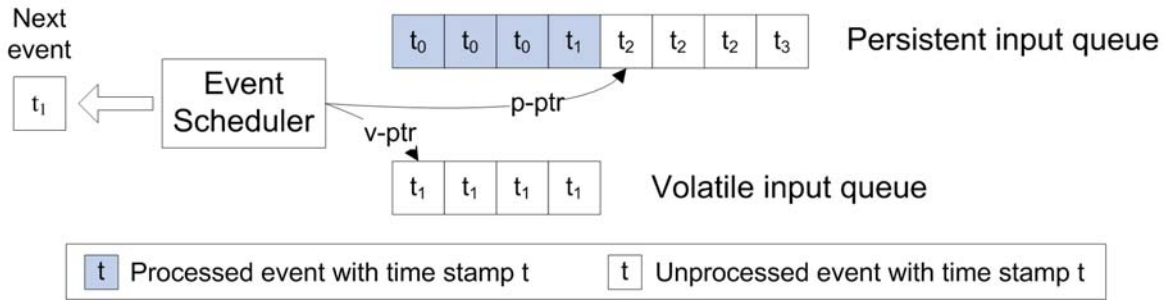


Figure 15. Dual-Queue Event Scheduling

The persistent input queue contains events sorted in LTSF order, including unprocessed events and those that have already been processed but not yet been fossil collected. On the other hand, the volatile input queue contains only simultaneous events that have not yet been processed in the current simulation phase (or round) at a specific virtual time. The event scheduler maintains two pointers, referred to as `p-ptr` and `v-ptr`, to reference the next available events in the corresponding input queues respectively. While `p-ptr` may need to be updated whenever the persistent input queue is modified (e.g., during event insertion and/or cancellation) in order to ensure that the pointer always refers to the first unprocessed persistent event with the smallest time stamp, `v-ptr` simply points to the first volatile event at the head of the volatile input queue. At each event selection point, the event scheduler compares the two events based on a set of event-scheduling rules and chooses one of them to

execute in the current simulation cycle. In effect, the event-scheduling rules allow the scheduler to adjust the priorities of the input queues dynamically on an event-by-event basis.

During a simulation phase, (X, t) events may arrive from remote NCs (e.g., X_{12} in Figure 13), and will be flushed to the FC and the destination Simulators in the next round of computation (e.g., X_{17} and X_{19} in Figure 13). To avoid unnecessary rounds in a transition phase, the NC needs to execute these remote (X, t) events immediately upon arrival so that more (X, t) events can be cached in the inter-node message buffer and sent to the FC in a batch (refer to Figure 8 in Section 2.6.1). Likewise, the Simulators may send messages to remote destinations during a collect phase (e.g., Y_3, Y_6 and X_7 in Figure 13). As these messages are potentially stragglers at the receiving ends, a delay in their delivery could postpone rollbacks at the destinations, leading to degraded performance. Note that, according to LTW Assumption 3, all these kinds of inter-node messaging are mediated by the persistent input queue in the TW domain. Therefore, the event-scheduling rules should grant a higher priority to persistent events than volatile events when they have the same time stamp. This principle is reflected in the event-scheduling algorithm given in Figure 16.

```
1.  when the scheduler is consulted to determine the next event
2.      if v-ptr = NULL (volatile input queue is empty), then
3.          if the time stamp of p-ptr > simulation stop time then
4.              return NULL
5.          else
6.              next_event = p-ptr
7.              p-ptr = the next unprocessed event in the persistent queue
8.              return next_event
9.          end if
10.     else
11.         if the time stamp of p-ptr <= the time stamp of v-ptr then
12.             next_event = p-ptr
13.             p-ptr = the next unprocessed event in the persistent queue
14.             return next_event
15.         else
16.             next_event = v-ptr
17.             remove the first event from the head of the volatile queue
18.             v-ptr = the new head of the volatile queue
19.             return next_event
20.         end if
21.     end if
22. end when
```

Figure 16. LTW Event-Scheduling Algorithm

This event-scheduling algorithm can be summarized succinctly in the following rules, which are listed in priority order (from highest to lowest).

Rule 1. Idle condition.

The next event is set to null if the volatile input queue is empty and the next available persistent event has a time stamp that is beyond the simulation stop time (line 4). In this case, the local simulation becomes idle on the host node, and the NC may be reactivated later upon the arrival of (X, t) events from the other nodes.

Rule 2. Simulation progress.

Whenever the volatile input queue becomes empty at the end of a simulation phase (or round), the event pointed by `p-ptr` is selected for execution if its time stamp is less than or equal to the simulation stop time (line 6). This rule ensures that the NC can (1) advance simulation time on the host node during the processing of the persistent (D, t) event returned from the FC, or (2) resume forward execution from the unprocessed persistent events in the wake of a rollback, or (3) reactivate the local simulation from the idle state as soon as (X, t) events from the other nodes become available in the persistent input queue.

Rule 3. Aggressive inter-node communication.

If the volatile input queue is not empty (i.e., in the middle of a simulation phase or round), a persistent event with a time stamp that is smaller than or identical to that of the volatile events will be chosen for execution instead (line 12). This rule guarantees that inter-node messages will be processed immediately by the NC once they are inserted into the persistent input queue.

Rule 4. LTSF event execution.

In all other cases, the scheduler chooses the next volatile event to execute in the current simulation cycle (line 16), enforcing a LTSF event execution on the host node.

It is worthwhile to point out that an event selected from the volatile input queue is removed from the queue by the scheduler (line 17), and this volatile event will be deleted by the receiving LP after execution. In contrast, an event chosen from the persistent input queue is simply marked as processed, and the `p-ptr` is moved to the next available persistent event afterwards (line 7 and line 13).

4.4. Aggregate State Management

Under the TW protocol, each TWLP needs to maintain its own persistent state queue in order to undo erroneous modifications to state variables during rollbacks. Although this approach allows for wide generality and straightforward implementation, it suffers from several drawbacks from a performance point of view. First, the persistent state data are scattered among individual TWLPs, making it difficult to perform efficient batch operations across different state queues. For example, all the state queues must be queried one by one during a fossil collection, a costly operation that could otherwise be performed more efficiently in a more concentrated fashion. Secondly, state restorations at the TWLPs are triggered solely by straggler or anti-messages, requiring an excessive message exchange between the TWLPs and putting a heavy burden on the underlying communication infrastructure. To address these performance issues, the LTW protocol introduces an *aggregate checkpointing scheme* that allows the lightweight LPs to delegate the responsibility of state management to the interface TWLP. In addition, an enhanced *risk-free infrequent state-saving mechanism* is proposed to reduce state-saving overhead in DEVS-based TW simulations. The following subsections present the LTW state management scheme.

4.4.1. Introducing an aggregate state manager

At the heart of the LTW state management scheme is an *aggregate state manager* created for the FC on each node. This state manager not only manages the state queue for the FC itself, but also those used by the child Simulators. Conceptually, one can consider that each Simulator still has its own persistent state queue, but under the control of the aggregate state manager at the FC. In addition, a Boolean flag, referred to as *dirty bit*, is associated with the state queue of each Simulator. Figure 17 shows the structure of the aggregate state manager.

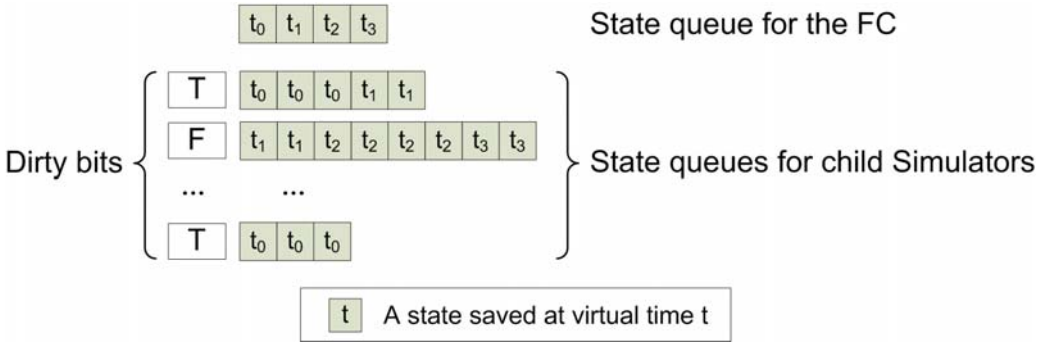


Figure 17. Structure of FC Aggregate State Manager

According to LTW Assumption 1, as discussed in Section 4.2, the FC knows the exact timing of when state changes will occur at the child Simulators. Nevertheless, the state of a Simulator should be saved *after* the processing of an event received from the FC because the state variables defined in the Simulator are subject to modification during the event execution. Moreover, usually *not all* of the Simulators are involved in the computation of a simulation phase (or round) at any given virtual time. Some of them may stay idle for an indefinite period of time. This is why the aggregate state manager needs to define a set of dirty bits for the child Simulators. After scheduling an input event for a child Simulator, the FC instructs the aggregate state manager to set the corresponding dirty bit for that Simulator. The actual checkpointing operation will be carried out only when the FC determines that the input events previously scheduled for the child Simulators have already been processed, as will be explained shortly. Further, the aggregate state manager will save the states only for those Simulators with dirty bits set to true. In short, the dirty bits are used to identify those *active* child Simulators that have processed at least one input event at the current virtual time. No dirty bit is associated with the state queue of the FC itself since the FC is always involved in the computation at each virtual time.

4.4.2. An Enhanced Risk-Free Infrequent State-Saving Strategy

In [Liu07], a **Message Type-based State-Saving (MTSS)** strategy has been proposed to reduce state-saving overhead in DEVS-based TW simulations. With this strategy, a TWLP needs to save its state only for a specific type of input events, which are processed at the *end* of each round of state transitions performed at a virtual time. As the saved states reflect the updates of state variables in the state transitions, they are *sufficient* for state restoration during potential rollbacks. In particular, the NC and the FC need to save states for (D, t) events, and the Simulators need to save states for (*, t) events. The checkpointing operations can be safely skipped for all the other types of events, reducing memory usage and state-saving overhead considerably (an improvement of up to 30% in memory consumption and state-saving time has been observed in the experiments) [Liu07]. In addition, the MTSS strategy is regarded as *risk-free* in the sense that, unlike other infrequent state-saving techniques, no coast-forward operation is required during rollbacks, even though fewer states are saved in the state queues.

The efficiency of the MTSS strategy, nonetheless, needs to be further improved since a TWLP may still save more states than necessary to recover from causality errors without resorting to coast-forward operation during rollbacks. As a transition phase may include multiple rounds of computation at any virtual time, a TWLP can save *multiple* states all at the same virtual time, even though only the *last* one will be used for the purpose of state restoration during rollbacks, wasting a certain amount of memory for saving past states.

To illustrate this point, let us revisit the message-passing scenario shown in Figure 9 of Section 2.6.2. Based on the MTSS strategy, Simulator S1 will save its state for both $*_{10}$ and $*_{26}$ at virtual time t_2. However, saving a state for $*_{10}$ is actually unnecessary because it is the state saved for $*_{26}$ (which is processed at the end of the *last* round of state transition at t_2) that will be used for state restoration should S1 be rolled back to t_2 later in the simulation. In order to avoid such unnecessary state saving in a simulation, the MTSS strategy is enhanced so that only a single state is saved for an active TWLP at the end of each distinct virtual time, as presented next. It also represents a *simulation phase oriented* optimization based on the concepts introduced in Section 2.6.2.

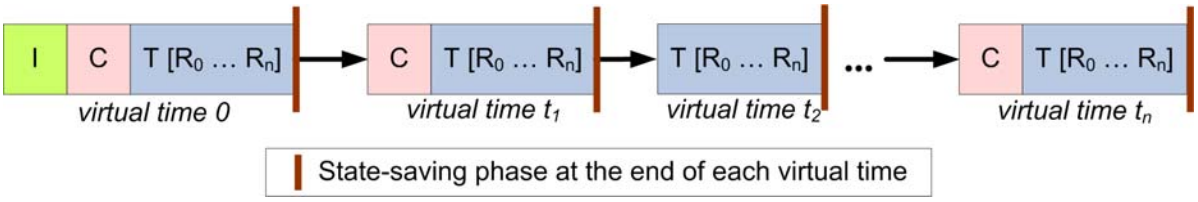


Figure 18. Introducing a State-Saving Phase for Each Virtual Time

Shown in Figure 18, a *state-saving phase* is added to the end of each virtual time, where the NC determines the next virtual time during the execution of a (D, t) event returned from the FC (see line 3.7 in Figure 8 of Section 2.6.1). If the local simulation time is about to be advanced to a new value, the NC first instructs the FC to save states for the *current* virtual time (i.e., between line 3.9 and 3.10 in Figure 8). Since all the input events scheduled for the FC and the Simulators on the host node have already been processed at this moment (refer to *Property 4 – Pending event property* in Section 2.6.3), the saved states include the latest updates of the state variables defined in the TWLPs. Only when the state-saving phase completes, can the NC send time-changing events to the FC to initiate the next simulation phase at the new virtual time. The state of the NC itself is still saved after processing a (D, t) event from the FC, just like in the original MTSS strategy. Hence, the actual checkpointing

operations are performed only in the state-saving phases throughout a simulation. Unlike the other types of simulation phases, which are marked by real phase-changing events sent between the NC and the FC, the state-saving phases do not involve any event execution. Instead, it is an abstraction to identify the state-saving points in the simulation process.

```
1.1.  when the FC sends a (*, t) to a child Simulator S (in a non-state-saving phase or round)
1.2.        if S.dirty_bit = FALSE, then
1.3.                S.dirty_bit = TRUE
1.4.        end if
1.5.  end when

2.1. when the NC requests a state-saving operation (in a state-saving phase)
2.1.        save a copy of the FC's current state in the FC's state queue
2.3.        for each child Simulator S do
2.4.                if S.dirty_bit = TRUE, then
2.5.                        save a copy of the Simulator's current state in its state queue
2.6.                        S.dirty_bit = FALSE
2.7.                end if
2.8.        end for each
2.9. end when
```

Figure 19. LTW State-Saving Algorithm

Figure 19 gives the LTW state-saving algorithm using the aggregate state manager introduced earlier. The algorithm consists of two parts. When the simulation executes in a simulation phase (or round) other than the state-saving phases, the aggregate state manager performs simple bookkeeping operations by setting the dirty bit to true whenever the FC sends a (*, t) event to a child Simulator (line 1.1 to 1.5). Note that the dirty bit does not need to be set repeatedly when the FC sends other types of events to the same Simulator, as it suffices to identify an *active* child Simulator when the actual state transition is performed during the execution of a (*, t) event.

On the other hand, when the simulation enters into a state-saving phase (just before the advancement of the local simulation time), the aggregate state manager first saves the state of the FC (line 2.1), and then it saves the states for those child Simulators whose dirty bits have been set to true (line 2.5). After saving the states, the dirty bits of the Simulators are reset back to false (line 2.6), making them ready to be used in the next virtual time.

When compared to the MTSS strategy, the LTW state-saving algorithm has two main advantages. First, the aggregate state manager only needs to set a Boolean flag for all but the last (*, t) event sent to a Simulator at a virtual time, which can be performed much quicker

than actually saving the states in the persistent state queue, resulting in better simulation performance. Secondly, only one state is saved for each active Simulator at any virtual time, regardless of how many rounds of state transitions may occur at that virtual time, reducing memory usage and the length of the state queue with accelerated queue operations.

In addition, the algorithm can be integrated with other optimizations to further reduce state-saving overhead. For instance, incremental state-saving techniques can be used in applications where only a small portion of the state data is modified during event execution.

4.5. Lightweight Rollback Mechanism

Under the TW protocol, causality errors are detected and recovered through rollbacks, which are triggered by the arrival of straggler and/or anti-messages at the TWLPs. With the LTW event and state management schemes, however, the lightweight LPs no longer maintain historical input/output events and states, making them unable to rely on the TW rollback mechanism. Therefore, a new *lightweight rollback mechanism* is required to allow the lightweight LPs to recover from potential causality errors without using anti-messages, as presented in the following subsections.

4.5.1. Full-Fledged, Interface, and Lightweight LPs

As mentioned in Section 4.2, the LTW protocol classifies the LPs on a node into three types, referred to as *full-fledged TWLP*, *interface TWLP*, and *lightweight LP*, which correspond to the NC, FC, and Simulators respectively in PCD++. Although they all execute optimistically, the TW mechanisms are realized quite differently for each type of LPs, as summarized below.

- **NC – full-fledged TWLP**.

The NC is the only full-fledged TWLP residing in the TW domain on a node. It executes as usual under the standard TW protocol (or an optimized version) using the persistent input, output, and state queues. Hence, the NC can be precluded from the discussion of the LTW rollback algorithm.

- **FC – interface TWLP**.

The FC serves as an interaction mediator between the TW and LTW domains on each node. It becomes a mixed-mode TWLP that makes use of both persistent and volatile input

queues, a persistent output queue, and an aggregate state manager. Since the output events sent to the child Simulators are no longer kept in its output queue, the FC cannot use anti-messages to trigger rollbacks at the Simulators. In addition, the FC is also responsible for restoring the states of the Simulators, if necessary, during rollbacks.

- **Simulator – lightweight LP**.

The Simulators are turned into lightweight LPs whose input queues become volatile, whose output queues are removed altogether, and whose state queues are delegated to the FC. As a result, they are neither expected nor allowed to carry out rollbacks on their own in the LTW protocol. Rather, the Simulators simply execute whatever input events received from the FC based on their *current* states, pretty much like in a sequential simulation, yet preserving the dynamics of the TW mechanism while reducing the operational overhead. Thus, the Simulators can be precluded from the LTW rollback algorithm as well.

4.5.2. A Lightweight Rollback Algorithm

Based on LTW Assumption 4, rollbacks always propagate from the FC to the Simulators on a node. That is, the FC is presented with an opportunity to *intercept* rollback propagation and to perform any necessary rollback operations on behalf of the child Simulators. Moreover, the incorrect input events previously executed by the Simulators have already been deleted in the volatile input queue during forward execution. This is one of the most prominent features of the LTW protocol, as it allows for saving the execution time that would otherwise be wasted on matching messages to their counterpart anti-messages in the persistent input queues, especially when a large number of events need to be annihilated during rollbacks. The rollback of the FC itself is still triggered by straggler and/or anti-messages received from the NC. Hence, the main purpose of the LTW rollback algorithm is to properly restore the Simulators' states, which are managed by the aggregate state manager at the FC.

One difficulty, however, is that the Simulators execute asynchronously and thus may have different LVTs. During a rollback, only the states of those Simulators that have involved in speculative computation need to be recovered. For example, if the rollback time is 100, the state of a Simulator that has stayed idle since virtual time 80 should not be restored at all. To solve this problem, the FC uses an array of **Latest state Change Time (LCT)** values to keep track of the *latest* virtual times when state transitions are performed at

the child Simulators in response to (*, t) events. The LCT value is updated whenever the FC sends a (*, t) event to a Simulator. Therefore, the FC can determine whether a Simulator has involved in speculative computation by comparing the rollback time with the LCT of the Simulator.

The LTW rollback algorithm consists of two components, one for the FC and the other for the event scheduler introduced in Section 4.3.2. Figure 20 gives the algorithm for the FC.

```
1.1. when the simulation is started
1.2.        create a LCT array of size N at the FC (where N is the number of child Simulators)
1.3.        for i = 1 to N do
1.4.              LCT(i) = ZERO
1.5.        end for
1.6. end when

2.1. when the FC sends a (*, t) to a child Simulator i
2.2.        if LCT(i) != t, then
2.3.              LCT(i) = t
2.4.        end if
2.5. end when

3.1.  when the FC receives a straggler or anti-message with virtual time T (i.e., rollback time = T)
3.2.        rollback the FC using the standard Time Warp mechanism (or an optimized version)
3.3.        call scheduler.rollback_volatile_queue(T)
3.4.        for each child Simulator i do
3.5.              if LCT(i) >= T then
3.6.                    find the last state (S_last) saved before T in the Simulator's state queue
3.7.                    restore the Simulator's state to S_last
3.8.                    delete the states saved after S_last from the Simulator's state queue
3.9.                    LCT(i) = S_last.LVT
3.10.             end if
3.11.       end for each
3.12. end when
```

Figure 20. LTW Rollback Algorithm for FC

At the beginning of a simulation, all LCT values are initialized to virtual time 0 (line 1.4). When the FC schedules a (*, t) event for a child Simulator, the corresponding LCT value is updated to the current virtual time t (line 2.3). If a rollback occurs at virtual time T, the FC first takes any necessary actions required by the TW mechanism to roll back its own speculative interactions with the NC (line 3.2). It then invokes the event scheduler to roll back the events currently available in the volatile input queue (line 3.3). Finally, the FC instructs the aggregate state manager to restore the states of the Simulators, if necessary (line 3.4 to 3.11).

State restoration for a Simulator is performed if the Simulator's LCT is greater than or equal to the rollback time (line 3.5), which ensures only those Simulators that have changed their states at or after the rollback time are rolled back. The state restoration operations are carried out in a similar way as in the TW protocol (line 3.6 to 3.8). After the state restoration, a Simulator's LCT value is reset to the LVT recorded in the recovered state (line 3.9) so that the FC can continue to track the latest state change time at the Simulator afterwards.

```
1. when rollback_volatile_queue(T) is invoked
2.         if v-ptr != NULL (volatile input queue is not empty) then
3.                 if the time stamp of v-ptr >= T then
4.                         delete all events in the volatile input queue
5.                 end if
6.         end if
7. end when
```

Figure 21. LTW Rollback Algorithm for Event Scheduler

The rollback algorithm for the event scheduler is shown in Figure 21. The scheduler simply performs a batch operation to clear the volatile input queue if the volatile events have a time stamp that is greater than or equal to the rollback time (line 4).

Using the above rollback algorithms, causality errors can be recovered more efficiently than in the TW protocol, due to, for the most part, the reduced message annihilations in the persistent input queue. Furthermore, all of the Simulators can be rolled back without sending anti-messages, no matter how many Simulators are created in a simulation, with decreased communication overhead. As a result, rollback propagation is restricted to the TW domains only, and the maximal width of a rollback is bounded by the total number of NCs and FCs created in a simulation (i.e., total number of nodes × 2). Besides, the depth of a rollback is also decreased because a majority of events executed in a simulation has been removed from the persistent input queue, thus limiting the number of events that need to be unprocessed in a rollback.

4.6. Implications of the LTW Protocol

This section briefly discusses a number of implications of the proposed LTW protocol, including its impact on global control mechanisms, compatibility with existing TW optimizations, and applicability to other TW-based PDES systems. The performance of the LTW protocol will be analyzed in Chapter 6 and summarized in Chapter 7.

- **Impact on global control mechanisms.**

Though developed largely as a local control mechanism, the LTW protocol also has an impact on several key global control mechanisms. As already mentioned in the previous sections, the LTW protocol can accelerate fossil collection in a TW simulation for the following reasons. First, the persistent input queue has been shortened after the introduction of the volatile input queue, and most of the anti-messages are removed from the persistent output queues (while the number of persistent output queues is fixed at two per each node, regardless of the scale of the simulation system), lowering the overhead associated with memory reclamation and improving the efficiency of fossil collection operations. Secondly, the amount of historical state data is reduced by the enhanced risk-free infrequent state-saving strategy, further decreasing the cost of fossil collection. Furthermore, most of the state data are now managed by the aggregate state manager in a more concentrated manner on each node, allowing for efficient batch operations in the state queues.

GVT computation is another global control mechanism that can benefit from the LTW protocol. In general, a GVT computation requires different processors (nodes) to compute their own local estimations, based on which a global GVT value is obtained and broadcast to all of the nodes in the simulation system [Mat93, Kan96, Fuj97, and Che05]. Hence, the cost of a GVT computation includes the computational overhead for local GVT estimation and the communication overhead for collecting and broadcasting new GVT updates. Although the LTW protocol is not intended to reduce the cost of inter-node communication, it does mitigate the overhead for local GVT estimation since only a few TWLPs (e.g., the NC and the FC in PCD++) need to be queried on a node, and the saving would be most pronounced in large-scale simulations where many lightweight LPs coexist on each node. Moreover, it is straightforward to integrate the LTW protocol with other optimized GVT algorithms to further improve performance.

In addition, the LTW protocol makes it possible to realize agile process migration in DEVS-based TW simulations. As model partitioning is carried out at the atomic level in PCD++, it is the lightweight Simulators that will be moved around to achieve dynamic load balancing at runtime. Under the LTW protocol, the appropriate decision points for process migration would be at the end of each state-saving phase when all of the volatile events have been processed (and deleted) by the Simulators and the states have been saved by the

aggregate state manager. The cost of transferring a Simulator to another node is thus minimized at these points, because only the persistent state data need to be migrated. Besides, concentrating the Simulators' state queues at the aggragate state manager allows for the state data of *multiple* Simulators to be packed in an efficient way before sending to a destination node, facilitating the migration of closely interrelated Simulators as a group.

- **Compatibility with existing TW optimizations.**

In essence, an optimistic parallel simulation under the LTW protocol can be simply viewed as an equivalent TW simulation, but on a smaller scale in terms of the number of TWLPs mapped on each node. Hence, many existing TW optimizations can be used transparently to further improve simulation performance. For instance, different state-saving strategies [Pre94, Tay00, and Fen06], event cancellation techniques [Lin91a, Nor02, and Che09a], and event set implementations [Bro88, Ron93, and Tan05a] can be applied to the TW domains directly. As the number of TWLPs is reduced significantly in large-scale simulations, these TW optimizations are likely to be realized more effectively with the LTW protocol, relieving the concern about scalability issues of the optimization algorithms.

Likewise, various techniques for optimism control can be incorporated into the LTW protocol (e.g., [Sok91, Ste93, Tay01, Qua01b, and Sup00]). In a way, the LTW protocol can be considered as complementary to the Local Time Warp protocol [Raj07] in the sense that the former is a purely optimistic approach to reducing TW operational overhead in the local simulation space on each node, while the latter is a locally optimistic approach to mitigating cascaded rollbacks in the global simulation space across different nodes. It is not difficult to envision that both approaches would be combined consistently in a TW simulation.

- **Applicability to other TW-based PDES systems**.

The LTW protocol could also be applied in other types of TW-based PDES systems by imposing an appropriate control over the LPs, as long as the systems under consideration satisfy the set of LTW assumptions outlined in Section 4.2.

Although the discussion of the LTW protocol considers only a single pair of TW and LTW domains (i.e., a one-to-one correspondence) on each node, this in no way presents a restriction on LP organization in the system. Depending on the specific needs of a simulation system, the division of local simulation space can be adapted in many different ways. For

example, multiple LTW domains can be utilized on a node to implement domain-specific formalisms in a hybrid multi-formalism based simulation, where a shared TW domain takes the responsibility of mediating interactions between different LTW domains on the host node as well as communicating with the other remote nodes.

As multicore processors are increasingly used as building blocks in high-performance multiprocessor systems, there is a growing interest in parallelizing a single discrete-event simulation not only between multiple processors at the cluster level, but also across different processing elements mounted on each multicore chip. Towards this goal, the next chapter first proposes a computing technique for parallel DEVS simulation on the IBM Cell processor, and then discusses possible approaches to *integration* of the proposed computing technique with other cluster-based PDES techniques in order to achieve both conservative and optimistic parallel simulation on multiprocessor systems built with multicore nodes. Indeed, as will be explained later in Section 5.6, the concept of *lightweight LPs* introduced in the LTW protocol can be used to facilitate such an integration effort in the context of DEVS-based TW simulations.

Chapter 5. Multicore Acceleration of DEVS Systems

This chapter proposes a new computing technique for high-performance parallel DEVS simulation on the IBM Cell processor. Section 5.1 outlines the research problem and the underlying design rationales. Section 5.2 generalizes the workload characteristics of different types of models and identifies two major computational kernels commonly found in demanding DEVS-based simulations. The optimization and parallelization algorithms developed for these computational kernels are presented in Section 5.3 and Section 5.4 respectively. Section 5.5 gives an architectural overview of the computing technique and summarizes the multi-grained parallelization strategy used for each computational kernel. Section 5.6 discusses several implications of the proposed technique. The performance impact of the computing technique will be analyzed in Chapter 6 and reviewed in Chapter 7.

5.1. Problem Statement and Design Methodologies

In an effort to address the various challenges of PDES on heterogeneous CMP architectures, as discussed in Section 3.3, the research presented in this chapter aims to develop efficient and flexible algorithms, collectively referred to as the **Multicore Acceleration of DEVS Systems (MADS)** technique, for high-performance parallel simulation of large-scale and/or complex P-DEVS and Cell-DEVS models on the Cell processor, while hiding, to a great extent, the technical details of multicore programming from general users. Particularly, the proposed MADS technique attempts to optimize and parallelize the sequential DEVS-based simulation process, previously hosted on a single cluster node (refer to Figure 5 and Figure 10 in Chapter 2), using the Cell processor so as to combine parallel simulation at the cluster level with accelerated parallel simulation on each multicore node. In addition, this chapter also provides valuable insight and practical guidance for other application developers who intend to port existing legacy simulation software to current and future multicore CMP architectures.

The following is a summary of the design rationales that have been employed in the development of the MADS technique.

- **Formalism-based general-purpose simulation**.

The MADS technique is developed based on the DEVS M&S framework, as introduced in Section 2.1, thus allowing for *general-purpose* P-DEVS and Cell-DEVS simulations, facilitating reuse of existing models, and reducing system validation and verification cost. Besides, instead of taking advantage of a priori knowledge of a specific model, the MADS technique makes use of *generalized* workload characteristics to speed up the execution of two types of computational kernels commonly found in a wide range of different models, while providing an extensible software architecture to accommodate additional kernels as needed, thus maximizing the applicability of the proposed technique.

- **Multi-grained parallelization strategy**.

In order to exploit the full potential of the Cell processor, the MADS technique adopts a *data-flow oriented* parallelization strategy that *explicitly* explores the fine-grained data-level and event-level parallelism inherent in the DEVS-based simulation process and combines multi-grained parallelism at different system levels in a coherent way. Unlike the traditional *LP-oriented* PDES techniques, the proposed parallelization strategy *directly* addresses the computational needs of different types of demanding kernels (which correspond to the major performance bottlenecks) in a simulation, making the achievable performance gain more deterministic and predictable.

- **Performance-centric design**.

Optimizing simulation performance is regarded as a paramount issue that needs to be considered in the design of the MADS technique. In addition to the exploitation of multi-grained parallelism, varied optimization strategies are utilized to improve data locality and to further streamline the parallelized kernel computation. As will be analyzed in this and the next chapters, these optimization strategies can lead to a significant performance improvement, which, in some cases, is even more prominent than that obtained by the parallelization strategies. Moreover, the MADS technique also draws upon the lessons learned in other case studies in the literature to enhance performance. For example, the various types of simulation data are managed in a way that allows for maximizing DMA transfer performance based on the conclusions derived in [Ara09 and Pet07], which state that peak DMA performance is achievable when the addresses of the data in both memory

domains are cache-line (128-byte) aligned and when the size of transfer is an *even* multiple of 128 bytes that is 512 bytes or larger.

- **Flexible software architecture**.

Although the MADS technique is intended to accelerate certain *common* computational kernels in DEVS-based simulations using the current Cell processor, the underlying software architecture is flexible enough for straightforward future expansion. For instance, the proposed technique does not assume a fixed number of SPE cores available on the processor, not only allowing for transparent portability to future versions of the Cell processor with potentially more on-chip processing elements, but also making it possible to implement user-controlled core allocation and reservation mechanisms in order to meet specific simulation requirements. Furthermore, as no assumption is made about the inter-node communication and synchronization mechanisms, the MADS technique can be readily integrated with other cluster-based PDES techniques (both conservative and optimistic approaches) to achieve high-performance parallel simulation on hybrid super cluster systems (e.g., the Roadrunner supercomputer [Bar08]).

- **Minimal user knowledge about multicore execution environment**.

The MADS technique tries to hide the technical complexity of multicore programming from general users whenever possible. This objective is achieved, in part, by virtue of the built-in CD++ specification language, which allows for defining Cell-DEVS models using a set of descriptive *local transition functions* without the need for low-level programming [Wai02b]. This extra level of abstraction is especially valuable on the Cell processor as non-expert users can focus on their modeling issues without being distracted by the details of multicore programming. Furthermore, the MADS technique provides the necessary support, in terms of memory control and kernel orchestration services, to assist the development of P-DEVS models on the Cell processor, thus allowing a modeler to benefit from improved simulation performance with minimal knowledge about the multicore execution environment.

5.2. Workload Analysis and Computational Kernels

A computational kernel is a compute-intensive function (or a group of closely related functions) executed on a processing element in order to realize certain functionalities of an

application by producing the intended outputs for some given inputs. To leverage the computing power of the Cell processor, as discussed in Section 3.3, a software developer needs to isolate the major computational kernels in an application and port them to the SPE cores for efficient execution. In addition, the granularity of a computational kernel should be small enough to fit into the limited on-chip LS, but also large enough to perform sufficient computation in order to hide (or overlap) DMA transfer latency [Var07].

However, extracting computational kernels from a general-purpose DEVS simulator is more difficult than doing so for a special-purpose application designed to solve a specific problem. For one thing, the workload of a DEVS-based simulation depends greatly on the simulated model, requiring the *generalization* of common workload characteristics for a wide variety of different models to be executed efficiently. Furthermore, the simulator is usually organized in terms of LPs, which may not be properly aligned with the boundaries of potential computational kernels. As a result, the workload analysis needs to take into account the varied computation tasks performed by different LPs in order to construct a whole picture that accurately reflects the real performance bottlenecks in the simulation system.

To this end, the original CD++ simulation engine, which implements the event-processing algorithms presented in Section 2.6.1 based on the flat LP structure, has been ported to the general-purpose PPE core of a Cell processor, resulting in a PPE-based *sequential* simulator called **CD++/PPE**. Two representative Cell-DEVS models are executed with the sequential simulator to generate simulation profiles, illustrating several typical workload characteristics exhibited in those models commonly found in the most demanding parallel DEVS simulations.

5.2.1. Large-Scale Simulations over a Long Period of Virtual Time

One type of models that calls for the use of PDES techniques can be found in the simulation of *large-scale* systems over a *long period of virtual time*, where a large number of LPs need to be synchronized repeatedly at many distinct virtual times, leading to a significant synchronization overhead that dominates the overall simulation performance.

The stationary wildfire propagation model introduced in [Wai06] is one such example. To attain high-resolution simulation results, this model uses a large two-dimensional cell space (1024×1024) with over one million cells to simulate fire spreading scenarios over 50

virtual hours following the Rothermel method [Rot72] based on a set of *predetermined* environmental parameters (more details on the wildfire model can be found in Section 6.1.1). In order to pinpoint the performance bottlenecks in the simulation, this wildfire model is executed with CD++/PPE on an IBM BladeCenter QS22 server [IBM10b], which features 3.2 GHz IBM PowerXCell 8i processors and 32 GB main memory. Table 1 gives the resulting simulation profile obtained on a PPE core, revealing the distribution of execution time among the major system components and between the different types of events processed by the LPs.

Table 1. Wildfire Simulation Profile on PPE

Event Type	Components				
	Simulators	**FC**	**NC**	**Bootstrap**	**Other Overhead**
(I)	3.06	0.90	—	—	—
(*)	515.69	16.96	—		
(@)	8.13	55816.60	—		
(X)	11.94	0	—		
(Y)	—	94.41	—		
(D)	—	112215.00	3.25		
Sum (s)	**538.82**	**168143.87**	**3.25**	**181.57**	**134.58**
Total (s)	**169002.10**				

It is clear that the main performance bottleneck resides at the FC (shaded Sum entry), consuming more than 99% of the total execution time. Moreover, the message-wise decomposition shows that the FC spends most of the time on processing (@) and (D) events, during which the child Simulators are synchronized at each virtual time. A closer look at this ***FC synchronization task*** reveals that the exact sources of the performance bottleneck lie in the two synchronization functions: (1) function `findImminents` that is called during the processing of (@) events (line 2.3 in Figure 7); and (2) function `findMinTime` that is called during the processing of (D) events (line 6.6 in Figure 7).

Table 2. FC Synchronization Task in the Wildfire Simulation

Function Name	No. of Invocations	Accumulated Execution Time (s)
`findImminents`	535,549	55804.30
`findMinTime`	1,071,099	112189.00

As shown in Table 2, each synchronization function occupies 99.98% of the execution time spent on processing the corresponding type of events at the FC. Together, they

constitute the most dominant performance bottleneck (99.4% of the total execution time), not only because these two functions are invoked frequently in *long-running* simulations, but also because a large amount of timing data need to be processed during each function invocation in order to synchronize all of the child Simulators in *large-scale* simulations.

Table 1 also shows that a secondary bottleneck exists at the Simulators, consuming 0.32% of the total execution time. A major component of this ***Simulator event-processing task*** is the execution of (*) events, where the local transition functions are evaluated at the cells. The significance of this bottleneck depends on the *complexity* of the model behavior. It is relatively minor in this example because the wildfire model uses simplified transition rules to approximate the real system. More complex rules would be required to obtain more precise approximation, leading to higher computational cost at the Simulators.

Table 3 gives the event counts observed in the wildfire simulation, which includes over one million simulation phases (i.e., an initialization phase, 535549 collect phases, and 535549 transition phases), as indicated by the numbers of phase-changing events (shaded).

Table 3. Event Counts in Wildfire Simulation

Event Type	LPs		
	Simulators	**FC**	**NC**
(I)	1,048,576	1	—
(*)	10,285,266	535,549	—
(@)	2,076,507	535,549	—
(X)	18,666,212	0	—
(Y)	—	2,076,507	—
(D)	—	13,410,349	1,071,099
Sum	**32,076,561**	**16,557,955**	**1,071,099**
Total	**49,705,615**		

Out of the over 49 million events executed in the simulation, the phase-changing events constitute only 4.31% of the total event population, while all the others are simultaneous events executed within different simulation phases at distinct virtual times, conforming to the computational property of the DEVS-based simulation process (Property 3 in Section 2.6.3).

5.2.2. Highly-Active Simulation of Complex Model Behavior

Another type of workload is exemplified in *highly-active* systems with *complex model behavior*, where the simulation performance is determined by the *intensive computation*

required for the LPs to execute user-defined model logic in the P-DEVS state transition functions (evaluation of local transition functions in the case of Cell-DEVS models).

To illustrate this kind of workload characteristic, a watershed model, previously discussed in [Zei97] and later redefined as a Cell-DEVS model in [Wai06], is used as an example for performance bottleneck analysis. This model simulates environmental influence on the hydrological dynamics of water accumulation over 30 virtual minutes using a three-dimensional cell space (320×320×2). More details on the watershed model are given in Section 6.1.2. Table 4 shows the simulation profile of the watershed model executed with the CD++/PPE simulator on a PPE core.

Table 4. Watershed Simulation Profile on PPE

Event Type	Components				
	Simulators	**FC**	**NC**	**Bootstrap**	**Other Overhead**
(I)	0.65	0.15	—	—	—
(*)	78082.60	75.27	—		
(@)	95.09	72.16	—		
(X)	122.58	0	—		
(Y)	—	905.40	—		
(D)	—	16.42	0.002		
Sum (s)	**78300.92**	**1069.40**	**0.002**	**25.22**	**488.12**
Total (s)	**79883.66**				

As expected, the ***Simulator event-processing task*** becomes the primary bottleneck (shaded Sum entry), representing approximately 98% of the total execution time. It is evident, in the message-wise decomposition, that the Simulators need to perform compute-intensive state transitions during the processing of (*) events.

Table 5. FC Synchronization Task in the Watershed Simulation

Function Name	No. of Invocations	Accumulated Execution Time (s)
`findImminents`	331	0.74
`findMinTime`	663	1.60

Unlike the wildfire simulation, the ***FC synchronization task*** incurs only a negligible computational cost in the watershed simulation (as shown in Table 5), for two reasons. First, the watershed model uses a much smaller cell space, which is about 20% of that used in the wildfire model, thus greatly reducing the amount of timing data that need to be processed in the synchronization functions. Secondly, the watershed model consists of fewer simulation

phases over a shorter period of virtual time, effectively reducing the number of invocations of the synchronization functions.

Table 6 gives the corresponding event counts in the watershed simulation. Although the simulation consists of only 663 phases, the overall event population is 6.8 times larger than what is observed in the wildfire simulation, demonstrating a much higher level of activity with an even larger proportion of simultaneous events exchanged between the LPs.

Table 6. Event Counts in Watershed Simulation

Event Type	LPs		
	Simulators	**FC**	**NC**
(I)	204,800	1	—
(*)	33,996,800	331	—
(@)	33,527,325	331	—
(X)	167,723,421	0	—
(Y)	—	33,527,325	—
(D)	—	67,728,925	663
Sum	**235,452,346**	**101,256,913**	**663**
Total	**339,221,007**		

The *FC synchronization task* and the *Simulator event-processing task*, referred to as the **FC Synchronization Kernel (FSK)** and the **Simulator Event-processing Kernel (SEK)** respectively thereafter, represent the two *common* performance bottlenecks in many DEVS-based simulations. In the wildfire and watershed simulations, the performance is dominated overwhelmingly by just one kernel. In general, however, the relative weights of the computational kernels can vary in different models and during a single simulation. To achieve high-performance parallel DEVS simulation on the Cell processor, the following sections propose the optimization and parallelization algorithms for each of these kernels.

5.3. FC Synchronization Kernel

The FSK consists of the two synchronization functions, which are invoked regularly by the FC at specific points throughout a simulation. As introduced in Section 2.6.1, these two functions perform computation based on the Simulator timing data contained in a C++ STL map called *timesOfNextStateChange* (<*SimulatorID*, *absoluteNextStateChangeTime*>). The computation is *data-intensive* in nature when a large number of Simulators are involved in a simulation. Moreover, as the timing data may not be allocated contiguously in the main

memory (depending on the internal implementation of the STL map structure), the computation is also considered as *memory-bound*, which could result in high cache miss rates on a hardware-managed cache memory architecture (e.g., the one used by the PPE). Hence, improving data locality is an important factor in the development of the FSK algorithms.

5.3.1. Optimizing FC Synchronization Task

From the *phase-oriented* view of the simulation process, as discussed in Section 2.6.2, function `findImminents` is called at the *beginning* of each collect phase (when a (@) event is received from the NC), whereas function `findMinTime` is invoked at the *end* of each collect ***and*** transition phases (when the *last* (D) event is returned from the child Simulators). A closer examination, however, shows that it is actually unnecessary to compute the next minimum state change time among the Simulators at the end of *collect* phases since these *transitory* phases do not advance virtual time at all (i.e., each collect phase must be followed by an ensuing transition phase at the same virtual time, as required by the P-DEVS formalism). Therefore, it is safe to eliminate the redundant invocations of function `findMinTime` in all of the collect phases.

This seemly trivial and straightforward optimization was not considered in the original FC event-processing algorithms mainly because these algorithms, developed with a pure *event-oriented* mindset that focuses on the processing of individual events, overlook the intrinsic correlation between events executed at *different* points in a simulation. In other words, the FC is ***memoryless*** in terms of event execution. Consequently, when a (D) event arrives, the FC cannot determine whether this is the result of a (@) or (*) event previously sent to a Simulator. To ensure correct synchronization, the FC has to invoke function `findMinTime` for *every* last (D) event received from the Simulators, even though it suffices to do so only if the (D) event is a logical consequence of a previous (*) event.

To implement this optimization in CD++/PPE, a flag called *currentPhase* is defined in the FC to keep track of the type of the current simulation phase, essentially making the FC a ***context-aware*** LP. This flag is reset to 0, 1, or 2 when a (@), (I), or (*) event is received from the NC respectively; and function `findMinTime` is invoked only if the flag has a *nonzero* value during the processing of (D) events at the FC (line 6.6 in Figure 7). Note that this optimization can also be applied in the original sequential CD++ to enhance performance.

The performance gain is immediate in large-scale and long-running simulations. In the updated wildfire simulation profile given in Table 7, for example, the time required for processing (D) events at the FC is decreased by 50.5% as the frequency of `findMinTime` invocations is reduced roughly by half. As a result, the overall simulation performance is improved by 34% accordingly.

Table 7. Wildfire Simulation Profile on PPE (Synchronization Optimized)

Event Type	Components				
	Simulators	**FC**	**NC**	**Bootstrap**	**Other Overhead**
(I)	3.07	0.91	—	—	—
(*)	497.40	14.38	—		
(@)	7.66	55044.50	—		
(X)	11.79	0	—		
(Y)	—	93.52	—		
(D)	—	55526.50	2.32		
Sum (s)	**519.92**	**110679.81**	**2.32**	**180.65**	**121.89**
Total (s)	**111504.59**				

Depending on the relative significance of the FSK, a certain degree of performance gain can be obtained in highly-active simulations of complex model behavior as well. In the watershed simulation, however, this optimization does not lead to a noticeable improvement since the simulation performance is determined by the SEK.

5.3.2. Flattening Simulator Timing Data

Before parallelizing the FSK on the Cell processor, the Simulator timing data need to be reorganized in the main memory for several reasons. First, as mentioned earlier in Section 5.1, efficient DMA transfer requires proper address alignment and data granularity in both memory domains (i.e., main memory and on-chip LS), which cannot be guaranteed by the standard C++ STL library. Secondly, the opacity of the STL map implementation not only makes it difficult to enhance data locality in the main memory, but also poses a challenge in partitioning the timing data across different SPE cores on the Cell processor.

In order to address the above issues, a *process ID allocation scheme* is used to allocate positive IDs for the Simulators (and their associated atomic models) continuously from 0 to (N-1), in which N is the total number of Simulators created in a simulation. On the other hand, the NC and the FC (along with the coupled models) use negative IDs. Therefore, the

FC can use a simple integer array, referred to as the **Time Array (TA)**, to hold the timing data for all child Simulators, where the *array indexes* serve as the Simulator IDs. In addition, the FC also creates an integer **Imminent ID Array (IA)** to contain the IDs of *imminent* Simulators obtained in function `findImminents` during each collect phase (i.e., IA is used as an output buffer). As a result, the original STL map structure, *timesOfNextStateChange*, is replaced by a flat array-based data layout, which is illustrated in Figure 22.

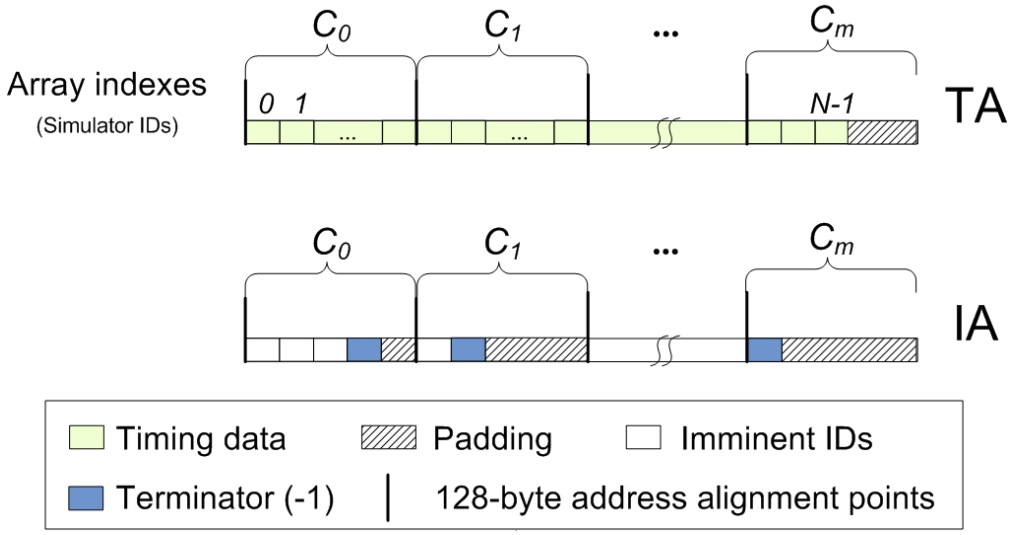


Figure 22. A Flat Data Layout for the FSK

Both TA and IA are partitioned into *m* chunks ($[C_0 \ldots C_m]$) as evenly as possible, where *m* is the number of SPEs allocated in a simulation to perform the FC synchronization task. Each chunk of data is aligned on a 128-byte boundary in the main memory for efficient DMA transfer to/from the LS of a specific SPE at runtime. To ensure proper address alignment, data padding may be applied to each IA chunk as well as to the last TA chunk, if necessary. Since only a fraction of the Simulators are imminent at any virtual time, the imminent IDs stored in an IA chunk (i.e., those obtained in `findImminents` based on the timing data in the corresponding TA chunk) is terminated by a -1 so that the FC can retrieve them efficiently without the need for a full traversal of the whole array.

As the TA indexes are used *implicitly* as Simulator IDs, this flat data layout also reduces the amount of data that need to be transferred across memory domains when executing the FSK on the SPEs. In addition, the exposure of the timing data in TA and the introduction of IA greatly facilitate the parallelization of the FSK following a ***data-flow*** oriented approach, as will be discussed in the next section. Thanks to the flat LP structure,

the Simulator timing data can be concentrated in a single array at the FC, rather than scattered around at many intermediate Coordinators in the LP hierarchy, relieving the FSK parallelization effort as well. Moreover, the simulation performance is expected to benefit from the improved data locality with reduced memory contention and cache miss penalty even when the FSK is executed on the PPE alone.

5.3.3. Processing Simulator Timing Data on SPE

The Simulator timing data are updated by the FC when the (D) events are received from the Simulators (line 6.3 in Figure 7), reflecting the next state change times scheduled individually at the child Simulators. In other words, these timing data are *mutually independent*, making it possible to dispatch different chunks of data to different SPEs for concurrent processing.

In a typical large-scale simulation that involves many Simulators, however, each TA chunk can contain a large amount of virtual time values, especially when the number of SPEs allocated for FSK execution is small. Considering the limited size of LS, it is necessary to further divide the timing data stored in a TA chunk into a sequence of blocks, each with a predetermined regular size. This block size can be adjusted at compile time, if necessary, to adapt to different model conditions. However, better DMA transfer performance can be expected when the block size is set to be an *even* multiple of 128 bytes in the range of from 512 bytes up to a maximum of 16KB. Similarly, the corresponding IA chunk is also divided into the same number of blocks with the same size as applied to the TA chunk.

On the SPE side, four data buffers are allocated in the LS: *a pair of input buffers* for reading timing data from a given TA chunk, and *a pair of output buffers* for writing imminent Simulator IDs to the corresponding IA chunk. Each LS data buffer has a size that is identical to the size of a TA block. Hence, a chosen TA block size determines the total amount of data that will be contained in an SPE's LS at any given time. For example, for a 16KB block size (which is sufficient to contain 4096 virtual time values or imminent IDs), a maximum of 64KB memory will be occupied by the data buffers in the LS. Using a larger block size not only allows for more efficient DMA transfers as more data can be transferred in one stroke, but also increases the computational granularity on the SPEs, making it more likely to overlap computation with concurrent memory I/O operations.

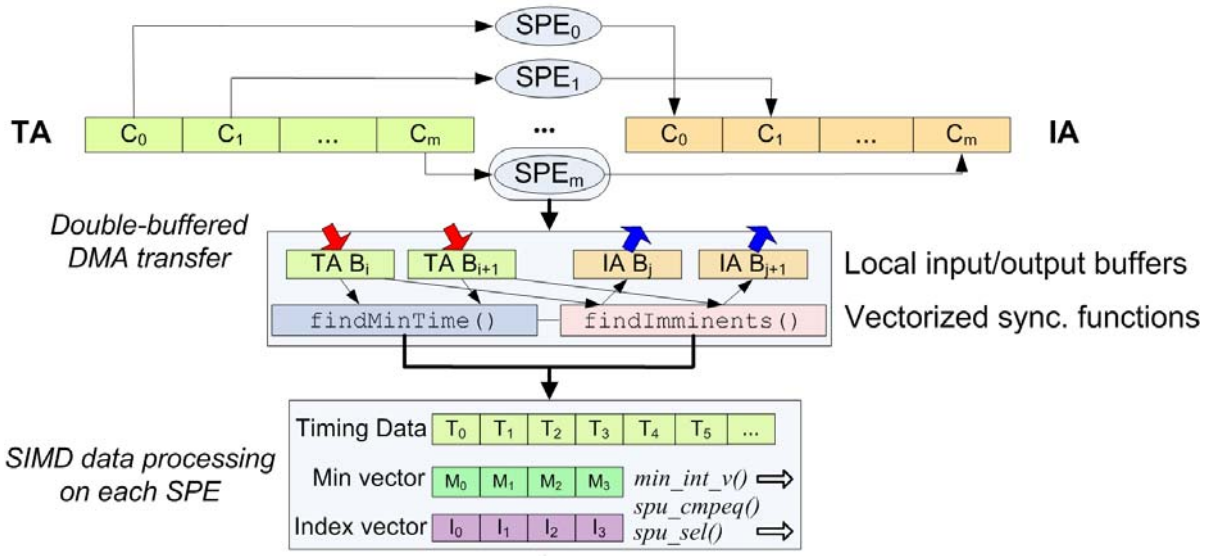


Figure 23. Parallel Data Processing on the SPEs

Figure 23 illustrates the parallelized data processing across multiple SPEs, where each SPE streams in and out the data in the corresponding TA and IA chunks using double-buffered DMA transfers. The two synchronization functions are decoupled from the FC and vectorized using SPE SIMD intrinsics to process data that have been made available in the local data buffers on each SPE.

MinVector : a 128-bit, 4-way integer vector initialized to {Inf, Inf, Inf, Inf};
TABuf0, *TABuf1* : the current and next input buffers for reading data from a TA chunk in blocks;
TAChunkSize : the size of a TA chunk;
TABlockSize : the size of a TA block;

1. **when** function **`findMinTime`** is invoked
2. Calculate *numOfDMATransfers* based on *TAChunkSize* and *TABlockSize*
3. Schedule an inbound DMA to transfer the 1st TA block into *TABuf0*
4. **for** i = 0 to (*numOfDMATransfers* − 1) **do**
5. Wait for the i^{th} inbound DMA to complete in the *current* input buffer
6. Schedule the $(i+1)^{th}$ inbound DMA to transfer the next TA block into the *next* input buffer
7. **for** j = 0 to (*TABlockSize* / 4 − 1) **do** //SIMD operations
8. Compare *MinVector* with 4 values in the current input buffer using `spu_cmpgt`
9. Update *MinVector*, if needed, to record the 4 smaller values using `min_int_v`
10. **end for**
11. **end for**
12. Calculate chunk-wise minimum = `min_vec_int4`(*MinVector*)
13. Send chunk-wise minimum to PPE through the outbound mailbox channel
14. **end when**

Figure 24. Function `findMinTime` Definition

Given in Figure 24, function `findMinTime` uses a 128-bit, 4-way integer Min Vector to scan the Simulator timing data that have been prefetched from the TA chunk into the current local input buffer (line 7 to 10). When the full chunk of data is processed, the Min Vector contains the four minimum values obtained in the 4 ways. These values are then compared horizontally to get the chunk-wise minimum (line 12), which is then sent to the PPE code via the outbound mailbox channel (line 13).

MinVector : a 128-bit, 4-way integer vector to hold the current global minimum;
IndexVector : a 128-bit, 4-way integer vector to hold 4 Simulator IDs;
TABuf0, *TABuf1* : the current and next input buffers for reading data from a TA chunk in blocks;
IABuf0, *IABuf1* : the current and next output buffers for writing data to an IA chunk in blocks;
baseId : the starting index of the TA chunk;
TAChunkSize : the size of a TA chunk;
TABlockSize : the size of a TA block;

1. **when** function **`findImminents`** is invoked
2. Read the PPE-determined global minimum from the inbound mailbox channel
3. Replicate the global minimum value in *MinVector*
4. Initialize *IndexVector* = {*baseId*, *baseId*+1, *baseId*+2, *baseId*+3}
5. Calculate *numOfDMATransfers* based on *TAChunkSize* and *TABlockSize*
6. Schedule an inbound DMA to transfer the 1st TA block into *TABuf0*
7. **for** i = 0 to (*numOfDMATransfers* − 1) **do**
8. Wait for the i^{th} inbound DMA to complete in the *current* input buffer
9. Schedule the $(i+1)^{th}$ inbound DMA to transfer the next TA block into the *next* input buffer
10. **for** j = 0 to (*TABlockSize* / 4 − 1) **do** //SIMD operations
11. Compare *MinVector* with 4 values in the current input buffer using `spu_cmpeq`
12. Select the imminent Simulator IDs from *IndexVector* using `spu_sel`
13. Write the imminent Simulator IDs continuously into the current output buffer
14. Update *IndexVector* = `spu_add`(*IndexVector*, 4)
15. **end for**
16. **if** the current output buffer is not full **then**
17. Write -1 to the current output buffer as a termination symbol
18. **end if**
19. Wait for the $(i-1)^{th}$ outbound DMA to complete in the *previous* output buffer
20. Schedule an outbound DMA to transfer the *current* output buffer to the IA chunk
21. **end for**
22. Wait for the last outbound DMA to complete
23. Notify PPE of completion through the outbound mailbox channel
24. **end when**

Figure 25. Function `findImminents` Definition

As shown in Figure 25, function `findImminents` replicates the PPE-determined *global minimum state change time* in the Min Vector (line 3). It uses another 128-bit, 4-way Index Vector to keep track of the 4 Simulator IDs corresponding to the entries in the current

input buffer, where the starting index of the TA chunk is given by the PPE code and stored in a variable called *baseId* (line 4). The imminent Simulator IDs are sifted through the Index Vector by comparing the global minimums in the Min Vector with the values in the current input buffer (line 10 to 15). At the end of the function, a status value is sent back to the PPE code through the outbound mailbox channel (line 23), indicating that the imminent IDs are made available in the IA chunk.

Though not shown in the above algorithms, by using multiple Min and Index Vectors as *simultaneous logical threads*, the compute-intensive loops (line 7 to 10 in Figure 24 and line 10 to 15 in Figure 25) can be unrolled to further speed up data processing on the SPEs. For example, using two pairs of Min and Index Vectors will allow 256-bit, 8-way SIMD operations to be performed sequentially during each iteration, with reduced looping overhead.

5.3.4. FSK Orchestration Algorithms

During simulation bootstrap, a control block is used to pass the FSK parameters (e.g., the starting addresses of the TA and IA chunks, the starting index of the TA chunk, and the TA chunk and block sizes) to each SPE thread that hosts an instance of the FSK. Throughout a simulation, the FSKs are invoked by the PPE code in a **RPC (Remote Procedure Call)** fashion. In addition to the two synchronization functions, a termination function called `terminateFSK` is defined to exit FSK execution on an SPE thread by the end of a simulation, as shown in Figure 26 (line 1.1 to 1.3).

```
void (*FSKFunctions[])(void) = {findMinTime, findImminents, terminateFSK};
stopFlag : a flag used to terminate the FSK main loop (initialized to false);

1.1. when function terminateFSK is invoked
1.2.    Set stopFlag to true
1.3. end when

2.1. when an SPE thread is started to execute the FSK
2.2.    Start an inbound DMA to fetch the control block from the main memory
2.3.    Wait until the control block becomes available in the LS
2.4.    While stopFlag is false do
2.5.       Read functionID from the inbound mailbox channel (blocking read)
2.6.       Call the corresponding FSK function using (*FSKFunctions[functionID])()
2.7.    end while
2.8. end when
```

Figure 26. FSK Main Loop and Function `terminateFSK`

Each function is associated with an integer *FSK function ID*: 0 for `findMinTime`, 1 for `findImminents`, and 2 for `terminateFSK`. These functions are managed with a function pointer array, referred to as *FSKFunctions*, so that they can be invoked directly using the received function ID as array index (line 2.6), without resorting to branching instructions on the SPEs.

```
1. when the last (D) event is received at the end of a transition phase
2.    Send 0 (functionID) to each FSK via the inbound mailbox channel
3.    Wait for all chunk-wise minimums to be returned from the outbound mailbox channels
4.    Record these chunk-wise minimums for later use in the next collect phase
5.    Merge the chunk-wise minimums to obtain the global minimum next state change time
6. end when
```

Figure 27. In-Place Invocation of Function `findMinTime` at the FC

On the PPE side, the FSK orchestration algorithm is quite straightforward. To invoke function `findMinTime`, as shown in Figure 27, the FC sends a mailbox message with a value of 0 to *all* of the SPE threads that host the FSKs when the last (D) event is received at the end of each *transition* phase (refer to line 6.6 in Figure 7). The FC then waits for the computation results to be returned from the FSKs. When all of the chunk-wise minimums are finally available, the FC merges them into a global minimum next state change time (line 5), which is then sent to the NC via a (D) event.

Two points need to be clarified here. First, function `findMinTime` is invoked ***in place*** by the FC in the sense that the FC is *blocked* until all of the chunk-wise minimums are returned from the SPE threads (line 3). Secondly, as will be explained shortly, the returned chunk-wise minimums are recorded in the main memory (line 4) in order to enhance the computational efficiency of function `findImminents` in the next round.

```
1. when the NC is about to send a (@) event to the FC (line 3.25 in Figure 8)
2.    for each FSK with a recorded chunk-wise minimum = the current global minimum do
3.       Send 1 (functionID) and the current global minimum to the FSK as 2 mailbox messages
4.    end for
5. end when
```

Figure 28. In-Advance Invocation of Function `findImminents` at the NC

Given in Figure 28, function `findImminents` is no longer called by the FC. Instead, it is invoked ***in advance*** by the NC when the simulation is about to be advanced to the next virtual time (refer to line 3.25 in Figure 8). Hence, when the FC needs to retrieve the

imminent IDs in the *collect* phase of the new virtual time, the FSKs have already executed for a while, overlapping the computation on the PPE and on the SPEs. Furthermore, an SPE thread is involved in the computation only if it actually found the current *global* minimum (line 2). As imminent Simulator IDs may not exist in some of the chunks at any virtual time, this approach would reduce the number of concurrent SPE threads required for the `findImminents` computation, with decreased memory contention for DMA transfer and enhanced FSK scalability.

```
1. when a (@) event is received at the beginning of a collect phase
2.     for each FSK that has been invoked by the NC do
3.         Wait for a status value to be returned from the FSK
4.         Fetch the imminent IDs from the corresponding IA chunk
5.     end for
6. end when
```

Figure 29. Retrieving Imminent IDs at the FC

As shown in Figure 29, for each SPE thread that has been invoked by the NC, the FC waits for a status message from the outbound mailbox channel (line 3), and then retrieves the imminent IDs from the corresponding IA chunk (line 4).

```
1. when the simulation is about to be terminated (line 3.11 in Figure 8)
2.     Send 2 (functionID) to each FSK via the inbound mailbox channel
3. end when
```

Figure 30. Terminating FSKs at the NC

The NC is responsible for terminating the SPE threads at the end of a simulation when the next simulation time goes beyond the user-specified stop time, as shown in Figure 30.

5.4. Simulator Event-Processing Kernel

The SEK includes the Simulator event-processing algorithms for (I), (@) and (*) events as well as the P-DEVS state transition functions defined in the atomic models (refer to Figure 6 in Section 2.6.1). Note that the SEK does *not* include the Simulator algorithm for processing (X) events for reasons that will be explained in Section 5.4.4. During each simulation phase, the SEK processes a set of input events scheduled by the FC based on the current states of the active Simulators (and their associated atomic models), and returns a set of output events back to the FC. The input/output events and the states are handled *independently* between individual Simulators. However, the event execution at the FC and the Simulators are

interrelated due to the event-scheduling relationship. In contrast to the FSK, which performs regular computation on a large amount of independent timing data, the SEK performs highly irregular, compute-intensive computation with a random memory access pattern, making the parallelization effort especially challenging on the SPE cores.

5.4.1. Event Level Parallelism

Unlike most LP-oriented PDES techniques, the parallelization of the SEK requires ***direct*** exploitation of the fine-grained event-level parallelism that is inherent in the DEVS-based simulation process. Furthermore, the event-level parallelism needs to be exposed in a way that allows for a ***data-flow*** oriented parallelization strategy. To illustrate such event parallelism, Figure 31 shows a step-by-step view of event processing in different types of simulation phases based on the flat LP structure.

These event-processing steps are briefly explained as follows. During the *initialization phase at virtual time 0*, the FC forwards the (I) event received from the NC to each of the **N** Simulators created in a simulation. As a result, each Simulator returns a (D) event to the FC, which in turn sends a (D) event back to the NC. A similar pattern can be seen in a *transition phase at virtual time t*, except that in this case the FC sends (*) events only to **K** *active* Simulators ($K \leqslant N$) whose associated atomic models have a state transition (δ_{int}, δ_{ext}, or δ_{con}) scheduled at the current virtual time. During a *collect phase at virtual time t*, the model outputs, encoded in (Y) events, are emitted from **M** *imminent* Simulators ($M \leqslant N$) whose associated atomic models execute their output functions (λ) in response to the (@) events received from the FC. Based on model coupling information, the FC routes these (Y) events to their destination Simulators as (X) events, which will be subsequently consumed as inputs to the receiving atomic models. Likewise, the FC also sends a (D) event back to the NC after processing all of the (D) events received from the Simulators.

Borrowing terminology from the parallel computing community, the fine-grained event-level parallelism in a DEVS-based simulation can be classified into two categories, which are referred to as *event-embarrassing parallelism* and *event-streaming parallelism* respectively. These two forms of event-level parallelism can be exploited in a way that avoids causality errors at any virtual time, as described below.

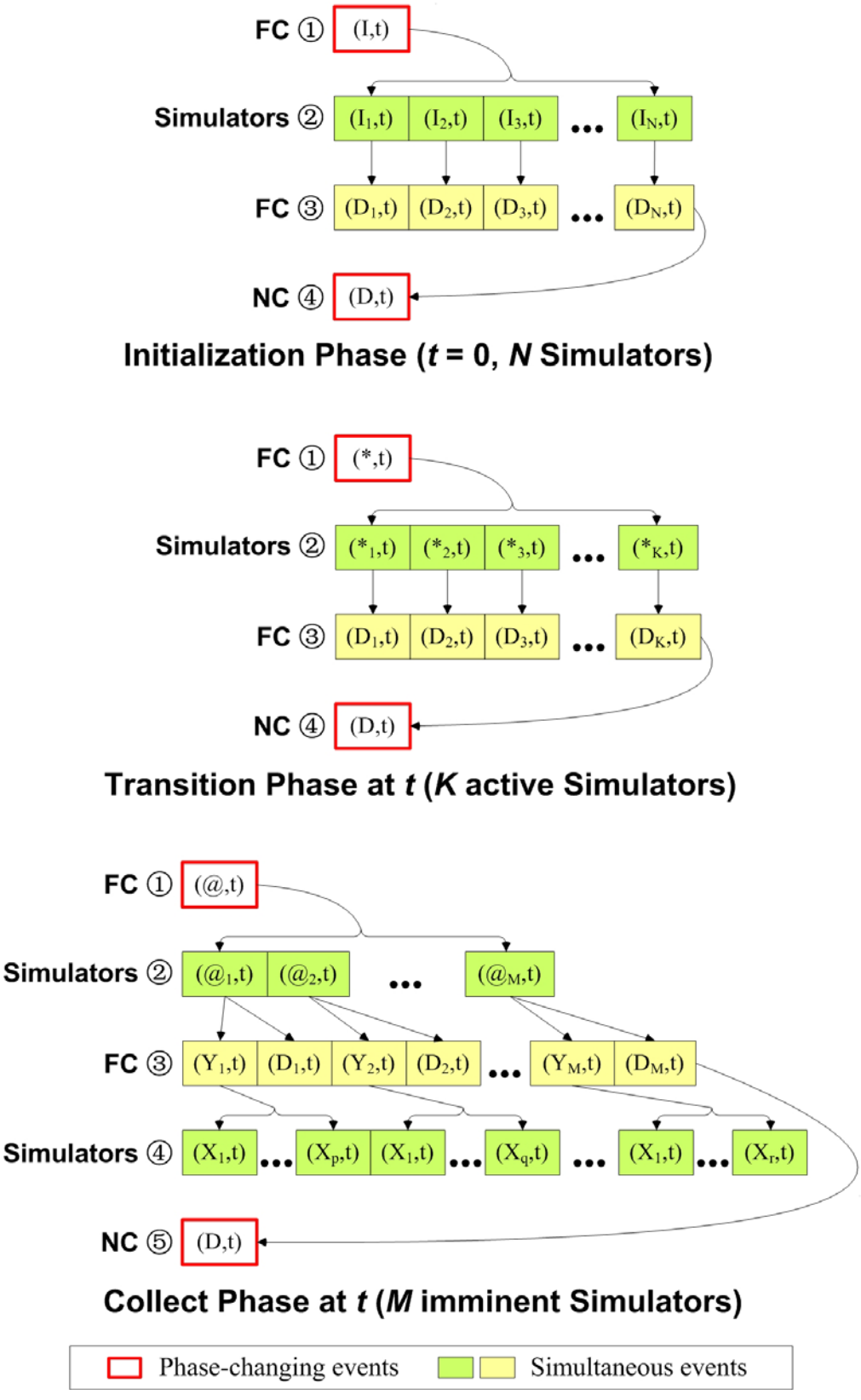


Figure 31. Event Level Parallelism in DEVS-based Simulation Process

- **Event-embarrassing parallelism**.

This type of event-level parallelism exists between the *independent* events executed *within* each step at the FC and the Simulators (shaded events in Figure 31). Since there is neither causal nor data dependency between them, these events can be executed concurrently in an arbitrary order.

- **Event-streaming parallelism**.

This type of event-level parallelism exists between the *causally-dependent* events executed in *consecutive* steps (i.e., *between* the FC and the Simulators). As the output events from the preceding step serve as the inputs to the step that follows, these events can be executed concurrently in a pipelined manner.

At the first and last steps of a simulation phase, the NC and the FC exchange *phase-changing events*, providing the natural *fork* and *join* points for simulation synchronization. In other words, the simulation can be viewed as being barrier-synchronized at the end of each simulation phase. Thanks to the flat LP structure, the number of synchronization points is minimized, exposing the maximum degree of event-level parallelism in the simulation process. Note that, according to the P-DEVS formalism, the simultaneous (X) events received by a Simulator in a collect phase (in step 4) must be consumed *as a whole* by the Simulator in the ensuing transition phase (in step 2) [Cho94].

5.4.2. LP Virtualization

To parallelize the SEK on the Cell processor, the Simulators (and their associated atomic models) need to be mapped to the SPE cores for concurrent execution. As there are usually many more Simulators than the number of available SPEs in a typical large-scale simulation, several important issues must be considered when developing a *partitioning scheme* for the Simulators.

First of all, the small size of the on-chip LS imposes a tight upper-bound on the total number of Simulators (and their associated atomic models) that can be hosted simultaneously on an SPE. Secondly, as discussed in Section 3.3.1, an SPE can execute only one thread at a time; and SPE thread context switch is both time-consuming and resource-demanding. Hence, it is impractical to swap in and out Simulators (as distinct SPE threads) between the main

memory and the LS at runtime without incurring an excessive operational overhead. Thirdly, in a typical simulation, only a fraction of the Simulators are actually active at any virtual time. Therefore, an ideal partitioning scheme should be able to map only those active Simulators to the available SPEs during execution. Finally, to enhance simulation performance, the partitioning scheme should also be flexible enough to facilitate dynamic load balancing between the SPEs.

In view of the common practice of Cell programming, where the SPEs are usually assigned with *reusable* tasks operating on a stream of data, this research addresses the above-mentioned issues through the concept of ***LP virtualization***, by which the Simulators (and their associated atomic models) are turned into *virtual LPs* that share the functionalities provided by a limited group of SPE threads, and the mapping of active Simulators to the SPE threads is determined dynamically at each virtual time throughout a simulation.

To this end, the state data originally encapsulated in the Simulator-atomic pairs of objects are separated from the event-processing and model logic. While the state data are maintained in the main memory, the Simulator event-processing algorithms and the model state transition functions are wrapped into the SEK and executed by the SPE threads. During a simulation, the state data of an active Simulator is matched to a specific SPE thread using a PPE-based SEK job-scheduling algorithm, as will be discussed in Section 5.4.6. On the other hand, the PPE is used to host the remaining *concrete LPs* such as the NC and the FC.

The following discussion assumes that *all* of the Simulators (and their associated atomic models) created in a simulation are executed on the SPEs. However, in some cases, not all of the Simulators are suitable for porting to these co-processors (i.e., some of the Simulators might still need to be implemented as concrete LPs on the PPE), a topic that will be discussed further in Section 5.6.

5.4.3. Virtual LP State Management

In Section 5.3.2, a *process ID allocation scheme* has been introduced to align the Simulator IDs so that the timing data can be managed conveniently in flat arrays. As will be discussed shortly in this and the next sections, it turns out that this process ID allocation scheme also facilitates the management of state and event data for the virtual LPs.

In the original CD++ environment, the Simulators and atomic models are created individually at random memory locations. In order to transfer the state data of virtual LPs across memory domains, the state variables previously defined in a Simulator-atomic pair of objects are repacked into a C struct, referred to as *virtualLPState*, which has an adjustable size of 512 bytes. Note that the size of *virtualLPState* needs to be big enough to contain the state data of any Simulator-atomic pair of objects, while at the same time, meeting the requirements of peak DMA transfer performance.

The state data of all of the virtual LPs are then stored in a flat 128-byte aligned array, referred to as the ***state buffer***, in the main memory, where the array indexes serve as the IDs of the virtual Simulators. On each SPE, a pair of 128-byte aligned ***local state caches*** is allocated in the LS to hold the state data of *at most two* active Simulators at any time. Figure 32 illustrates the resulting layout of state data in both memory domains.

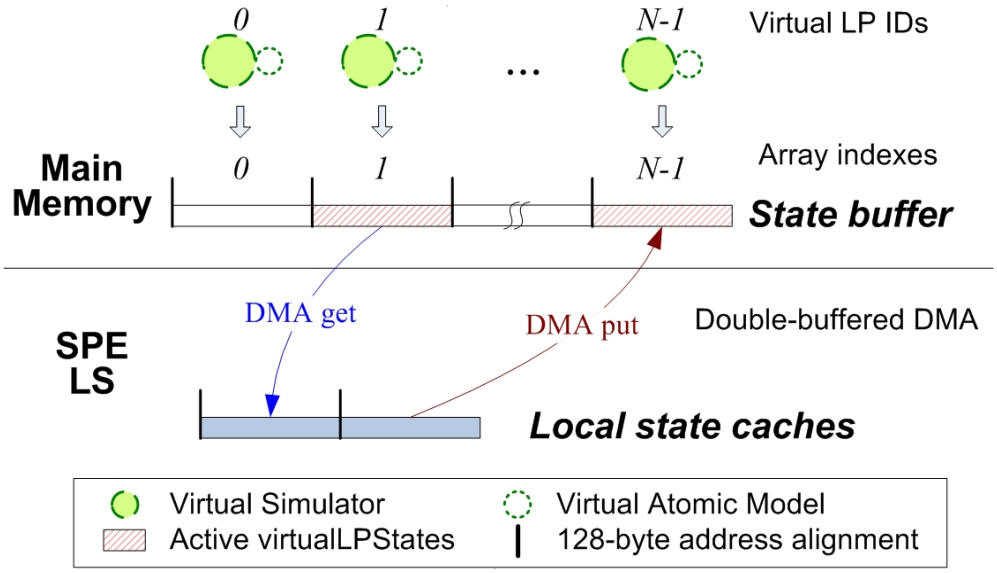


Figure 32. Virtual Simulator State Management

In addition to enhanced data locality, this state management scheme circumvents the hardware limitation of small LS size, while hiding memory latency by allowing for prefetching the state data of the *next* active Simulator concurrently with event execution for the *current* one.

5.4.4. Decentralized Event Management

For efficient DMA transfer of input and output events for the virtual LPs, the raw data included in all types of CD++ event objects are encoded in a uniform-sized C struct of 32

bytes; and a pair of flat 128-byte aligned arrays, referred to as the ***current event buffer*** and the ***backup event buffer*** respectively, is allocated in the main memory to exchange the *simultaneous* events passed between the FC and the virtual Simulators during a simulation phase. Each event buffer entry has an adjustable size of 1KB to hold up to 32 events at a time for a dedicated virtual Simulator.

To allow for double-buffered DMA transfer of event data, a pair of 128-byte aligned ***local event caches*** is allocated in the LS of an SPE to contain the input/output events for the *current* and the *next* active Simulators. Each local event cache has the same size as an event buffer entry in the main memory.

At any step of a simulation phase, the FC and a Simulator may exchange exactly one control message (either an (I), or (@), or (*) event) and *optionally* a list of content messages (either (X) or (Y) events). Hence, the first slot in each event buffer entry is reserved for passing the control messages, whereas the following slots are used for passing the content messages, if any. This convention allows the FC and the Simulators to immediately separate the control event from the other content events, without even checking the actual event types during event processing.

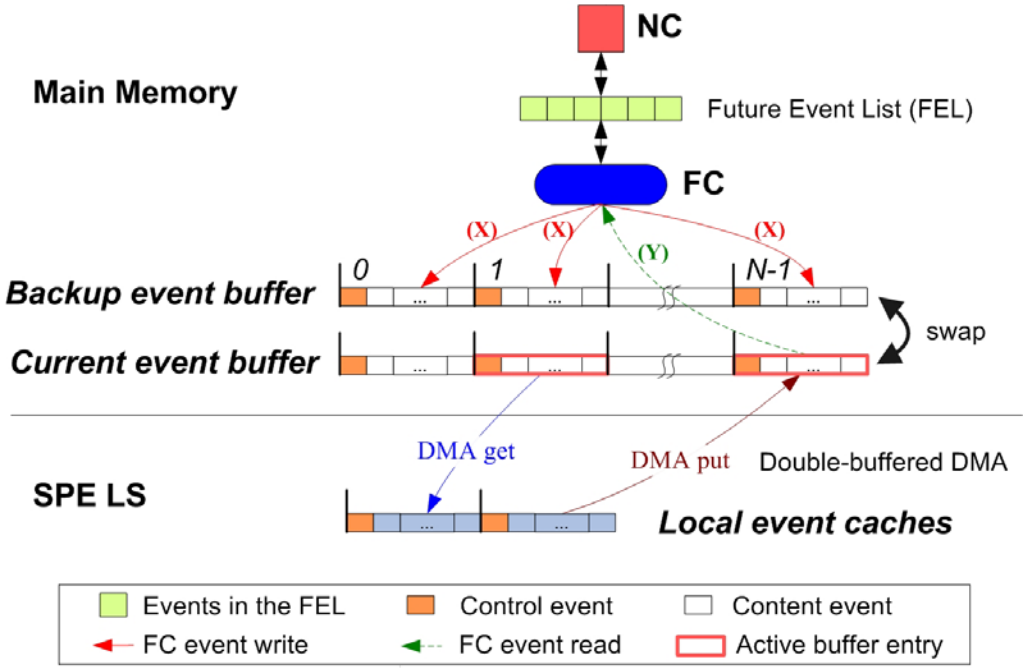


Figure 33. Virtual Simulator Event Management

As shown in Figure 33, the original FEL is used to send *phase-changing events* between the FC and the NC only at the beginning and the end of each simulation phase.

Together, the FEL and the event buffer entries form a network of bidirectional communication channels with a star topology centered at the FC.

During a collect phase, the FC translates (Y) events received from the source Simulators into (X) events that will be subsequently processed by the destination Simulators (refer to line 4.5 to 4.9 in Figure 7). Instead of putting these (X) events into the entries of the *current event buffer*, the FC writes them into the corresponding entries of the *backup event buffer*. Hence, using a pair of event buffers allows the FC to process the (Y) events, received from some of the virtual Simulators, on the PPE concurrently with event execution of the other virtual Simulators on the SPEs without additional synchronization, facilitating the exploitation of event-streaming parallelism. After writing the (*) events as well as any additional (X) events into the backup event buffer entries at the beginning of the ensuing transition phase (refer to line 5.3 to 5.13 in Figure 7), the FC resets an integer flag, referred to as *eventBufferIndex* (either 0 or 1), to swap the two event buffers. On the other hand, the virtual Simulators always work on the *current event buffer* as determined by the FC.

This decentralized event management scheme has several major advantages. First, multiple input/output events of a Simulator are stored in the same event buffer entry, allowing them to be transferred efficiently in a single DMA operation. Secondly, all of the simultaneous events are removed from the FEL, reducing event queue operational overhead. Thirdly, the simultaneous events are read and written directly in the two event buffers without the need for dynamic memory allocation and deallocation, further decreasing the operational cost. Fourthly, the Simulators no longer need to process (X) events during collect phases, reducing the overhead of DMA operations required for transferring event data and simplifying the SEK algorithms. As the (X) and (*) events targeting a Simulator are packed together in the same event buffer entry by the FC, they can be consumed as a whole in the transition phases, satisfying the P-DEVS event-processing requirement as mentioned in Section 5.4.1. Finally, the hardware-controlled PPE cache memory would also be better utilized because of increased event data locality in the main memory.

5.4.5. Evaluating Local Transition Functions on SPE

As mentioned earlier, the behavior of Cell-DEVS atomic models is specified using a set of *local transition functions* coded in the CD++ specification language [Wai02b]. Each local

transition function consists of several *state transition rules*, which are evaluated sequentially by active cells at each virtual time to determine their future states. A state transition rule is composed of three expressions separated by spaces, defining a *postcondition*, a *delay*, and a *precondition* for the rule. During evaluation, a state transition rule is fired if its precondition is found to be true; and the rule's postcondition defines the cell's next state, which will be sent to the neighboring cells after a period calculated from the delay expression. In the original CD++ simulator, these state transition rules are represented as *syntax trees* (one syntax tree for each rule expression), which are loaded into system main memory during simulation bootstrap, and the evaluation is performed *recursively* at runtime [Wai02b].

However, recursive computation on the SPEs is problematic due to the very limited size of runtime call stack and the lack of stack overflow protection on these cores [IBM09]. Furthermore, the syntax trees are not well-suited for efficient DMA transfer on the Cell processor. To solve these problems, the syntax trees derived from the state transition rules of each local transition function are converted into a sequence of floating-point values organized in *postfix* format and concatenated in a flat 128-byte aligned array called the ***rule buffer*** in the main memory, as shown in Figure 34. A packed state transition rule has four components, including a **precondition tree (*CT*)**, a **delay tree (*DT*)**, a **postcondition tree (*PT*)**, and a ***rule header*** that gives the numbers of syntax nodes included in the syntax trees.

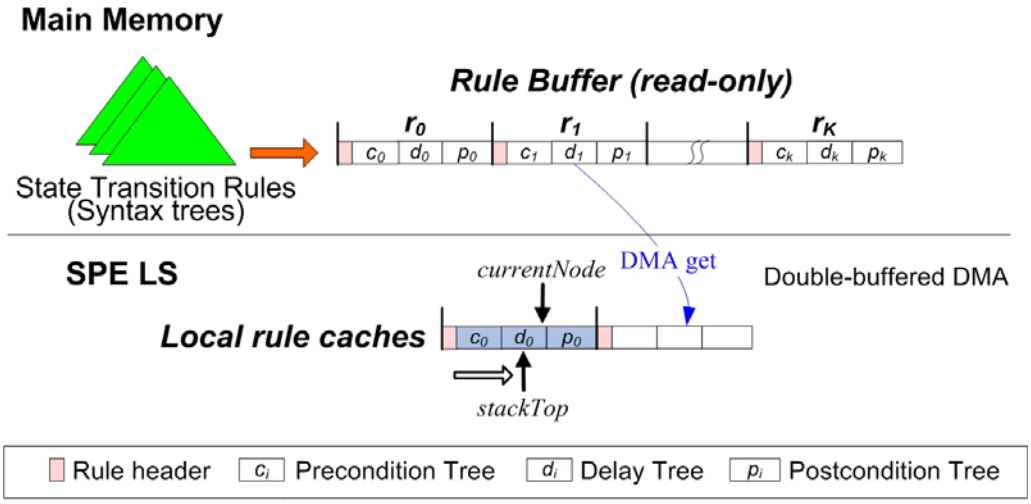


Figure 34. Evaluation of a Local Transition Function

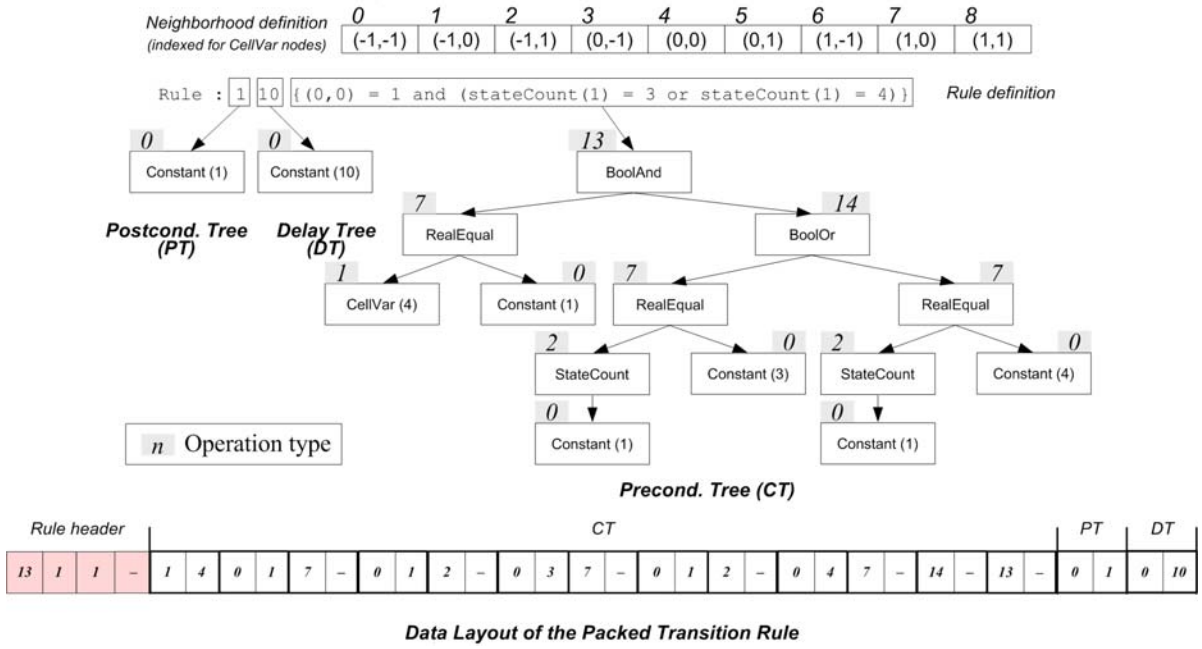
A pair of 128-byte aligned ***local rule caches*** is allocated in the LS of an SPE, where each local rule cache has a size that is sufficient to contain *any* one of the rules packed in a main memory rule buffer. Double-buffered DMA transfer is used to fetch the *next* rule from

a rule buffer while an SPE thread is busy evaluating the *current* one. Note that, unlike the state and event data, the transition rules are *read-only* in a simulation, allowing them to be accessed simultaneously by multiple SPE threads without the need for synchronization.

To pack the syntax trees into a rule buffer, a *rule-packing scheme* is used to *index* the various types of syntax nodes supported in the CD++ specification language using an *operation type* starting from 0. Each node in a syntax tree is represented by two values: an integer *operation type* and an optional floating-point *operand value* (an operand value is required to represent certain types of syntax nodes, while in the other cases, it is used as a placeholder to ensure a uniform size for all types of syntax nodes). Table 8 lists the representation of some of the CD++ syntax nodes.

Table 8. CD++ Syntax Node Representation (Partial)

Syntax Node	Op. Type	Representation	Description
Constant	0	*<0, constantValue>*	A constant value
CellVar	1	*<1, neighborIndex>*	Retrieve the value of a neighboring cell
StateCount	2	*<2, –>*	Count the neighboring cells with a given value
RealEqual	7	*<7, –>*	Compare two given operands (true if equal)
BoolAnd	13	*<13, –>*	Boolean AND of two given operands
BoolOr	14	*<14, –>*	Boolean OR of two given operands
…	…	…	…



Data Layout of the Packed Transition Rule

Figure 35. Transforming the Syntax Trees of a State Transition Rule

Using a state transition rule borrowed from the classical Life Game model [Gar70], as an example, Figure 35 illustrates this rule-packing scheme, which transforms the syntax trees of a rule into a flat data layout that is suitable for DMA transfer and computation on the SPEs. Note that the CT is packed before the PT and DT as the precondition will be evaluated first.

On the SPEs, the original recursive rule evaluation algorithm is replaced by an *iterative* one that scans a local rule cache one syntax node at a time. Moreover, by virtue of two pointers (shown as *currentNode* and *stackTop* in Figure 34), the local rule cache *itself* is used as a *software-managed call stack* to hold the intermediate operands (i.e., the stack is overlapped with the syntax nodes that have already been evaluated in the local rule cache), allowing the rules to be evaluated ***in place*** without burdening the SPE runtime call stack.

Figure 36 shows a skeleton of the SEK rule evaluation algorithm. For each type of syntax nodes, an *evaluation function* is defined on the SPEs. To reduce branching instructions during rule evaluation, these functions are called directly through a function pointer array, referred to as *evalFunctions*, which uses the operation types of the syntax nodes as array indexes.

These evaluation functions are defined in a similar way. Each of them performs the required operation based on operands retrieved from either the syntax node itself (e.g., line 1.2), or the state data available in the current local state cache (e.g., line 2.2), or the intermediate operands on top of the stack in the current local rule cache. The computation result is then pushed back into the stack at *stackTop*, growing and shrinking the software-managed call stack accordingly.

New types of syntax nodes can also be easily added, if desired, to meet the requirements of specific Cell-DEVS models. In this case, a modeler only needs to pack the new syntax nodes following the rule-packing scheme described earlier, and implement the corresponding evaluation functions (with new entries added to *evalFunctions*) on the SPEs.

When a local transition function is invoked in a Cell-DEVS atomic model, the *next* rule to be evaluated is prefetched into the *next* local rule cache (line 3.5), while the syntax trees available in the *current* local rule cache are executed (line 3.6 to 3.25). Once a valid rule is found (line 3.11), the cell's new state and delay values are computed (line 3.17 and 3.23) and the rule evaluation is terminated (line 3.24).

```
void (*evalFunctions[])(void) = {evalConstant, evalCellVar, ···}
currentNode : the address of the current syntax node (the optional operand is referred by currentNode+1)
stackTop : the top of the software-managed stack in the current rule cache
ruleCache0, ruleCache1 : the current and the next rule caches in the LS

//Definition of function evalConstant and evalCellVar (others are omitted here)
1.1.  when function evalConstant is invoked
1.2.     currentRuleCache[++stackTop] = constant in current node          //push the constant on stack
1.3.  end when

2.1.  when function evalCellVar is invoked
2.2.     Retrieve the value of a neighboring cell from the current local state cache using neighborIndex
2.3.     currentRuleCache[++stackTop] = retrieved cell value              //push the cell value on stack
2.3.  end when

3.1.  when a local transition function is evaluated
3.2.     Start an inbound DMA to fetch the 1st rule from the rule buffer into the currentRuleCache
3.3.     for each rule in the rule buffer do
3.4.        Wait for the current inbound DMA to complete in currentRuleCache
3.5.        Start the next inbound DMA to fetch the next rule into the nextRuleCache
3.6.        Reset currentNode = 0; stackTop = -1
3.7.        for each node in CT do
3.8.           Evaluate the node with (*evalFunctions[ currentRuleCache[currentNode] ])()
3.9.           Move currentNode to the next syntax node in CT
3.10.       end for
3.11.       if currentRuleCache[stackTop] is true then                    //i.e., if CT is found to be true
3.12.          Update stackTop for evaluation of PT
3.13.          for each node in PT do
3.14.             Evaluate the node with (*evalFunctions[ currentRuleCache[currentNode] ])()
3.15.             Move currentNode to the next syntax node in PT
3.16.          end for
3.17.          Record the new cell value obtained
3.18.          Update stackTop for evaluation of DT
3.19.          for each node in DT do
3.20.             Evaluate the node with (*evalFunctions[ currentRuleCache[currentNode] ])()
3.21.             Move currentNode to the next syntax node in DT
3.22.          end for
3.23.          Record the delay value obtained
3.24.          break;                                                     //Skip the remaining rules
3.25.       end if
3.26.    end for
3.27. end when
```

Figure 36. Double-Buffered Rule Evaluation on the SPEs

5.4.6. Processing SEK Jobs on SPE

During a simulation phase, an *SEK job* executes a set of events scheduled for an active Simulator that has been mapped to an SPE thread. The execution assumes that the event and state data of the active Simulator have already been made available in the current *local event*

cache and *local state cache* respectively in the LS, whereas memory control and SEK orchestration are considered as separate supporting services, which will be presented in the next section. Decoupling SEK job execution from these services allows a modeler to implement the output (λ) and state transition functions (δ_{int}, δ_{ext}, and δ_{con}) for P-DEVS atomic models on the SPEs in a way that is similar to what they do in the original sequential CD++ simulator (SIMD vectorization is desired, but not absolutely required, in the implementation of these P-DEVS functions), without having to cope with the details of DMA transfer and thread scheduling in the multicore environment, thus reducing the complexity of software development and promoting developer productivity with reduced M&S cost.

The SEK job-processing algorithms are encapsulated in three *job handlers*, referred to as `initJobHandler`, `collJobHandler`, and `transJobHandler`, which are invoked during the initialization, collect, and transition phases accordingly. Figure 37 gives a skeleton of the SEK job-processing algorithms.

In essence, they follow the definition of the Simulator event-processing algorithms for (I), (@), and (*) events respectively (refer to Figure 6 in Section 2.6.1), with a few exceptions. First, these SEK job-processing algorithms are *coarse-grained* because, as mentioned in Section 5.4.4, the local event cache may contain *multiple* input events (a control message and a list of content messages) scheduled for a Simulator. All of these input events are processed with a single invocation of the respective job handler. Secondly, the generated output events and updated state variables are *directly* written in the current local event and state caches, which will then be transferred back to the corresponding entries in the current event and state buffers in the main memory.

Note that, in the case of Cell-DEVS models, the P-DEVS functions have already been implemented in the SEK job-processing algorithms. Specifically, the SEK rule evaluation algorithm, as shown in Figure 36, is triggered in `transJobHandler` when the external (δ_{ext}) and confluent (δ_{con}) state transitions are performed at the cells (line 3.5 and 3.10). Besides, SPE SIMD intrinsics are also used to parallelize the handling of multiple content messages in the local event cache for Cell-DEVS models.

currentEventCache : the current event cache (input events of an active Simulator)
currentStateCache : the current state cache (state data of an active Simulator)
numOfContentEvents : number of content events currently available in *currentEventCache*
t_L, *ta*, t_N, and *e* are defined as usual and included in *currentStateCache*

1.1 **when** function **`initJobHandler`** is invoked
1.2 $t_L = 0$; *ta* = infinity
1.3 Initialize other state variables in *currentStateCache*
1.4 Write a (D, 0) event with *ta* into the 1st slot of *currentEventCache*
1.5 **end when**

2.1 **when** function **`collJobHandler`** is invoked
2.2 Get current virtual time t from (@, t) in the 1st slot of *currentEventCache*
2.3 t_L = t; *ta* = 0
2.4 Trigger output function in the atomic model, Y = λ(*currentStateCache*)
2.5 Write (Y, t) events into *currentEventCache* (starting from the 2nd slot)
2.6 Update *numOfContentEvents* to record the number of (Y, t) events written
2.7 Write a (D, t) event with *ta* into the 1st slot of *currentEventCache*
2.8 **end when**

3.1 **when** function **`transJobHandler`** is invoked
3.2 Get current virtual time t from (*, t) in the 1st slot of *currentEventCache*
3.3 **if** $t_L \leq$ t < t_N **then**
3.4 $e = t - t_L$; $ta = t_N$ - t
3.5 *currentStateCache* = δ_{ext}(*currentStateCache*, e, *currentEventCache*+1)
//Call the local transition function for a Cell-DEVS atomic model here
3.6 Set *numOfContentEvents* = 0 //(X, t) events consumed
3.7 **else if** t = t_N **and** *numOfContentEvents* = 0 **then**
3.8 *currentStateCache* = δ_{int}(*currentStateCache*)
3.9 **else if** t = t_N **and** *numOfContentEvents* != 0 **then**
3.10 *currentStateCache* = δ_{con}(*currentStateCache*, *currentEventCache*+1)
//Call the local transition function for a Cell-DEVS atomic model here
3.11 Set *numOfContentEvents* = 0 //(X, t) events consumed
3.12 **end if**
3.13 t_L = t
3.14 Write a (D, t) event with *ta* into the 1st slot of *currentEventCache*
3.15 **end when**

Figure 37. SEK Job-Processing Algorithms

On the PPE side, the IDs of the active Simulators are used as the SEK job IDs, which are scheduled by the FC in each simulation phase through a set of ***pending job queues*** (one for each SPE thread that hosts an instance of the SEK). Each job queue is a 128-byte aligned integer array containing the pending job IDs for an SEK, as depicted in Figure 38. A pair of ***local job caches*** is allocated in an SPE's LS so that the job IDs can be transferred across

memory domains *in chunks* using double-buffered DMA. Each chunk has an adjustable size of 32 job IDs (128 byte in total). An SEK processes the pending job IDs available in the *current* local job cache sequentially. These job IDs are used as *offsets* to calculate the addresses of the event and state buffer entries in the main memory when accessing the data of active Simulators from the SPEs.

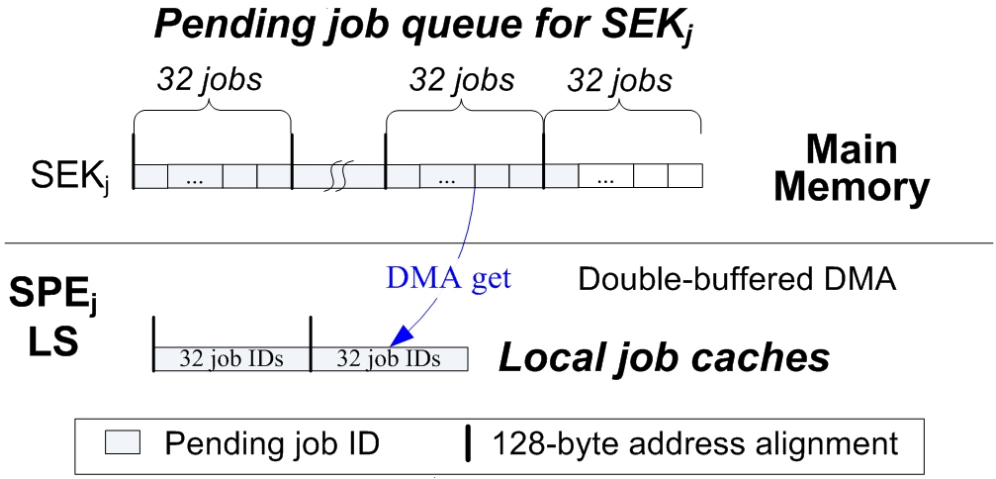


Figure 38. Pending Job Queue for an SEK

At the beginning of a simulation phase, the FC first executes the phase-changing events in the FEL, and then directly writes the generated events into the event buffer entries based on the IDs of the receiving Simulators. These Simulator IDs are inserted into the pending job queues under a certain *job-scheduling policy*, thus mapping the active Simulators to the SEKs. Since the SEK jobs executed in a simulation phase are of the same type with similar computational intensity, simple yet effective scheduling policies (e.g., round-robin, shortest-queue-first, or weighted round-robin) can be used to achieve *fine-grained* dynamic load-balancing between the SPE threads.

5.4.7. An SEK Memory Control and Notification Algorithm

Several key services must be provided on the SPEs to support the SEK job-processing algorithms given in Figure 37. These services include a *memory control service* for efficient prefetching of simulation data using double-buffered DMA transfer and a *notification service* for notifying the PPE code about the progress of job processing on an SPE. Together, they constitute the *SEK memory control and notification algorithm*, which is encapsulated in a function called `provideService`. Figure 39 shows a skeleton of the algorithm.

```
NOTIFY_FREQ : user-defined SEK notification frequency (one signal per NOTIFY_FREQ job completions)
JOB_CACHE_SIZE : user-defined size of the local job caches

void (*jobFunctions[])(void) = {initJobHandler, collJobHandler, transJobHandler}

numOfSignals : number of signals to be sent to the PPE when processing a chunk of pending jobs
jobIndex : the index of a pending job ID in the current local job cache

curEventBufAddr : the main memory starting address of the current event buffer
stateBufAddr : the main memory starting address of the state buffer
eventBufEntryAddr : the main memory address of an event buffer entry
stateBufEntryAddr : the main memory address of a state buffer entry

jobCache0, jobCache1 :  the current and the next local job caches
eventCache0, eventCache1 : the current and the next local event caches
stateCache0, stateCache1 :  the current and the next local state caches

sekCtrMsg : a control message received from the PPE to invoke the SEK on this SPE thread
totalPendingJobs : total number of pending jobs to be processed (given by PPE)

1.  when function provideService is invoked
2.     Schedule an inbound DMA to transfer the 1st chunk of pending job IDs into the currentJobCache
3.     Calculate curEventBufAddr based on sekCtrMsg
4.     Calculate the number of pending job chunks based on totalPendingJobs and JOB_CACHE_SIZE
5.     for each full chunk of job IDs in the pending job queue do
6.        Reset jobIndex = 0
7.        Wait for the current inbound DMA to complete in the currentJobCache
8.        Schedule an inbound DMA to transfer the next chunk of pending job IDs into the nextJobCache
9.        Calculate eventBufEntryAddr based on curEventBufAddr and  currentJobCache[jobIndex]
10.       Schedule an inbound DMA to transfer input events from eventBufEntryAddr into the currentEventCache
11.       Calculate stateBufEntryAddr based on stateBufAddr and currentJobCache[jobIndex]
12.       Schedule an inbound DMA to transfer state data from stateBufEntryAddr into the currentStateCache
13.       Update eventBufEntryAddr and stateBufEntryAddr for the next job with ID currentJobCache[jobIndex+1]
14.       Calculate numOfSignals = JOB_CACHE_SIZE / NOTIFY_FREQ
15.       for n = 0 to (numOfSignals-1) do
16.          for j = 0 to (NOTIFY_FREQ-1) do
17.             Wait for the current inbound DMA to complete in the currentEventCache and currentStateCache
18.             Wait for the previous outbound DMA to complete in the nextEventCache and nextStateCache
19.             Schedule an inbound DMA to transfer input events from eventBufEntryAddr into the nextEventCache
20.             Schedule an inbound DMA to transfer state data from stateBufEntryAddr into the nextStateCache
21.             Call SEK job-handling function with (*jobFunctions[ functionID derived from sekCtrMsg ])()
22.             Schedule an outbound DMA to transfer output events from currentEventCache to the original eventBufEntryAddr
23.             Schedule an outbound DMA to transfer updated state from currentStateCache to the original stateBufEntryAddr
24.             jobIndex++
25.             Update eventBufEntryAddr and stateBufEntryAddr for the next job with ID currentJobCache[jobIndex+1]
26.          end for                          //NOTIFY_FREQ jobs have been processed in the pending job queue
27.          Wait for the last outbound DMA to complete
28.          Send the actual number of finished SEK jobs (NOTIFY_FREQ) to the PPE through the outbound mailbox channel
29.    end for
30.    //Code for handling incomplete chunk of job IDs in the pending job queue is omitted here
31. end when
```

Figure 39. SEK Memory Control and Notification Algorithm

The purpose of the memory control service is to fetch the input events and states of the active Simulators into the local event and state caches, and to transfer the generated output events and updated states back to the original event and state buffer entries in the main memory after job execution. The transfer of event and state data, however, relies on the availability of pending job IDs in the local job caches since these job IDs are used to

calculate the main memory addresses of the corresponding event and state buffer entries (line 9, 11, 13, and 25). Hence, double-buffered DMA transfer needs to be considered at two distinct layers, including the *job-data layer* for transfer of pending job IDs in chunks (line 2 and 8) and the *simulation-data layer* for transfer of event and state data for each individual job within a chunk (line 10, 12, 19, 20, 22 and 23). Once the simulation data become available in the local event and state caches, the corresponding SEK job handlers are invoked using a function pointer array called *jobFunctions* (line 21), allowing for overlapping SEK job execution with concurrent memory I/O on an SPE.

On the other hand, the notification service sends signals periodically to the PPE code, indicating the actual number of SEK jobs that have been processed in the pending job queue since the last notification (line 28). In this way, the FC can process the output events generated from those Simulators without waiting for the completion of all pending jobs, exploiting event-streaming parallelism between the PPE and the SPEs on a Cell processor. The frequency of notification (NOTIFY_FREQ) can be adjusted at compile time.

Note that a control message, ***sekCtrMsg***, is used to determine the address of the *current event buffer* in the main memory (line 3) and the function ID of a proper SEK job handler (line 21). Moreover, the total number of job IDs in the pending job queue, ***totalPendingJobs***, must be known in order to schedule DMA transfers on the SPEs (line 4). These two variables are initialized at the beginning of each simulation phase in the SEK main loop, as will be presented in the next section.

5.4.8. SEK Orchestration Algorithms

When a simulation starts, the addresses of the various buffers allocated in the main memory are passed, through a control block, to a group of SPE threads dedicated to SEK execution. Similar to the FSK, the SEK also includes a termination function, referred to as `terminateSEK`, which is invoked at the end of a simulation to exit SEK execution on an SPE thread. Figure 40 gives the definition of the SEK main loop and the termination function.

The SEK computation is triggered on an SPE thread upon the arrival of two mailbox messages (i.e., *sekCtrMsg* and *totalPendingJobs* as defined in Figure 39) (line 2.5 to 2.6). In response, either `provideService` or `terminateSEK` is invoked through a function pointer array, referred to as *SEKFunctions*, based on the value of *sekCtrMsg* (line 2.7).

```
void (*SEKFunctions[])(void) = {provideService, terminateSEK};
stopFlag : a flag used to terminate the SEK main loop (initialized to false);

1.1. when function terminateSEK is invoked
1.2.    Set stopFlag to true
1.3. end when

2.1. when an SPE thread is started to execute the SEK
2.2.    Schedule an inbound DMA to fetch the control block from the main memory
2.3.    Wait until the control block becomes available in the LS
2.4.    While stopFlag is false do
2.5.       Read sekCtrMsg from the inbound mailbox channel (blocking read)
2.6.       Read totalPendingJobs from the inbound mailbox channel (blocking read)
2.7.       Call an SEK function with (*SEKFunctions[ function ID derived from sekCtrMsg ])()
2.8.    end while
2.9. end when
```

Figure 40. SEK Main Loop and Function `terminateSEK`

While *totalPendingJobs* is just an integer that indicates the total number of pending jobs that have been scheduled for an SEK, *sekCtrMsg* encodes three parameters compactly in its four least significant bits, as illustrated in Figure 41. These parameters are then extracted with bit operations at different points in the SEK algorithms (*eventBufferIndex* and *SEKJobHandlerID* are used in line 3 and line 21 of Figure 39 respectively, and *SEKFunctionID* is used in line 2.7 of Figure 40).

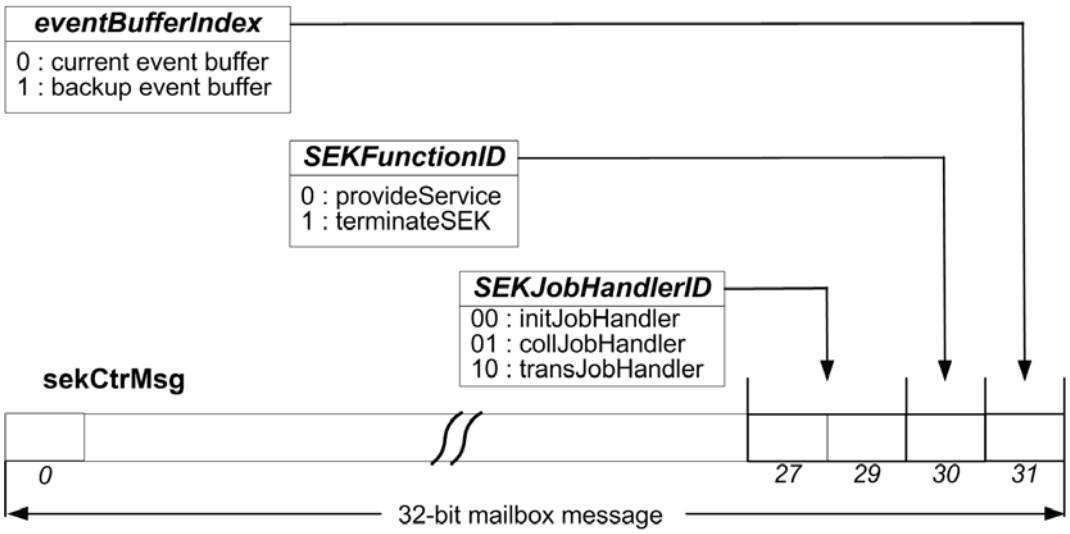


Figure 41. SEK Control Message Bit Pattern

```
//constants for setting the bits in sekCtrMsg (already reflecting currentPhase and SEKFunctionID)
SEK_INIT_SIG = 0;              SEK_TERMINATE_SIG = 2;
SEK_COLLECT_SIG = 4;       SEK_INTERNAL_SIG = 8;

SEK_NUM : number of SPE threads devoted to SEK execution (given by user at compile time)

eventBufferIndex : determine the current event buffer (0 or 1), initialized to 0
numOfPendingJobs[SEK_NUM] : record the numbers of pending jobs scheduled for the SEKs

1.1.  when an (I, 0) is received from the NC at the beginning of an initialization phase
1.2.     Execute the received (I, 0) event (see line 1.2 in Figure 7)
1.3.     Insert all Simulator IDs into the job queues using a given job-scheduling policy
1.4.     Set sekCtrMsg = SEK_INIT_SIG | eventBufferIndex
1.5.     for each SEK_i do
1.6.        Send sekCtrMsg through the inbound mailbox channel
1.7.        Send numOfPendingJobs[i] through the inbound mailbox channel
1.8.     end for
1.9.     //Wait for notifications from the SEKs and process output events in parallel on PPE
1.10. end when

2.1.  when a (@, t) is received from the NC at the beginning of a collect phase
2.2.     Execute the received (@, t) event (see line 2.2 to 2.7 in Figure 7, excluding line 2.6)
2.3.     Insert imminent Simulator IDs into the job queues using a given job-scheduling policy
2.4.     Set sekCtrMsg = SEK_COLLECT_SIG | eventBufferIndex
2.5.     for each SEK_i do
2.6.        if numOfPendingJobs[i] is not zero then
2.7.           Send sekCtrMsg through the inbound mailbox channel
2.8.           Send numOfPendingJobs[i] through the inbound mailbox channel
2.9.        end if
2.10.    end for
2.11.    //Wait for notifications from the SEKs and process output events in parallel on PPE
2.12. end when

3.1.  when a (*, t) is received from the NC at the beginning of a transition phase
3.2.     Execute the received (*, t) event (see line 5.2 to 5.13 in Figure 7, excluding line 5.5 and 5.11)
         //Note: (X, t) and (*, t) events are written into the backup event buffer for the Simulators
3.3.     Update eventBufferIndex = (eventBufferIndex + 1) & 1              //swap event buffer
3.4.     Insert active Simulator IDs into the job queues using a given job-scheduling policy
3.5.     Set sekCtrMsg = SEK_INTERNAL_SIG | eventBufferIndex
3.6.     for each SEK_i do
3.7.        if numOfPendingJobs[i] is not zero then
3.8.           Send sekCtrMsg through the inbound mailbox channel
3.9.           Send numOfPendingJobs[i] through the inbound mailbox channel
3.10.       end if
3.11.    end for
3.12.    //Wait for notifications from the SEKs and process output events in parallel on PPE
3.13. end when
```

Figure 42. SEK Orchestration Algorithm on PPE (Part I)

On the PPE side, as shown in Figure 42, the FC sends two mailbox messages to the SEKs at the beginning of each simulation phase when the pending job IDs have been inserted into the job queues based on some job-scheduling policy (line 1.5 to1.8, line 2.5 to 2.10, and line 3.6 to 3.11). After that, the FC waits for notification signals from the SEKs (line 1.9, 2.11, and 3.12). Once notified, the FC can immediately proceed to process the generated output events that have been made available in the corresponding entries of the current event buffer, thus overlapping SEK execution on the SPEs with output event processing on the PPE. This part of SEK orchestration algorithm is encapsulated in a function called `processSEKOutputs`, which is given in Figure 43.

```
totalPendingJobs : sum of numOfPendingJobs[i] (i.e., total number of pending jobs)

1.1.  when function processSEKOutputs is invoked by the FC
1.2.    while totalPendingJobs is not zero do
1.3.      for each SEK that has been executed do
1.4.        non-blocking check outbound mailbox channel for potential SEK notifications
1.5.        if a notification signal is received from SEK_i then
1.6.          Mark the completed job IDs in the corresponding job queue
1.7.        end if
1.8.      end for
1.9.      for each non-empty job queue do
1.10.       for each completed SEK job whose output events have not yet been processed do
1.11.         Process the output events in the current event buffer entry
1.12.       end for
1.13.       break;                          //go back to polling (line 1.3)
1.14.     end for
1.15.   end while
1.16.   Process the output events for any remaining completed SEK jobs in the job queues
1.17.   Send phase-changing events back to the NC via the FEL
1.18. end when
```

Figure 43. SEK Orchestration Algorithm on PPE (Part II)

The FC performs two major tasks in function `processSEKOutputs`, namely checking for notification signals from the SEKs (line 1.3 to 1.8) and processing output events for the completed SEK jobs (line 1.9 to 1.14). Note that an SPE's outbound mailbox channel can contain at most one message at a time. That is, if a previously sent notification signal has not yet been received by the PPE, an SEK will be blocked when it tries to send another signal after processing NOTIFY_FREQ more jobs. To reduce the possibility of blocking at the SEKs, the FC uses *non-blocking* polling to quickly scan the status of the outbound mailbox channels (line 1.4), and processes the output events from *just one* SEK (even when output

events have been produced from multiple SEKs) before going back to poll the channels again (line 1.13). In other words, as long as there are still notifications to be received from the SEKs, the algorithm tries to shorten the channel-polling interval, while at the same time allowing for concurrent event execution on the PPE. When all of the pending jobs are finished by the SEKs, the FC executes the remaining output events (line 1.16) and sends phase-changing events to the NC via the FEL (line 1.17), ending the current simulation phase.

```
1. when the simulation is about to be terminated (line 3.11 in Figure 8)
2.    Set sekCtrMsg = SEK_TERMINATE_SIG
3.    for each SEK_i do
4.       Set numOfPendingJobs[i] = 0
5.       Send sekCtrMsg through the inbound mailbox channel
6.       Send numOfPendingJobs[i] through the inbound mailbox channel
7. end when
```

Figure 44. Terminating SEKs at the NC

At the end of a simulation, the SEKs are terminated by the NC with two special mailbox messages (SEK_TERMINATE_SIG and zero), as shown in Figure 44.

5.5. Parallel DEVS Simulation on the Cell Processor

Based on the FSK and SEK algorithms presented in the previous two sections, Figure 45 gives an architectural overview of the proposed MADS computing technique, showing the major thread components along with the data flow and interactions between them.

During simulation bootstrap, the *PPE main thread* spawns a *PPE helper thread*, which in turn creates a set of *SPE threads* (one on each SPE). The NC and the FC are executed respectively by the two PPE threads, which share the FEL in the main memory to process phase-changing events in a producer-consumer fashion. The SPE threads, on the other hand, are divided into two groups: one for the FSKs and the other for the SEKs. More SPE threads should be reserved for the FSKs in large-scale and long-running simulations with moderate model complexity, whereas more SPE threads are needed to speed up the SEKs in medium-sized simulations of complex model behavior over a relatively short period of virtual time.

The actual number of SPE threads to be created in each group can be adjusted at compile time based on the knowledge of the simulated model. If the number of threads is set to zero for a group, the corresponding computational kernel will be hosted on the PPE instead. This is useful in models where the overall simulation performance is dominated

overwhelmingly by just one type of the kernels (e.g., the wildfire and watershed simulations as analyzed in Section 5.2), thus obviating the need for parallel execution of the other type of kernel that constitutes only a negligible performance bottleneck.

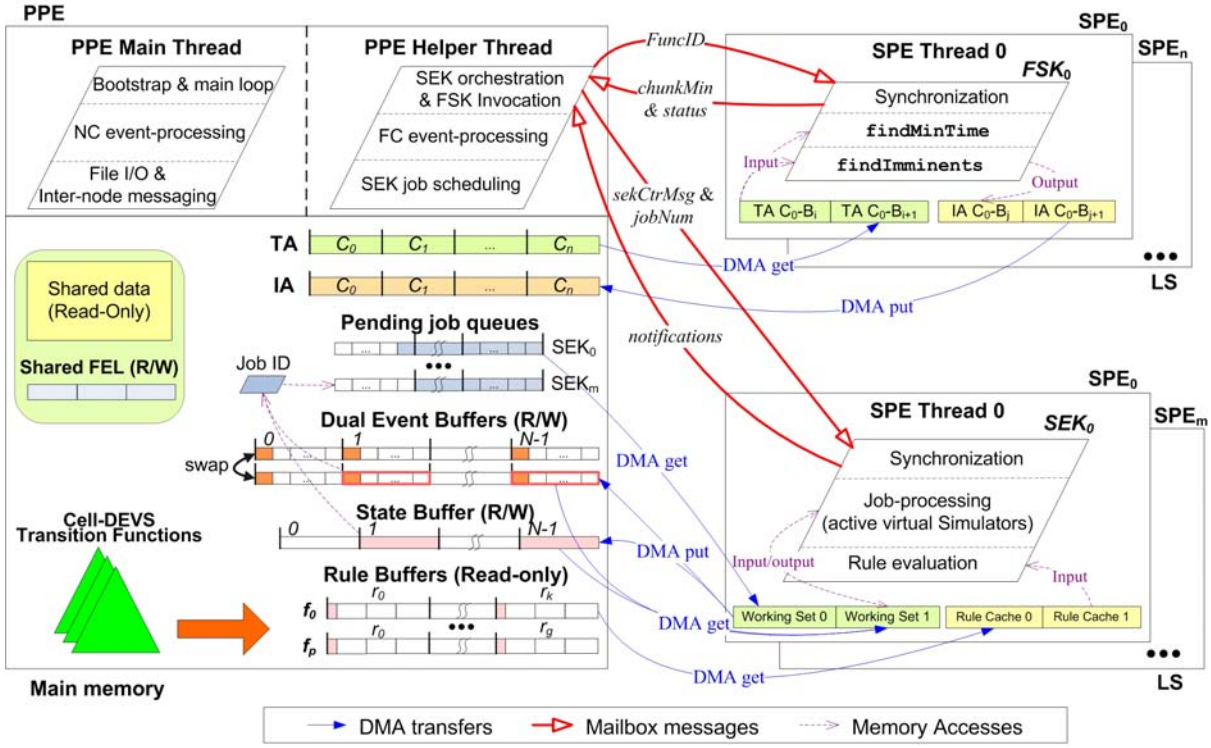


Figure 45. Architectural Overview of the MADS Technique

The following is a summary of the multi-grained parallelization strategies employed in the FSK and SEK algorithms.

- **Multi-grained parallelization strategy for the FSK**.

The FSK algorithms realize multi-grained parallelism on the Cell processor as follows. ***Thread-level parallelism*** is applied across multiple SPEs, each of which hosts an instance of the FSK that works on a corresponding pair of TA and IA chunks to exploit the massive ***data-level parallelism*** existed in the FC synchronization task. On an SPE thread, ***data-streaming parallelism*** is utilized to process the chunk of data as a stream of blocks, hiding memory latency with double-buffered DMA transfers. In addition, the synchronization functions are implemented using explicit SPE SIMD intrinsics to explore ***vector parallelism***. Moreover, ***loop-level parallelism*** is used to further accelerate data processing on the SPEs by virtue of simultaneous logical threads.

- **Multi-grained parallelization strategy for the SEK**.

Taking advantage of the fine-grained ***event-level parallelism*** inherent in the DEVS-based simulation process, the SEK algorithms combine multi-grained parallelism at different system levels, as follows. ***Thread-level parallelism*** is achieved both on the PPE itself and between the PPE and a group of SPEs, where each SPE thread executes an instance of the SEK. During each simulation phase, the independent events targeting different active Simulators are executed concurrently at distinct SEKs, realizing ***event-embarrassing parallelism***. Moreover, ***event-streaming parallelism*** is utilized by executing the causally-dependent events passed between the Simulators (running on the SPEs) and the FC (running on the PPE) in a two-stage pipeline. Double-buffered DMA is applied at multiple layers to transfer pending job IDs, simulation data of individual jobs, and rule data in the case of Cell-DEVS models, hiding memory latency with ***data-streaming parallelism***. On each SPE thread, the SEK job-processing algorithms are implemented with SPE SIMD intrinsics whenever possible to explore ***vector parallelism***. Due to the irregular nature of the event computation, only partial vectorization is applied to parallelize the most time-consuming loops in the SEK code. Throughout a simulation, the PPE main thread handles file I/O and inter-node messaging in parallel with event computation at the helper and the SPE threads, exploiting the PPE hardware SMT capability to realize ***compute-I/O parallelism***.

5.6. Implications of the MADS Technique

This section briefly discusses the implications of the proposed MADS technique, including possible approaches to accommodating additional types of computational kernels, supporting P-DEVS and Cell-DEVS simulations that cannot be fully parallelized on the SPEs, and integrating with existing cluster-based PDES techniques on hybrid super cluster systems. It also summarizes several generalizable concepts and methods of the MADS technique for PDES on other CMP architectures.

- **Accommodating additional computational kernels.**

The MADS technique is intended to accelerate two types of typical computational kernels commonly found in demanding DEVS-based simulations. However, this does not exclude the possibility of incorporating other types of computational kernels (e.g., random number generation) into the software architecture to address the needs of specific models. To

accommodate a new computational kernel, a user can reserve a group of SPEs to host the parallelized kernel computation, while using the rest of the SPEs to support the FSK and/or the SEK. In an extreme case, both FSK and SEK can be executed on the PPE if the new kernel becomes the solely predominant bottleneck in a simulation. The PPE helper thread can also be extended to orchestrate the SPE threads dedicated to the new kernel. In this sense, the MADS technique allows for a modular, extensible software architecture that can be adapted to varied performance and simulation requirements.

- **Supporting P-DEVS and Cell-DEVS models with SPE-incompatible components.**

Although the FSK can always be parallelized on a Cell processor, if desired, in *any* P-DEVS and Cell-DEVS simulations, it may *not* always be possible, or suitable, to port *all* of the Simulators (and their associated atomic models) to the SPEs. In some cases, as pointed out in Section 5.4.2, a portion of the Simulators have to be implemented as *concrete LPs* on the general-purpose PPE. One example is that the state transition functions defined in certain atomic models require frequent access to a legacy library that is unsuitable to be hosted on the co-processors (e.g., due to irregular memory access patterns, excessive working storage requirements, or control-intensive pointer-chasing computations that are not straightforward to be redesigned for efficient execution on the SPEs). As a result, a simulation may include a mix of both *virtual* and *concrete* Simulators, which must be scheduled properly during each simulation phase.

This issue can be solved within the MADS framework as follows. During simulation bootstrap, the concrete Simulators are created *after* those virtual Simulators so that they are associated with greater process IDs (i.e., virtual Simulators use IDs in the range of [0…M-1], while concrete Simulators use IDs in the range of [M…N-1], where *M* and *N* represent the *number of virtual Simulators* and the *total number of Simulators* respectively). The event and state data of both types of Simulators are still managed in the event and state buffers as usual. For concrete Simulators, the SEK job-processing algorithms (Figure 37) are implemented on the PPE to directly operate on the corresponding event and state buffer entries in the main memory, while the SEK orchestration algorithm (Figure 42) is enhanced accordingly so that the concrete Simulators can process events on the PPE in parallel with virtual Simulator event-processing on the SPEs in a simulation phase.

- **Integrating with cluster-based PDES techniques**.

The MADS technique can also be integrated with cluster-based PDES techniques to achieve both conservative and optimistic parallel simulation on hybrid multiprocessor clusters with multicore nodes.

Integration of the MADS technique with cluster-based conservative PDES algorithms (both synchronous and asynchronous) can be realized in a relatively straightforward way. Since inter-node messaging is handled by the NC at the PPE main thread (refer to Property 2 in Section 2.6.3), the parallelized execution of computational kernels on a multicore node, carried out by the FC and the Simulators at the PPE helper thread and the SPE threads, is essentially invisible at the cluster level, thus allowing existing conservative synchronization algorithms to be implemented at the PPE main thread in a similar way as before.

On the other hand, combining the MADS technique with cluster-based optimistic PDES algorithms (e.g., the TW protocol) is a more elaborate task, mainly because the TWLPs are required to perform complex checkpointing and rollback operations using sophisticated data structures, a control-intensive computation with highly-irregular memory access pattern, posing a significant challenge to porting the TWLPs to the SPEs on a Cell processor. This task, however, can be greatly simplified by taking advantage of the LTW protocol proposed in Chapter 4. Under the LTW protocol, as analyzed in Section 4.5.1, the Simulators are turned into *lightweight LPs* that execute in a manner quite similar to that in a sequential simulation. As the lightweight LPs are shielded from the complexity of optimistic synchronization at the cluster level, they can be readily implemented as *virtual LPs* on a Cell processor using the MADS technique presented in this chapter.

- **Generalizing the MADS concepts for PDES on CMP architectures**.

Several key concepts and methods derived from the MADS technique could be generalized to achieve efficient PDES on other CMP architectures. First of all, the MADS technique directly tackles different types of computational kernels identified in the simulation process, providing an alternative method that would be more effective in alleviating PDES performance bottlenecks on CMP architectures than the traditional LP-oriented model-decomposition approach. Moreover, the proposed methods for simultaneous exploitation of multi-grained parallelism at different system levels would offer valuable

insight on developing new PDES algorithms for leveraging the architectural features of emerging multicore processors. In view of the fact that the performance discrepancy between memory latency and processor speed continues to widen (a problem known as the "Memory Wall" [McK04]), the data restructuring and macroscopic prefetching strategies employed in the MADS technique would shed some light on how to organize a PDES program from a data-flow perspective in order to increase data locality and ensure timely data availability on CMP architectures in general. Furthermore, the concept of LP virtualization could be extended to other multicore and shared-memory multiprocessors to improve processor utilization, to address resource constraints of the underlying hardware architecture, and to realize fine-grained load balancing. Last but not least, the methods used in the MADS technique for hiding the technical details of multicore programming from non-expert users would also be of practical importance in designing PDES systems on CMP architectures to enhance system usability, to promote modeler productivity, and to reduce M&S cost.

Chapter 6. Performance Analysis

This chapter analyzes the performance of the LTW protocol and the MADS technique proposed in the previous two chapters. Section 6.1 introduces the benchmark models used in the experiments. Section 6.2 summarizes the experimental configurations and performance metrics. Section 6.3 presents a comparative performance evaluation of the TW and the LTW protocols on distributed-memory multiprocessor clusters, while Section 6.4 evaluates the potential of the MADS technique to accelerate large-scale DEVS-based simulations on the Cell processor.

6.1. Introduction to the Benchmark Models

Two realistic environmental models with varied workload characteristics were tested in the experiments, namely a *wildfire propagation model* and a *watershed model*. These models have been studied extensively in the DEVS research community (see, e.g., wildfire simulation [Nta04, Hu07, Nta08, and Fil09] and watershed simulation [Zei93, Moo96, Ame01, and Bro09]), thus allowing them to be used as *de facto* benchmarks for performance evaluation. In [Wai06], the wildfire and watershed models have been redefined as executable Cell-DEVS models in the CD++ specification language, as briefly described in this section.

6.1.1. Definition of a Wildfire Model

Two versions of the wildfire model were evaluated in the experiments, including a *simplified* version, referred to as ***Fire1***, which uses predetermined fire spread rates at reduced runtime computational cost; and a *generalized* version, referred to as ***Fire2***, which computes fire spread rates dynamically based on environmental parameters obtained at runtime, with a higher computational intensity. Both versions simulate fire propagation scenarios over 50 virtual hours in a 2D cell space.

- **The Fire1 model**.

The *Fire1* model [Wai06] uses the Rothermel method [Rot72] to obtain the spread rate in every direction prior to the simulation based on a specific set of environmental parameters

(a fuel model type number of 9, a southwest wind at a speed of 24.135 km/h, and a cell size of 15.24×15.24 m^2), as summarized in Figure 46.

```
Wind direction = 225.00 (bearing)  Wind speed = 24.135 km/h   Fuel model type = 9
Cell width = 15.24 m (E-W)         Cell height = 15.24 m (N-S)

Max spread rate = 17.967136
0° rate = 5.106976 distance = 15.24      45° rate = 17.967136 distance = 21.552615
90° rate = 5.106976 distance = 15.24     135° rate = 1.872060 distance = 21.552615
180° rate = 1.146091 distance = 15.24    225° rate = 0.987474 distance = 21.552615
270° rate = 1.146091 distance = 15.24    315° rate = 1.872060 distance = 21.552615
```

Figure 46. Predetermined Spread Rates for the *Fire1* Model [Wai06]

Figure 47 gives a skeleton of the *Fire1* model definition. A cell's value stands for the virtual time when the cell is ignited (zero for a non-burning cell). The precondition of a transition rule is used to detect the presence of fire in a specific neighboring cell. For example, the first rule will be triggered if the current cell (0,0) is non-burning and the southwest neighbor (1,-1) has already been ignited. Hence, the fire will spread to the current cell at virtual time (1,-1) + (21.552615/17.967136), which becomes the new value of the current cell. This value will be sent to the neighboring cells after a delay of (21.552615/17.967136) * 60000 ms, the interval between the current virtual time and the expected ignition time at the cell.

```
type : cell         dim : (1024,1024)   delay: inertial     border : nowrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,0) (0,1) (1,-1) (1,0) (1,1)
localtransition : FireBehavior

[FireBehavior]
rule : {(1,-1)+(21.552615/17.967136)} {(21.552615/17.967136)*60000} {(0,0)=0 and 0<(1,-1)}
rule : {(1,0)+(15.24/5.106976)} {(15.24/5.106976)*60000} {(0,0)=0 and 0<(1,0)}
rule : {(0,-1)+(15.24/5.106976)} {(15.24/5.106976)*60000} {(0,0)=0 and 0<(0,-1)}
...
```

Figure 47. A Skeleton of the *Fire1* Model Definition in CD++ [Wai06]

This *Fire1* model (with a size of 1024×1024) has been analyzed in Section 5.2.1, and it is used to evaluate both the LTW protocol and the MADS technique in this chapter.

- **The Fire2 model**.

To support more sophisticated wildfire simulation applications, the *Fire1* model has been generalized in this research to allow for determination of fire spread rates based on changing environmental parameters [Har08]. Specifically, the CD++ specification language has been extended to include a new syntax node, referred to as `fsr`, which calculates the spread rate in any given direction at runtime by invoking the `fireLib` library [Bev96].

Figure 48 shows a skeleton of the generalized *Fire2* model when defined under the same environmental conditions as shown in Figure 46.

```
type : cell         dim : (1024,1024)   delay: inertial     border : nowrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,0) (0,1) (1,-1) (1,0) (1,1)
localtransition : FireBehavior

[FireBehavior]
rule : {(1,-1)+(21.552615/fsr(225,9,273.4,225))} {(21.552615/fsr(225,9,273.4,225))*60000}
                                                                {(0,0)=0 and 0<(1,-1)}
rule : {(1,0)+(15.24/fsr(180,9,273.4,225))} {(15.24/fsr(180,9,273.4,225))*60000}
                                                                {(0,0)=0 and 0<(1,0)}
rule : {(0,-1)+(15.24/fsr(270,9,273.4,225))} {(15.24/fsr(270,9,273.4,225))*60000}
                                                                {(0,0)=0 and 0<(0,-1)}
...
```

Figure 48. A Skeleton of the *Fire2* Model Definition in CD++

Comparing Figure 47 with Figure 48, the spread rate in a given direction is no longer a fixed constant in the *Fire2* model. Instead, it is the computation result of the `fsr` syntax node based on a set of four parameters (i.e., azimuth, fuel type, wind speed, and wind direction), which are provided by the runtime environment of an application (e.g., a global weather service). Therefore, highly dynamic and realistic simulation results can be obtained by feeding real-time environmental data into the model. As expected, the time for processing a (*, t) event at the Simulators becomes 6.68 times longer than what is required in the *Fire1* model (calibrated on a 3.2GHz Intel Xeon processor), a significant increase in computational intensity.

Note that the *Fire2* model is only used to evaluate the LTW protocol on multiprocessor clusters. Executing this model on the Cell processor would require porting the third-party `fireLib` library to the SPEs, an undertaking that is beyond the scope of this dissertation.

6.1.2. Definition of a Watershed Model

The *Watershed* model, as defined in [Wai06], uses a 3D cell space to simulate water accumulation in a drainage basin over 30 virtual minutes under constant rain condition (7.62 mm/h) based on a set of hydrological equations [Zei97]. In addition, different types of ground soil (grasses and stones) are also considered in the *Watershed* model by defining zones with different local transition functions within the cell space. Figure 49 shows a skeleton of the *Watershed* model definition in the CD++ environment.

```
type : cell          dim : (320,320,2)   delay: inertial     border : nowrapped
neighbors : (-1,0,0) (0,-1,0) (0,0,0) (0,1,0) (1,0,0) (-1,0,1) (0,-1,1) (0,0,1) (0,1,1) (1,0,1)
zone : grass {(0,0,0)..(319,50,0)}             stones { (0,270,0)..(319,319,0) }
localtransition : hydrology

[grass]
rule : {0.07 + (0,0,0) - if(((((0,0,1) + (0,0,0))>((-1,0,1) + (-1,0,0)))),(((((0,0,0) + (0,0,1)
- (-1,0,0) - (-1,0,1))/1000) * (0,0,0))/1000),0) - if(((((0,0,1) + (0,0,0))>((1,0,1)  +
(1,0,0)))),(((((0,0,0) + (0,0,1) - (1,0,0) - (1,0,1))/1000) * (0,0,0))/1000),0) - if(((((0,0,1)
+ (0,0,0))>((0,-1,1)+(0,-1,0)))),(((((0,0,0) + (0,0,1) - (0,-1,0) - (0,-1,1))/1000) * (0,0,0))/
1000),0) - if(((((0,0,1) + (0,0,0))>((0,1,1) + (0,1,0)))),(((((0,0,0) + (0,0,1) - (0,1,0) -
(0,1,1))/1000) * (0,0,0))/1000),0) + if(((((-1,0,1) + (-1,0,0))>((0,0,1) + (0,0,0)))),((((-
1,0,0) + (-1,0,1) - (0,0,0) - (0,0,1)) * (-1,0,0))/1000),0) + if(((((1,0,1) + (1,0,0))>((0,0,1)
+ (0,0,0)))),((((1,0,0) + (1,0,1) - (0,0,0) - (0,0,1)) * (1,0,0))/1000),0) + if(((((0,-1,1) +
(0,-1,0))>((0,0,1) + (0,0,0)))),((((0,-1,0) + (0,-1,1) - (0,0,0) - (0,0,1)) * (0,-1,0))/
1000),0) + if(((((0,1,1) + (0,1,0))>((0,0,1) + (0,0,0)))),((((0,1,0) + (0,1,1) - (0,0,0) -
(0,0,1)) * (0,1,0))/1000),0) } 1000 { cellpos(2)=0 }
rule : { (0,0,0) } 1000 { t }

... //local transition functions for [stones] and [hydrology] are omitted here
```

Figure 49. A Skeleton of the *Watershed* Model Definition in CD++ [Wai06]

In the model, the height of accumulated water at a cell depends on the rain intensity, the water exchanged with the neighboring cells (both inflows and outflows), and the amount of water absorbed by ground soil of different types. A local transition function thus computes future height values for the cells at each virtual time, taking into account the initial water level, the cumulative rain precipitation, the dynamic water flow between the cells, and the specific soil condition. The 3D cell space is composed of two planes: plane 0 and plane 1. While the former represents the ever-changing heights of retained water at different cells, the latter defines the topographical configuration of the terrain that remains unchanged throughout a simulation.

This *Watershed* model (with a size of 320×320×2) has been analyzed in Section 5.2.2, and it is used in the performance analysis of both the LTW protocol and the MADS technique in this chapter.

6.2. Experimental Configurations and Performance Metrics

The **PCD++** simulator [Liu07] has been extended in this research to include the LTW protocol proposed in Chapter 4. The sequential **CD++/PPE** simulator, previously introduced in Section 5.2, has been parallelized on the Cell processor using the MADS technique proposed in Chapter 5, resulting in a new parallel simulator called **CD++/Cell**. The performance of the LTW protocol and the MADS technique was studied in the experiments using the PCD++ and CD++/Cell simulators respectively on their intended platforms.

Note that the performance results presented in the following sections not only depend on the degree of parallelism available in the tested models, but also depend on the specific experimental configurations summarized here. Consequently, they should be viewed as indicators of potential performance gain that is achievable by the proposed techniques.

6.2.1. Configuration and Metrics for the LTW Protocol

The performance of the LTW protocol was evaluated on a cluster of 28 HP Proliant DL140 servers running on Linux WS 2.4.21 and communicating over Gigabit Ethernet using MPICH 1.2.7 [Gro09], which is a portable implementation of the MPI standard [Gro99]. Each cluster node features dual 3.2GHz Intel Xeon processors with 1GB 266MHz main memory and 2GB disk swap space. Note that severe memory-swapping activities will occur if the maximum space requirement of a simulation approaches (or goes beyond) the physical limit of 1GB on a node. Moreover, a simulation will fail to complete when the memory usage cannot be contained within the maximum allowable virtual memory space of 3GB (i.e., the accumulated size of physical memory and disk swap space).

The benchmark models were executed in a series of stress tests with a total of 160 different test cases. For each test case, the performance of the LTW protocol was compared with that of the customized/optimized TW protocol as implemented in the PCD++ simulator (along with its WARPED middleware layer) [Liu07]. To ensure a fair comparison, the PCD++ simulator was configured to be the same for both TW and LTW simulations in all aspects, except those directly related to the LTW algorithms under study. In particular, both protocols employed aggressive cancellation, multi-list implementation of event set (applied to the persistent input queue in the LTW simulations), copy state-saving optimized with the MTSS strategy [Liu07] (the LTW simulations were further optimized with the enhanced risk-free state-saving strategy proposed in Section 4.4.2), and the pGVT algorithm [Kan96]. Moreover, the PCD++ event-logging capability was turned off in all test cases to minimize the impact of file I/O operations on simulation performance. In addition, the corresponding test cases were based on the same model partitioning scheme, which divides a cell space into horizontal rectangles (or rows) as evenly as possible across the participating cluster nodes.

Table 9 shows a list of 14 performance metrics (organized into 5 categories) collected in the experiments through extensive instrumentation and measurement. Among them, the

total execution time (T) and the *maximum memory consumption* (MEM) are the two primary metrics that indicate the overall simulation performance.

Table 9. Performance Metrics Defined for the LTW Protocol

Category	Metrics	Description
Overall	**T**	Total Execution time (s)
	MEM	Maximum memory consumption (MB)
Event set	**PEE**	Number of events executed in persistent input queue
	VEE	Number of events executed in volatile input queue
	PQLen	Average length of persistent input queue
	VQLen	Average length of volatile input queue
Check-pointing	**SS**	Total number of states saved in state queues
	OPT-SK	Number of states reduced by the enhanced risk-free strategy
Fossil collection	**FCT**	Average time spent on a single fossil collection (ms)
Rollback	**PriRB**	Number of primary rollbacks
	SecRB	Number of secondary rollbacks
	RB	Total number of rollbacks (i.e., PriRB + SecRB)
	EI	Number of events imploded in persistent input queue
	ER	Number of events unprocessed in persistent input queue

To demonstrate the absolute performance of both protocols, the benchmark models were also executed using a sequential CD++ simulator on a single cluster node, with the corresponding metrics (denoted as T_{seq} and MEM_{seq}) collected in the experiments. Using the *sequential* simulation as the baseline case, the *overall speedup* of a parallel simulation on N cluster nodes is thus defined as follows.

$$\text{Overall Speedup} = \frac{T_{seq}}{T(N)}, \text{ where } N>1 \quad (1)$$

The other metrics in Table 9 provide additional insight into the impact of the LTW algorithms on the optimistic parallel simulation, allowing for an objective assessment of the effectiveness of the proposed synchronization protocol.

The experimental results for each test case were averaged over 20 independent simulation runs[7]. Besides, for those test cases executed on multiple cluster nodes, the results were also averaged over the participating nodes to obtain a *per-node* evaluation. The lengths

[7] A more comprehensive performance evaluation with thorough sensitivity analysis is beyond the scope of this dissertation and can be addressed in future research.

of the event queues (i.e., PQLen and VQLen) were averaged over samples collected every 20 event insertions in the queues.

6.2.2. Configuration and Metrics for the MADS Technique

The performance of the MADS technique was analyzed on an IBM BladeCenter QS22 server [IBM10b] with two 3.2 GHz IBM PowerXCell 8i processors and 32 GB main memory. Note that the Cell processors included in a QS22 server are interconnected through a Rambus FlexIO interface bus using the fully coherent **Broadband Interface (BIF)** protocol, thus allowing a Cell application to scale across the two Cell processors (with 2 PPEs and 16 SPEs in total) in a transparent manner, even though the inter-processor communication via FlexIO has a relatively lower bandwidth and higher latency than the intra-processor on-chip communication via EIB [IBM09].

The CD++/Cell simulator was implemented on Red Hat Enterprise Linux 5.2 using the IBM SDK for Multicore Acceleration 3.1 [IBM10a]. The *Fire1* and the *Watershed* models of varied sizes were used to evaluate the impact of the FSK and SEK algorithms respectively.

The major performance metrics are the *total execution time* (T) of the overall simulation and the *turn-around time* (T) of each synchronization function in the case of the FSK. Another metric, referred to as *scale-up*, is defined as follows to measure how the execution performance scales as a function of the number of SPEs involved in a computation.

$$\text{Scale-up} = \frac{\text{T(PPE with one SPE)}}{\text{T(PPE with N SPEs)}}, \text{ where N>1} \qquad (2)$$

Since the PPE differs from the SPEs in many striking features, the scale-up definition given in (2) uses the total execution time (or turn-around time) attained with the CD++/Cell simulator on ***the PPE with one SPE*** (instead of on the PPE alone) as the baseline case, leading to a more conservative estimate than what would be obtained from the traditional definition of speedup (which is based on a purely sequential execution) because the baseline case has already exploited a certain degree of parallelism on the Cell processor (e.g., data-streaming parallelism and SIMD vector parallelism).

When the SEK is executed on multiple SPEs, the SEK jobs were scheduled using the round-robin policy. For the FSK, the size of each TA block was set to 16KB in all of the test cases. Again, event-logging was turned off in the experiments, and the performance analysis

was based on experimental results averaged over 20 independent simulation runs to strike a balance between data reliability and testing effort.

6.3. Evaluation of the LTW Protocol

This section analyzes the performance of the LTW protocol using the benchmark models introduced in Section 6.1. These models have been verified prior to the actual performance testing to ensure that the parallel simulations generate the same results as in the corresponding sequential simulations.

Several notations are used in the following tables: a "×" mark indicates a failed test case due to memory exhaustion, while a shaded table entry attributes the poor performance to severe memory swapping activities. A "—" mark stands for a case excluded in the testing because either the performance trend is already clear in the series, or the model cannot be divided further based on the given partitioning scheme. The best execution time obtained in each series is also highlighted.

Table 10 gives the total execution time (T) and the maximum memory consumption (MEM) obtained in the simulations of the *Fire1* model with varied sizes on different numbers of cluster nodes. It is clear that the LTW protocol outperforms its TW counterpart in all of the successful test cases. First, the maximum memory consumption is decreased by 45% up to 92% on each node, making it possible to execute larger models on fewer cluster nodes and reducing the simulation cost considerably. Secondly, the total execution time is decreased by 24% up to 60% among those test cases with sufficient memory, and this outstanding improvement in execution time is achieved with a much smaller memory footprint at the same time.

Figure 50 shows the overall simulation speedups achieved in those test cases with sufficient memory, demonstrating that the LTW protocol attains better and more consistent simulation performance than the TW protocol. Note that, in some cases (e.g., 50×50 *Fire1* on 2 and 4 nodes), the performance of the TW simulation is even worse than that of the sequential execution (with a speedup of less than 1), mainly because the excessive communication and operational overhead incurred in the optimistic parallel simulation. Such scenarios, however, do not arise in the LTW cases tested in the experiments.

Table 10. Total Execution Time and Maximum Memory Consumption for *Fire1*

Size	Seq.	Prot.	Metric	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28
50×50	5.54 (T) 29.11 (MEM)	**TW**	**T**	×	9.08	5.87	5.26	***5.01***	5.39	5.49	5.55	5.95	—	—	—	—	—	—
			MEM	×	813.57	220.42	109.94	61.79	43.73	34.84	26.37	22.22	—	—	—	—	—	—
		LTW	**T**	5.78	3.61	3.02	2.98	***2.78***	3.01	3.23	3.25	3.54	—	—	—	—	—	—
			MEM	63.53	65.83	27.42	20.58	14.25	13.24	11.98	9.95	9.31	—	—	—	—	—	—
100×100	56.07 (T) 110.59 (MEM)	**TW**	**T**	×	×	2749.13	484.91	40.09	35.66	34.46	***32.35***	33.51	32.53	32.44	33.4	35.0	35.19	35.96
			MEM	×	×	2279.42	1492.31	882.82	576.61	410.19	307.79	244.6	197.97	162.92	137.77	121.47	103.03	91.75
		LTW	**T**	78.21	43.84	31.62	24.35	23.58	22.61	22.26	***21.62***	21.86	21.88	22.03	22.2	22.0	22.46	21.76
			MEM	405.5	373.25	271.62	160.26	110.94	82.65	66.75	55.65	48.18	43.55	38.92	36.22	34.05	32.3	29.94
150×150	260.65 (T) 242.69 (MEM)	**TW**	**T**	×	×	×	×	×	1516.48	893.43	572.83	314.03	202.71	141.46	***140.98***	142.63	142.01	143.18
			MEM	×	×	×	×	×	2309.12	1935.02	1449.83	1131.65	906.07	744.91	623.9	527.05	460.76	404.44
		LTW	**T**	1489.77	517.92	394.56	122.44	112.93	110.63	111.7	109.67	107.02	107.23	105.27	107.1	106.75	104.88	***104.74***
			MEM	1418.85	1294.08	986.62	660.31	415.01	296.96	230.4	186.68	161.7	137.22	123.85	105.07	96.8	90.88	85.09
200×200	815.43 (T) 432.13 (MEM)	**TW**	**T**	×	×	×	×	×	×	×	×	×	×	4324.31	1236.26	1065.79	881.61	***737.14***
			MEM	×	×	×	×	×	×	×	×	×	×	1848.93	1560.7	1528.73	1188.06	1058.7
		LTW	**T**	12571.7	6894.36	1425.16	920.86	646.56	350.58	334.77	331.2	333.12	326.7	327.56	327.46	***322.93***	330.03	327.24
			MEM	1679.36	1644.54	1393.66	1229.82	1145.6	805.17	582.49	431.18	393.15	291.49	244.01	209.52	235.47	186.1	188.68

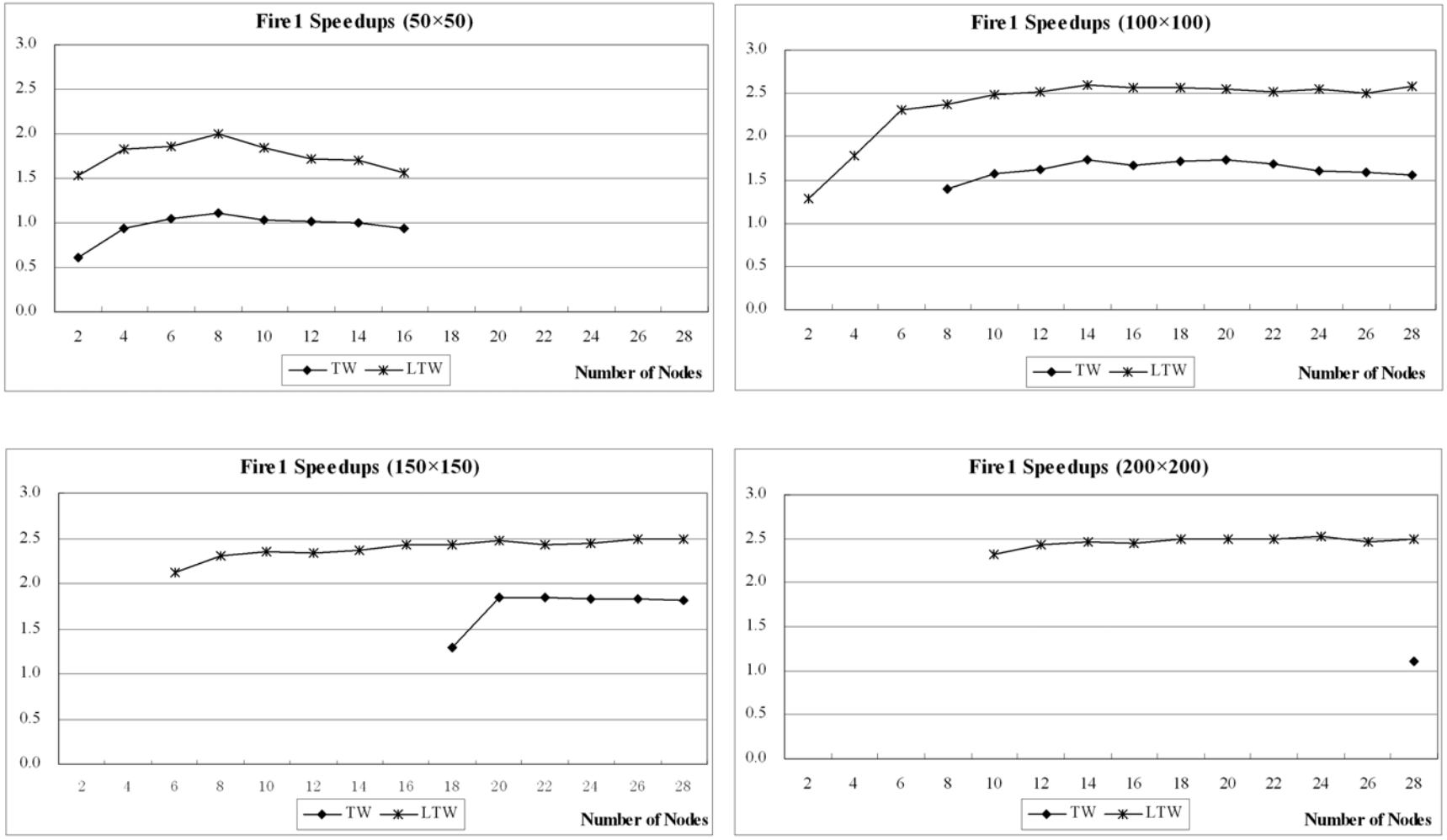


Figure 50. *Fire1* Overall Speedups (Test Cases with Sufficient Memory)

In order to figure out the underlying reasons that cause the differences in simulation performance, the other metrics are compared. Using the test case of 100×100 *Fire1* on 14 nodes, as an example, Table 11 presents a comparison of the synchronization protocols.

Table 11. Comparison for 100×100 *Fire1* on 14 Nodes

Metrics	TW	LTW	LTW vs. TW
PEE	96685.07	10597.71	
VEE	0	67214.07	
PQLen	24798.12	2636.95	↓ **89.37%**
VQLen	0	121.89	
SS	52819.64	22675.14	↓ **57.07%**
OPT-SK	0	18445.36	
FCT	488.14	84.15	↓ **82.76%**
PriRB	613.14	604.00	↓ **1.49%**
SecRB	11922.07	981.14	↓ **91.77%**
RB	12535.21	1585.14	↓ **87.35%**
EI	61751.93	5826.36	↓ **90.56%**
ER	48118.79	5790.93	↓ **87.97%**

Thanks to the introduction of the volatile input queue, the average length of the persistent input queue is shortened significantly by 89.37% in the LTW simulation, reducing the overhead of event queue operations and memory consumption considerably. On the other hand, the volatile input queue is kept short throughout the simulation with an average length of just 121.89 events, despite the fact that a majority of 86.38% input events executed on each node have been turned into volatile under the LTW protocol.

Owning to the enhanced risk-free infrequent state-saving strategy, which further reduces the number of state-saving by 44.86% on top of the MTSS strategy, the total number of states saved in the LTW simulation is 57.07% fewer than in the TW simulation, leading to less memory usage as well.

As expected, the time spent on a fossil collection operation is decreased from 488.14 ms to just 84.15 ms accordingly, a significant reduction of 82.76%.

When comparing the rollback performance, the LTW protocol also demonstrates a big advantage over the TW counterpart. The number of secondary rollbacks is reduced by 91.77%, showing that rollback propagation is effectively contained within the TW domain on each node. Moreover, the number of primary rollbacks is reduced slightly by 1.49%, which, combined with the fact that the total number of events executed on each node (i.e., PPE +

VEE) is decreased by 19.52%, suggests a more stable system with less speculative computation. As a consequence, the numbers of events imploded and unprocessed in the persistent input queue are both declined by roughly 90%, further accelerating rollback operations in the simulation.

The experimental results for the *Fire2* and the *Watershed* models are given in Table 12 and Table 13 respectively. Again, the LTW protocol reduces maximum memory consumption by approximately 34% up to 92% in the *Fire2* simulations and by 73% up to 93% in the *Watershed* simulations. The reduction in memory usage is more prominent for the *Watershed* model largely because, with a higher proportion of simultaneous events exchanged between the LPs at each virtual time, a larger percentage of states are reduced with the enhanced risk-free infrequent state-saving strategy.

For those test cases with sufficient memory, the total execution time is decreased by 13% up to 32% in the *Fire2* simulations and by 5% up to 91% in the *Watershed* simulations. A general trend reflected in the experimental results is that the reduction in the total execution time and maximum memory consumption is greater for models with larger sizes, indicating an improved scalability of the synchronization algorithms.

The overall simulation speedups achieved in the *Fire2* and the *Watershed* simulations with sufficient memory are shown in Figure 51and Figure 52 respectively. The impact of the LTW protocol is most evident in the *Watershed* simulations, where the number of simultaneous events executed at each virtual time grows with the model sizes. As model size increases, the parallel simulation under the TW protocol suffers from an overwhelming increase in both state-saving overhead and rollback cost, resulting in a performance degradation that essentially nullifies the effectiveness of the TW synchronization protocol, even when given adequate memory (e.g., in all but the smallest *Watershed* cases). In contrast, significant speedups are achieved in the LTW cases, demonstrating that the LTW protocol is especially adept at efficient execution of simultaneous events in optimistic parallel simulations.

Table 12. Total Execution Time and Maximum Memory Consumption for *Fire2*

Size	Seq.	Prot.	Metric	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28
50×50	19.29 (T) 29.52 (MEM)	**TW**	**T**	×	20.89	13.93	12.19	10.91	***10.41***	10.8	10.64	10.84	10.55	11.31	12.51	12.76	13.39	13.44
			MEM	×	800.26	226.82	108.41	65.37	46.29	34.54	28.13	23.23	20.19	18.19	16.31	14.81	13.72	12.85
		LTW	**T**	20.26	14.23	10.38	9.69	9.46	8.84	9.01	8.51	8.64	8.4	***8.32***	9.28	9.34	9.51	10.27
			MEM	81.24	66.92	34.99	22.6	17.77	14.65	13.02	11.76	10.83	10.17	9.63	9.29	8.99	8.73	8.49
100×100	119.95 (T) 109.57 (MEM)	**TW**	**T**	×	×	3284.37	460.32	68.67	54.63	52.03	48.92	48.58	46.96	***46.37***	47.53	48.69	49.39	49.97
			MEM	×	×	2159.1	1319.08	658.14	576.72	411.14	310.95	240.42	198.4	163.47	149.65	112.23	99.94	83.78
		LTW	**T**	206.16	114.98	60.09	54.37	51.22	44.11	41.61	40.37	38.87	37.55	***35.54***	36.83	36.23	36.46	36.48
			MEM	314.37	285.18	248.32	137.73	102.24	81.63	65.57	54.35	48.91	45.62	42.6	38.42	35.75	33.84	32.03
150×150	414.25 (T) 243.71 (MEM)	**TW**	**T**	×	×	×	×	×	4448.08	2487.95	651.06	394.92	244.97	167.25	***164.79***	167.42	165.64	168.88
			MEM	×	×	×	×	×	1817.71	1375.23	1399.3	1086.72	905.96	744.91	562.55	532.14	425.91	399.51
		LTW	**T**	1592.43	493.61	223.65	178.2	174.63	165.84	168.66	167.14	140.67	140.21	137.0	134.3	136.11	***133.1***	134.01
			MEM	1210.37	924.16	641.79	586.92	385.41	269.62	205.4	172.18	139.44	122.16	112.93	104.22	94.47	89.39	85.79
200×200	1033.61 (T) 424.96 (MEM)	**TW**	**T**	×	×	×	×	×	×	×	×	×	12112.7	3206.02	1501.28	1202.48	900.05	***764.21***
			MEM	×	×	×	×	×	×	×	×	×	1943.55	1785.9	1618.94	1522.69	1475.58	1243.95
		LTW	**T**	11707.5	3363.07	1339.92	1173.69	562.68	414.52	412.92	412.89	381.1	376.58	417.44	373.11	372.6	***370.04***	371.56
			MEM	1661.95	1562.62	1267.71	1292.97	885.61	438.81	363.5	313.96	289.68	274.55	240.98	227.23	208.62	192.61	173.08

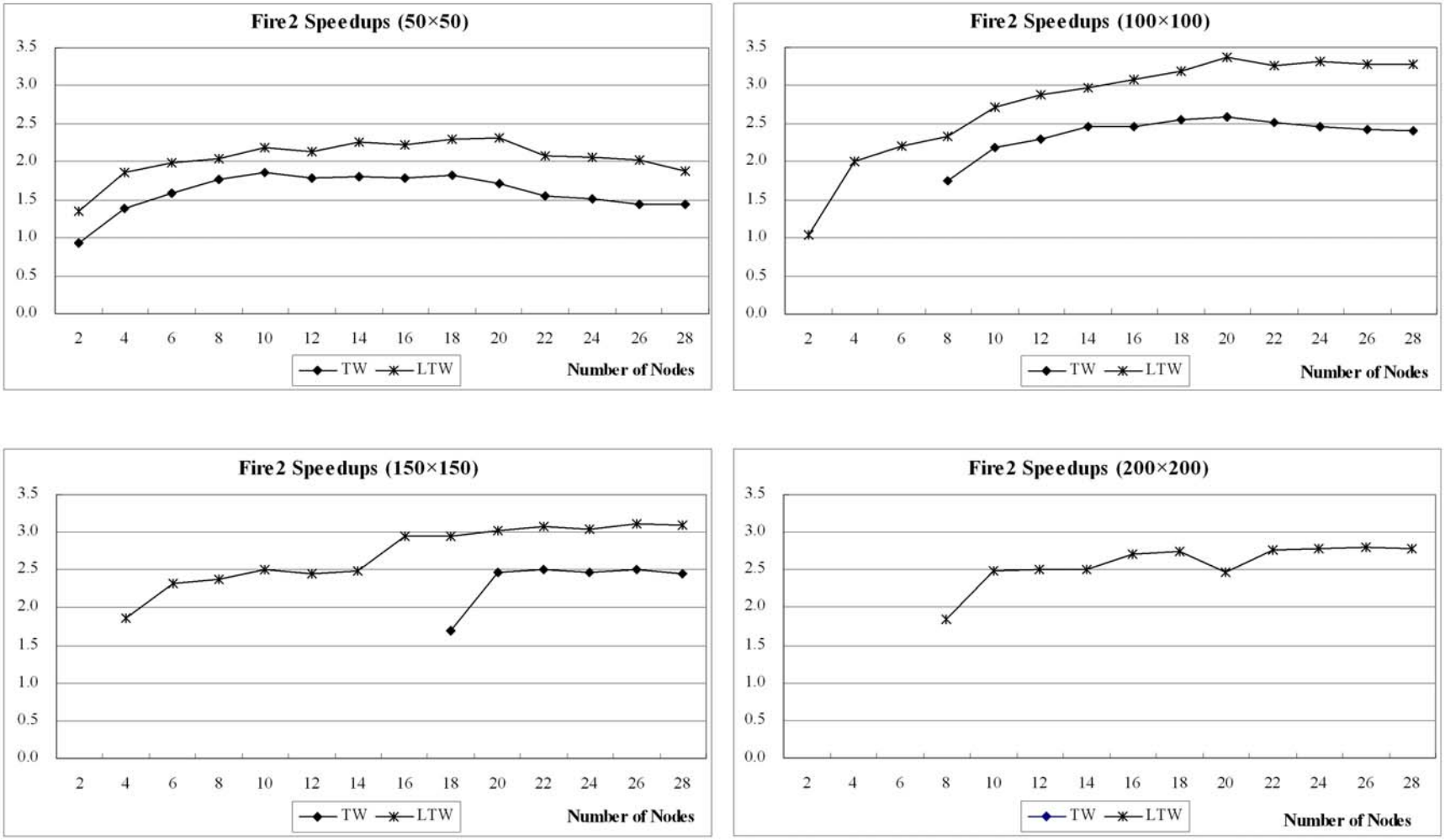


Figure 51. *Fire2* Overall Speedups (Test Cases with Sufficient Memory)

Table 13. Total Execution Time and Maximum Memory Consumption for *Watershed*

Size	Seq.	Prot.	Metric	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28
15×15×2	258.27 (T) 43.99 (MEM)	TW	T	×	×	2059.62	899.49	***84.97***	87.06	86.59	88.76	—	—	—	—	—	—	—
			MEM	×	×	1718.02	997.21	691.2	536.37	422.49	333.53	—	—	—	—	—	—	—
		LTW	T	262.99	171.18	112.69	100.54	***79.45***	82.27	82.08	82.59	—	—	—	—	—	—	—
			MEM	45.66	27.91	148.48	121.54	128.96	113.14	101.29	90.39	—	—	—	—	—	—	—
20×20×2	471.86 (T) 72.67 (MEM)	TW	T	×	×	×	×	2451.7	857.3	757.65	724.55	***638.97***	676.42	—	—	—	—	—
			MEM	×	×	×	×	1618.94	1180.67	967.51	778.53	643.52	535.21	—	—	—	—	—
		LTW	T	473.81	268.87	181.94	155.09	140.14	***104.77***	108.52	109.58	110.35	112.87	—	—	—	—	—
			MEM	76.02	40.04	164.35	136.36	130.82	149.81	137.24	129.85	115.87	111.99	—	—	—	—	—
25×25×2	735.39 (T) 115.48 (MEM)	TW	T	×	×	×	×	×	×	×	2002.73	1948.95	1922.21	1705.19	1597.08	***1585.6***	—	—
			MEM	×	×	×	×	×	×	×	1519.54	1434.77	1262.59	1063.03	774.38	663.21	—	—
		LTW	T	748.49	469.65	306.25	257.18	195.16	176.19	172.39	***136.18***	136.37	142.69	143.86	139.54	141.85	—	—
			MEM	119.8	70.46	164.86	128.68	131.07	132.81	132.27	153.82	141.87	128.25	114.39	113.95	103.44	—	—
30×30×2	1041.39 (T) 168.46 (MEM)	TW	T	×	×	×	×	×	×	×	×	5381.55	4475.37	3133.72	3130.89	2920.06	***2765.2***	2784.83
			MEM	×	×	×	×	×	×	×	×	2192.96	1867.83	1602.25	1388.87	1206.87	1055.31	924.49
		LTW	T	1098.11	616.28	390.68	293.33	237.82	208.26	204.82	198.27	169.12	168.45	168.01	165.54	165.64	166.55	***162.43***
			MEM	174.08	89.69	163.07	164.18	151.55	171.62	148.91	138.31	117.57	139.45	156.5	149.91	130.2	122.69	114.69

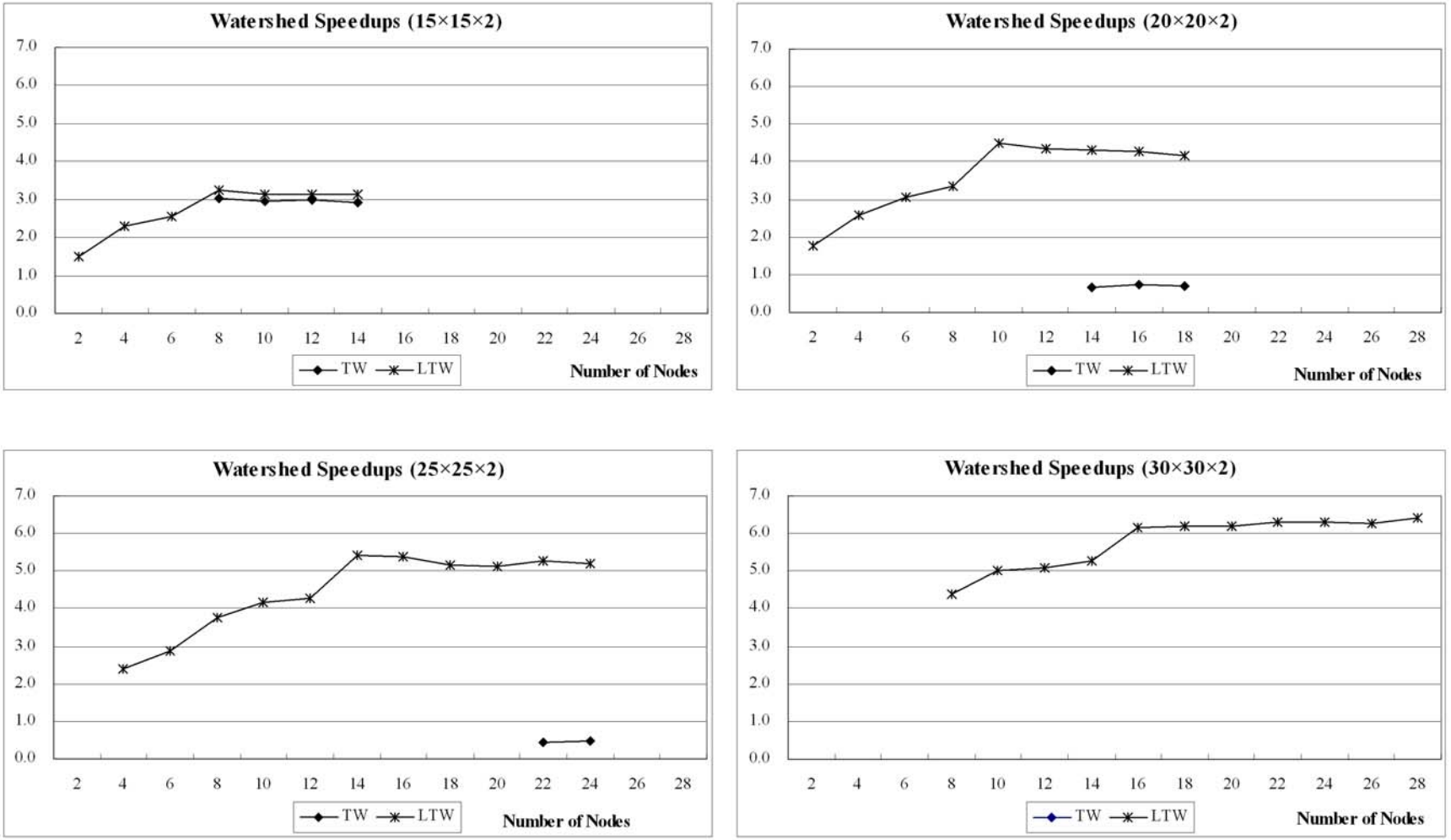


Figure 52. *Watershed* Overall Speedups (Test Cases with Sufficient Memory)

Table 14 and Table 15 compare the other metrics measured in the 100×100 *Fire2* simulation on 20 nodes and the 20×20×2 *Watershed* simulation on 18 nodes respectively. From these comparisons, a similar pattern can be observed in terms of performance improvement, suggesting that the LTW protocol is suitable for simulating models with different computation and communication characteristics.

Table 14. Comparison for 100×100 *Fire2* on 20 Nodes

Metrics	TW	LTW	LTW vs. TW
PEE	68346.55	11658.75	
VEE	0	56057.00	
PQLen	17533.37	2149.91	↓ **87.74%**
VQLen	0	75.31	
SS	33833.00	17565.40	↓ **48.08%**
OPT-SK	0	15591.10	
FCT	245.12	58.36	↓ **76.19%**
PriRB	769.95	740.55	↓ **3.82%**
SecRB	12794.35	2036.45	↓ **84.08%**
RB	13564.30	2777.00	↓ **79.53%**
EI	46877.55	7197.90	↓ **84.65%**
ER	29512.45	6651.60	↓ **77.46%**

Table 15. Comparison for 20×20×2 *Watershed* on 18 Nodes

Metrics	TW	LTW	LTW vs. TW
PEE	1253641.94	361457.78	
VEE	0	856256.00	
PQLen	334016.67	77790.62	↓ **76.71%**
VQLen	0	26.04	
SS	371273.33	73186.94	↓ **80.29%**
OPT-SK	0	288247.50	
FCT	61313.67	395.63	↓ **99.35%**
PriRB	173.50	159.94	↓ **7.81%**
SecRB	22816.67	2165.33	↓ **90.51%**
RB	22990.17	2325.28	↓ **89.89%**
EI	625210.33	175521.11	↓ **71.93%**
ER	569337.94	172280.33	↓ **69.74%**

In summary, the experimental results indicate that the LTW protocol outperforms the TW counterpart in various aspects, including shortened execution time, reduced memory consumption, lowered operational cost, accelerated event queue operations, and enhanced system stability and scalability.

6.4. Evaluation of the MADS Technique

This section evaluates the performance impact of the MADS technique using the *Fire1* and the *Watershed* models introduced in Section 6.1. Based on the workload analysis presented in Section 5.2, the *Fire1* model is used to examine the performance of the FSK algorithms, while the *Watershed* model is used to gauge the performance of the SEK algorithms. The correctness of the parallel simulations has been verified before the actual performance testing.

6.4.1. Performance of the FSK algorithms

Using the 1024×1024 *Fire1* model previously analyzed in Chapter 5, Figure 53 summarizes the total execution times (T) attained on the PPE with the sequential CD++/PPE simulator and on 1 to 16 SPEs with the parallel CD++/Cell simulator. In the diagram, the PPE-based sequential executions are denoted as **ORG** (refer to Table 1 of Section 5.2.1), **SYN** (refer to Table 7 of Section 5.3.1), and **FLT**.

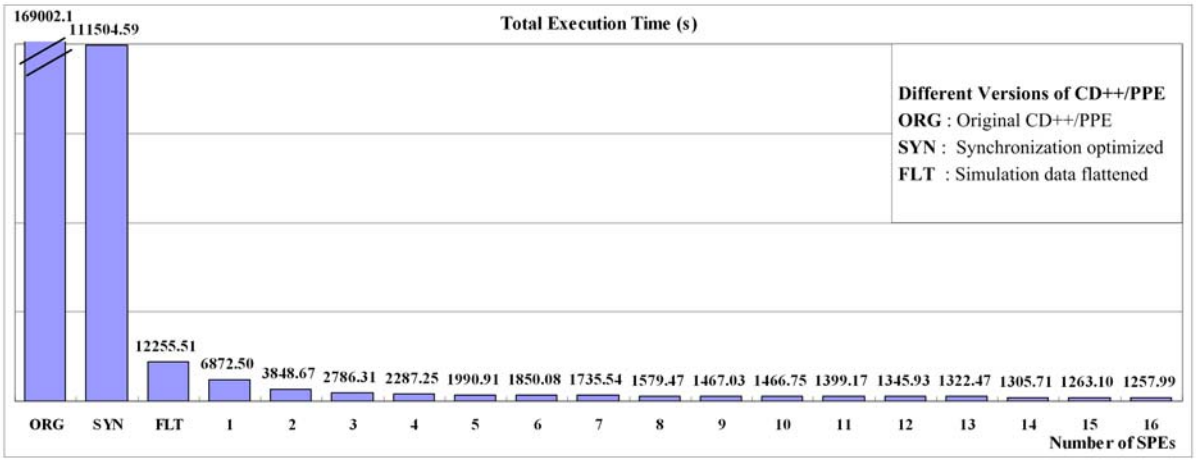


Figure 53. Total Execution Time in the 1024×1024 *Fire1* Simulation

When the simulation data are reorganized in flat arrays, the total execution time (**FLT**) is reduced by a factor of 13.79 from what is attained with the original CD++/PPE (**ORG**) and by a factor of 9.09 from the synchronization-optimized execution (**SYN**). As the performance of the *Fire1* simulation is largely determined by the FSK that has a *regular access pattern* when processing the Simulator timing data *contiguously* in the main memory, the increased data locality greatly improves the utilization of the cache hierarchy of the PPE, resulting in a significant reduction in memory contention and execution time.

To illustrate this point, Table 16 gives the simulation profile of the PPE-based **FLT** execution. Comparing Table 16 with Table 7, it is evident that the biggest improvement comes from the FC, where the time spent on processing (@) and (D) events is reduced by more than 89% from the **SYN** execution. In addition, the simulation bootstrap time is decreased by 50% since most of the simulation data are allocated and initialized in batches with big arrays. The use of the event buffers also accelerates Simulator event execution and FEL operations (Other overhead) by 7.1% and 23.81% respectively.

Table 16. 1024×1024 *Fire1* Simulation Profile on PPE (Data Flattened)

Event Type	Components				
	Simulators	**FC**	**NC**	**Bootstrap**	**Other Overhead**
(I)	2.58	0.76	—	—	—
(*)	491.68	12.60	—		
(@)	6.24	5650.61	—		
(X)	—	0	—		
(Y)	—	76.41	—		
(D)	—	5821.37	1.62		
Sum (s)	**500.51**	**11561.75**	**1.62**	**89.10**	**102.53**
Total (s)	**12255.51**				

With the introduction of one SPE, the parallel simulation runs 1.78 times faster than the sequential **FLT** execution, mainly because of more efficient data processing algorithms and the exploitation of data-streaming parallelism on the SPE. Overall, the parallel CD++/Cell simulator further reduces the total execution time from the best sequential performance of over 3 hours on the PPE (**FLT**) to approximately 20 minutes when the FSK is parallelized on 16 SPEs (or a factor of up to 9.74). Together, the FSK optimization and parallelization algorithms accelerate the 1024×1024 *Fire1* simulation on a QS22 server by a large factor of up to 134.34 over the original CD++/PPE implementation.

Figure 54 shows the scale-ups of the FSK itself based on the *turn-around time*s measured for the individual synchronization functions. Both synchronization functions exhibit a significant level of scalability (super-linear in the case of `findImminents`), due to SIMD code vectorization on the SPEs and reduced memory latency with double-buffered DMA transfer. Function `findImminents` performs better than function `findMinTime` because of the reasons that have been explained in Section 5.3.4. First, all of the SPE threads are engaged in the `findMinTime` computation, whereas an SPE thread is involved in the

`findImminents` computation only if it has found the global minimum. Secondly, function `findMinTime` is called in place by the FC, while function `findImminents` is called in advance by the NC once the next simulation time is determined. As a whole, the FSK attains an overall scale-up of 13.4 on 16 SPEs.

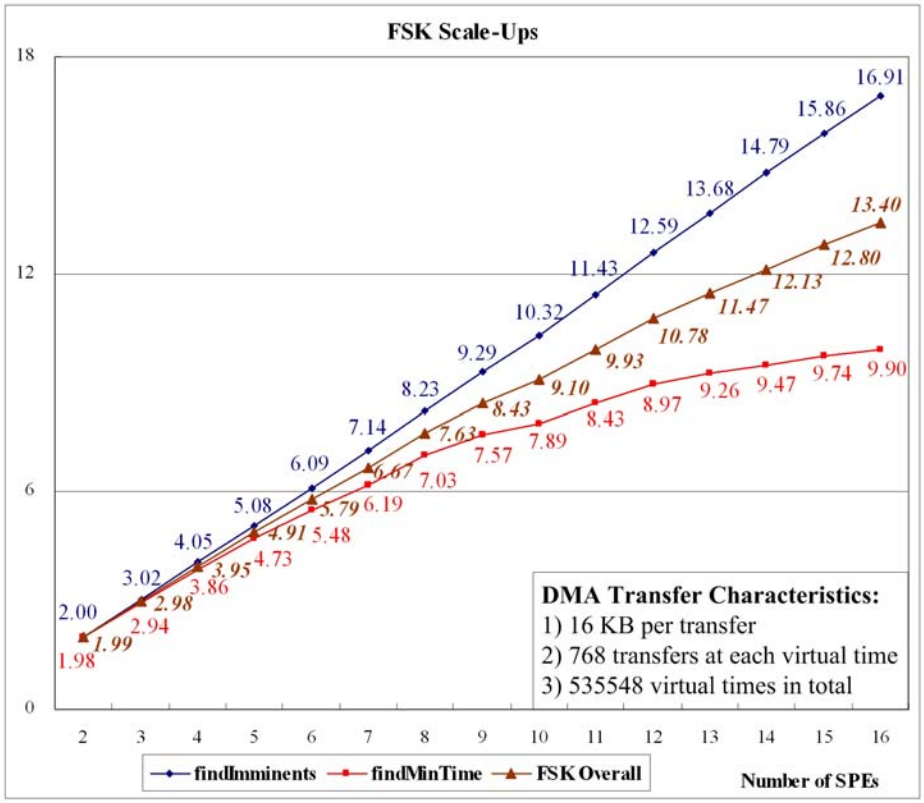


Figure 54. FSK Scale-Ups in the 1024×1024 *Fire1* Simulation

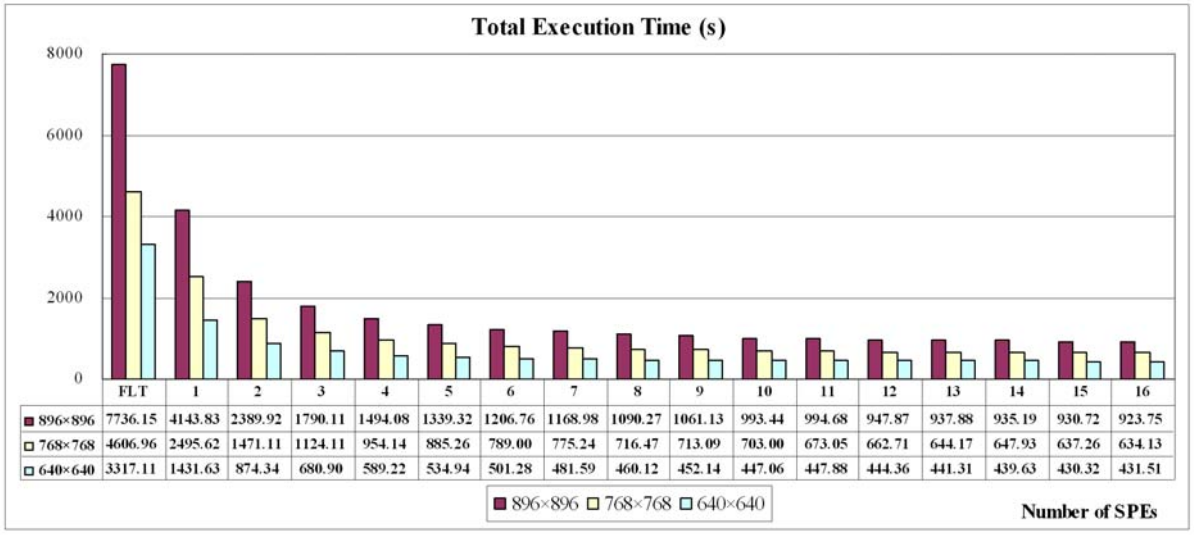


Figure 55. Total Execution Time in *Fire1* Simulations with Varied Sizes

Figure 55 presents the execution times attained in the *Fire1* simulations of different sizes (896×896, 768×768, and 640×640) with the optimized CD++/PPE and with the CD++/Cell on up to 16 SPEs. The parallel simulations with 16 SPEs run 8.37, 7.27, and 7.69 times faster than the best sequential executions on the PPE (**FLT)** in the 896×896, 768×768, and 640×640 *Fire1* simulations respectively.

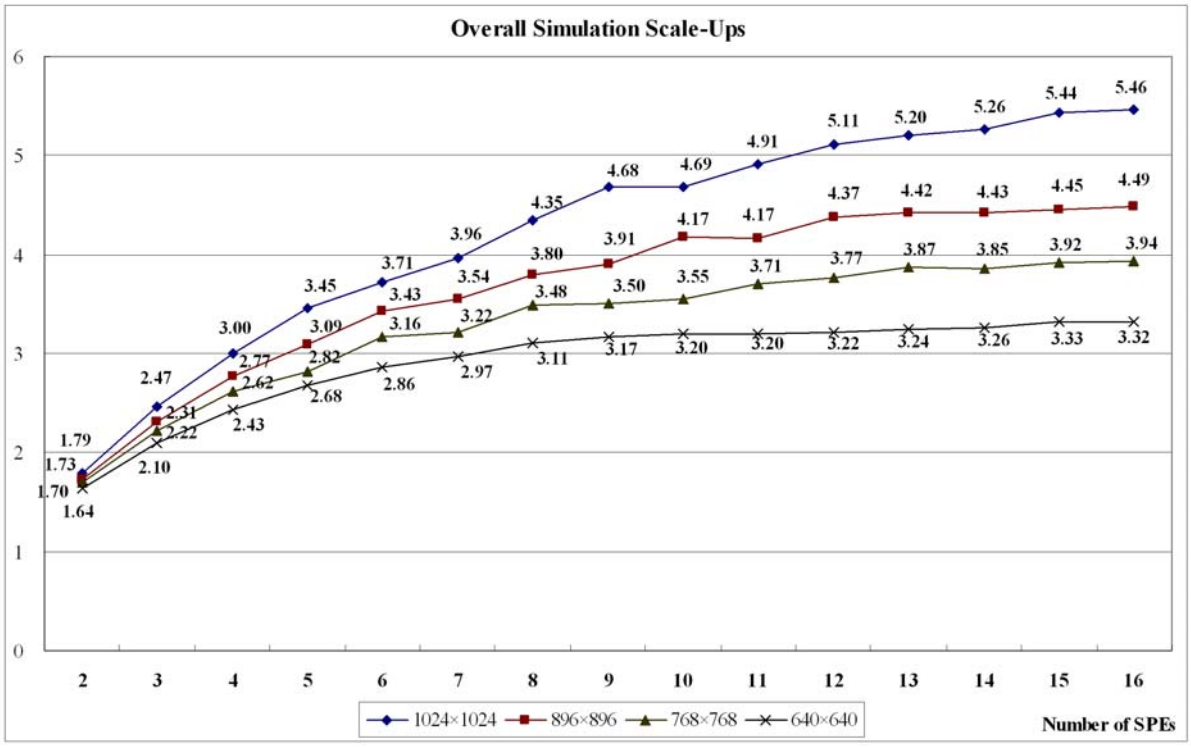


Figure 56. Overall Simulation Scale-Ups in *Fire1* Simulations

Figure 56 shows the overall simulation scale-ups achieved in the *Fire1* simulations of varied sizes on 2 to 16 SPEs with the parallel CD++/Cell simulator. The experimental results indicate that the FSK algorithms can obtain better scalability in larger simulations and on a greater number of SPEs. Since the FSK is a data-intensive, high-throughput kernel with a relatively light computational intensity, the overall simulation performance depends primarily on the effective bandwidth provided by the EIB and MIC of a Cell processor when transferring data to/from the main memory. Hence, as long as the memory path has not yet been saturated, more DMA transfer requests from the SPEs can be handled concurrently with improved utilization of the memory bandwidth, resulting in a higher level of scalability in larger simulations using multiple SPE cores.

6.4.2. Performance of the SEK algorithms

The 320×320×2 *Watershed* model, previously analyzed in Chapter 5, has been executed on the PPE and across the SPEs of a QS22 server to evaluate the performance of the SEK algorithms. The resulting total execution times (T) are summarized in Figure 57, where the sequential executions with the original and the optimized CD++/PPE simulator are denoted respectively as **ORG** (refer to Table 4 of Section 5.2.2) and **FLT** (which combines flattened simulation data management with optimized FC synchronization task). The synchronization-optimized sequential execution (**SYN**) is not shown separately in the figure because, as mentioned in Section 5.3.1, optimizing the FC synchronization task alone does not have a noticeable impact on the *Watershed* simulation performance.

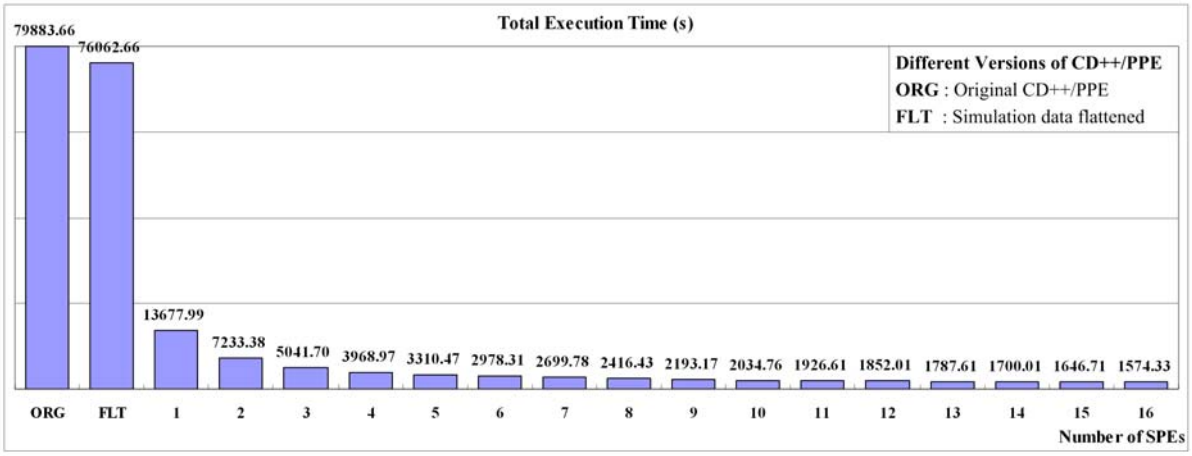


Figure 57. Total Execution Time in the 320×320×2 *Watershed* Simulation

Unlike the 1024×1024 *Fire1* model, flattening the simulation data in the 320×320×2 *Watershed* model leads to only a marginal improvement of 4.78% in terms of total execution time (i.e., **FLT** vs. **ORG**), due to the following reasons. First, the performance of the *Watershed* simulation is dominated by the *compute-intensive* SEK, rather than the *data-intensive* FSK as in the *Fire1* simulation. Therefore, the performance of the *Watershed* simulation is much less sensitive to the improved data locality. Secondly, at any virtual time, the event and state data of *different* active Simulators are stored in *different* event and state buffer entries, which may *not* be contiguous in the main memory. As a result, while the FSK can fully benefit from the enhanced locality of Simulator timing data, the SEK may exploit increased data locality only during the processing of multiple input/output events for a *single* active Simulator as these events are packed together in the *same* event buffer entry.

Given in Table 17, the updated **FLT** execution profile illustrates the quantitative impact of flattening simulation data on the *Watershed* simulation performance. When compared to the **ORG** execution, the Simulator and FC event execution time is reduced by 3.92% and 37.91% respectively, thanks to improved data locality in the event buffers. Moreover, the time spent on FEL operations (Other overhead) is decreased by 70.12% since all of the simultaneous events, which constitute a vast majority of the total event population, are executed without using the FEL. Besides, the simulation bootstrap time is declined by 16.61% due to the batch allocation and initialization of simulation data in flat arrays.

Table 17. 320×320×2 *Watershed* Simulation Profile on PPE (Data Flattened)

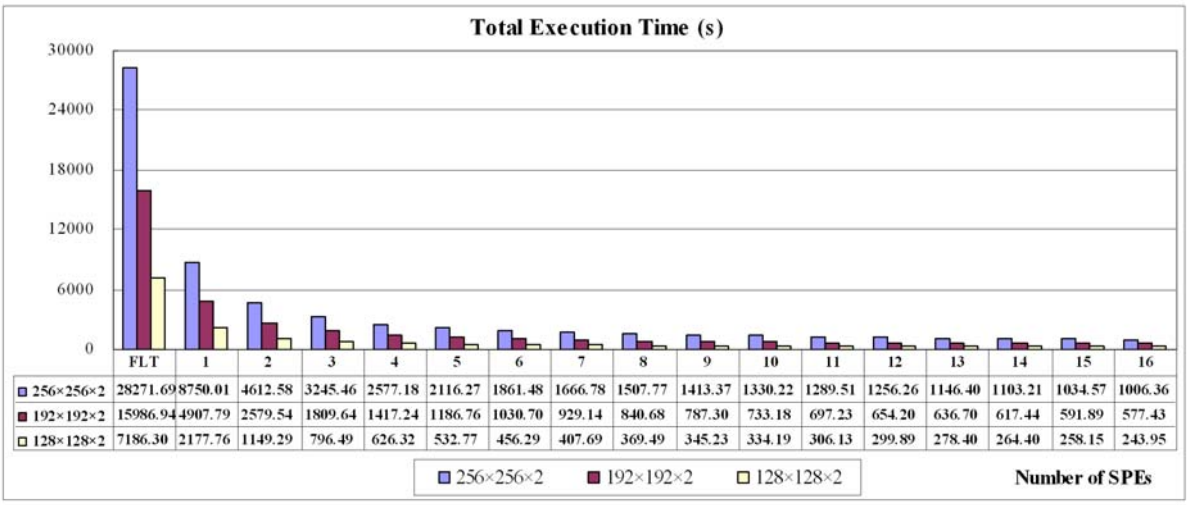
Event Type	Components				
	Simulators	**FC**	**NC**	**Bootstrap**	**Other Overhead**
(I)	0.36	0.03	—	—	—
(*)	75198.80	15.42	—		
(@)	32.64	37.27	—		
(X)	—	0	—		
(Y)	—	596.99	—		
(D)	—	14.26	0.001		
Sum (s)	**75231.80**	**663.97**	**0.001**	**21.03**	**145.86**
Total (s)	**76062.66**				

When the simulation is executed with one SPE, the total execution time is shortened dramatically by a factor of 5.56 over the best sequential execution (**FLT**). There are several reasons for this exceptional performance gain. First, memory latency is minimized effectively because of the software-managed multi-layered double buffering strategy for DMA transfer of all kinds of simulation data (job IDs, events, states and rules) to and from the SPE. Secondly, using the PPE along with an SPE allows for pipelined event execution between the FC and the virtual Simulators, taking advantage of the event-streaming parallelism. Thirdly, the SEK is implemented in SIMD-aware C code on the SPE, making it more efficient than the object-oriented scalar C++ implementation on the PPE. In addition, the SEK computation is further boosted on the SPE with various low-level code optimizations, which include improving data access efficiency by proper LS address alignment, reducing runtime call stack usage by in-place rule evaluation and by replacing function parameters and local variables with aligned global variables and registers, removing

branches whenever possible or using branch hints explicitly, and enhancing performance by loop unrolling and in-line substitution.

Overall, the total execution time is reduced from the best sequential performance of more than 21 hours (**FLT**) to just 26 minutes when the SEK is parallelized on 16 SPEs (or a significant improvement by a factor of up to 48.31). Comparing to the original CD++/PPE execution (**ORG**), the SEK optimization and parallelization algorithms together accelerate the 320×320×2 *Watershed* simulation by a factor of up to 50.74 on a QS22 server.

Figure 58 shows the execution times obtained in the *Watershed* simulations of varied sizes (256×256×2, 192×192×2, and 128×128×2) with the optimized CD++/PPE and with the CD++/Cell on up to 16 SPEs. The parallel simulations with 16 SPEs run 28.09, 27.69, and 29.46 times faster than the PPE-based **FLT** executions respectively.



	FLT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
256×256×2	28271.69	8750.01	4612.58	3245.46	2577.18	2116.27	1861.48	1666.78	1507.77	1413.37	1330.22	1289.51	1256.26	1146.40	1103.21	1034.57	1006.36
192×192×2	15986.94	4907.79	2579.54	1809.64	1417.24	1186.76	1030.70	929.14	840.68	787.30	733.18	697.23	654.20	636.70	617.44	591.89	577.43
128×128×2	7186.30	2177.76	1149.29	796.49	626.32	532.77	456.29	407.69	369.49	345.23	334.19	306.13	299.89	278.40	264.40	258.15	243.95

Figure 58. Total Execution Time in *Watershed* Simulations with Varied Sizes

Figure 59 gives the overall simulation scale-ups attained in the *Watershed* simulations of varied sizes. The experimental results suggest that, regardless of the difference in the size of the models, the *Watershed* simulations exhibit similar scalability across the SPEs. Since the *Watershed* simulation is mainly dominated by the compute-intensive SEK hosted on the SPEs, the cumulative computing power of the available SPEs becomes a major limiting factor in the overall simulation performance. Therefore, as long as the SPEs are fully utilized, the total execution time is expected to improve at a similar rate when an increasing number of SPEs join the parallel simulation.

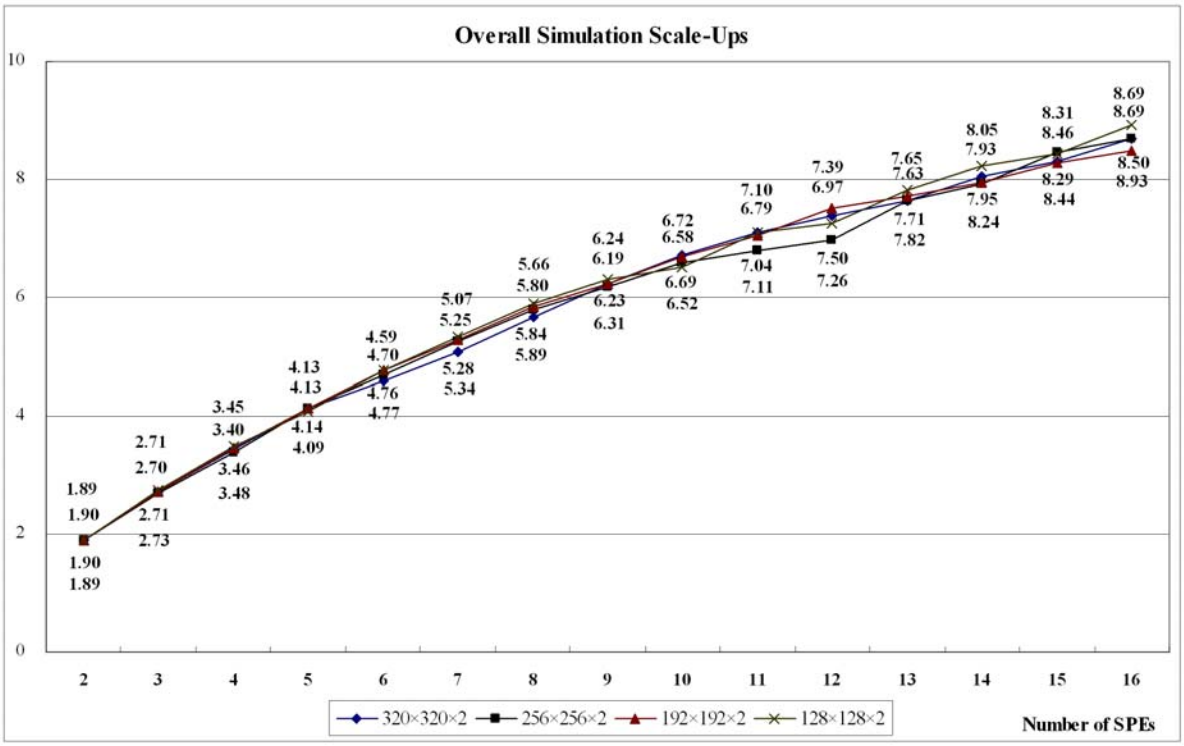


Figure 59. Overall Simulation Scale-Ups in *Watershed* Simulations

As illustrated in Figure 56 and Figure 59, the scale-ups grow a bit slower when more and more SPEs are used in the simulations, mainly because the overhead of kernel orchestration increases with the number of SPEs. In addition, when the number of SPEs goes beyond eight, the scaling of the simulation performance also suffers from reduced bandwidth and increased latency of inter-processor communication through the FlexIO interface bus, resulting in a change in the slope of the scale-ups.

In summary, the experiments demonstrate that the proposed MADS technique can be used to accelerate both memory-bound and compute-bound computational kernels in demanding parallel DEVS simulations on the heterogeneous Cell processor. In the *Fire1* and *Watershed* simulations, significant performance improvements have been obtained with the FSK and SEK algorithms, proving that porting DEVS-based simulators to heterogeneous multicore platforms such as the Cell processor is worth the effort.

Chapter 7. Conclusion and Future Work

This chapter summarizes the content of the dissertation, outlines the major contributions of the research, and proposes a list of open issues and avenues for future research. Section 7.1 recaps the primary objectives and areas of research covered in the dissertation. Section 7.2 reviews the key high-level contributions made in the research. Section 7.3 suggests several future research directions.

7.1. Summary of the Dissertation

This dissertation addressed software development and performance issues that arise in large-scale parallel simulation of P-DEVS and Cell-DEVS models. In particular, this dissertation was primarily concerned with improving the performance of DEVS-based TW simulation on distributed-memory multiprocessor clusters and achieving high-performance parallel DEVS simulation on heterogeneous CMP architectures as exemplified by the IBM Cell processor. To fulfill these two objectives, a **Lightweight Time Warp (LTW)** protocol and a **Multicore Acceleration of DEVS Systems (MADS)** technique have been proposed, and their effectiveness has been evaluated quantitatively in the CD++ environment using different benchmark models with varied characteristics. The rest of this section briefly summarizes the main concepts and qualitative results of these two research endeavors.

The LTW protocol exploits the intrinsic computational properties of the DEVS-based simulation process to improve TW execution efficiency, while decreasing memory consumption at the same time. This is achieved by classifying the LPs hosted on each node into three categories: *full-fledged TWLP*, *interface TWLP* and *lightweight LPs*, which correspond to the *NC*, the *FC*, and the *Simulators* in PCD++ respectively. Such an arrangement enables a purely optimistic TW simulation to be driven by only a few full-fledged TWLPs, whereas most of the processes are turned into lightweight LPs whose input and output events are no longer maintained in persistent queues and whose states are managed by the interface TWLP in a concentrated manner. Moreover, the MTSS strategy is enhanced to further reduce the number of states saved in the state queues without the need

for coasting forward during rollbacks. As a consequence, both forward execution and rollback recovery are accelerated; and rollback propagation is restricted to between the full-fledged and the interface TWLPs only, lowering the possibility of cascaded rollback and enhancing system stability and scalability. The simulation performance also benefits from facilitated queue operations, fossil collection, and GVT estimation. Due to the elimination of past input/output events and the reduction in the number of historical states, the lightweight LPs can be migrated between cluster nodes efficiently at decreased computation and communication cost. Besides, the LTW protocol can be readily integrated with many other TW optimization strategies to further improve simulation performance, and its applicability can be extended to other types of TW-based PDES systems under certain conditions and with an appropriate control of the LPs.

The MADS technique adopts a formalism-based performance-centric approach to general-purpose P-DEVS and Cell-DEVS simulation on the Cell processor. To this end, the workload characteristics of different types of models are generalized; and a variety of optimization and parallelization strategies are used to accelerate both *memory-bound* and *compute-bound* computational kernels commonly found in demanding parallel DEVS simulations. As these computational kernels directly reflect the major performance bottlenecks in the system, the proposed technique is more targeted than the traditional LP-oriented approach, making the achievable performance gain more deterministic and predictable. In addition to explicit exploitation of the inherent data and event parallelism in the simulation process, the MADS technique combines multi-grained parallelism at different levels of the system to leverage the full potential of the Cell processor, while hiding, to a great extent, the technical details of multicore programming from non-expert users. The various sorts of simulation data are reorganized in flat array-based buffers, improving simulation performance with increased data locality and facilitating the parallelization of the computational kernels from a *data-flow* perspective. By virtue of the concept of *LP virtualization*, the MADS technique allows for efficient mapping of an arbitrary number of Simulators to a limited set of SPE cores dynamically throughout a simulation, improving processor utilization, addressing the resource constraints of the underlying hardware architecture, and making it possible to achieve fine-grained dynamic load-balancing in a straightforward way. Furthermore, the MADS technique also provides a flexible software

architecture that can be easily extended to accommodate extra computational kernels, to support simulations with SPE-incompatible model components, and to integrate with other cluster-based conservative and optimistic PDES techniques.

The outcomes of the research would help bridge the gap between PDES algorithms developed for traditional multiprocessor clusters and those for emerging CMP architectures, paving the way towards achieving large-scale high-performance parallel simulation on hybrid supercomputers.

7.2. Review of Key Contributions

This section reviews the key high-level contributions made in the two primary areas of research, namely the LTW protocol and the MADS technique, highlighting how the various challenges discussed in Chapter 3 have been addressed in the dissertation.

7.2.1. Lightweight Time Warp Protocol

The LTW protocol takes a *proactive* approach to addressing the challenges of DEVS-based TW simulations, improving performance *without* complicating the synchronization protocol unnecessarily, sacrificing potential parallelism, or introducing a noticeable extra operational overhead. The key contributions of the LTW protocol are summarized as follows.

- Introduced a *high-level abstraction* that represents a DEVS-based simulation concisely as a well-structured multi-phased process, revealing the general execution pattern hidden behind the complex message flow among the LPs.
- Identified several *model-independent computational properties* of the underlying simulation process, providing the basis for developing new optimization and synchronization algorithms for DEVS-based TW simulations.
- Introduced the concept of *volatile events* in order to obviate the need for saving most of the input and output events during forward execution, reducing memory consumption, shortening the persistent input queue with accelerated queue operations, and alleviating the overhead of fossil collection.
- Developed an event management scheme that holds the *simultaneous events* exchanged between the interface TWLP and the lightweight LPs temporarily in a *volatile input queue*, which can be implemented using simple data structures with constant-time

queue operations, allowing for efficient execution of a large number of simultaneous events in DEVS-based TW simulations.

- Enhanced the *risk-free infrequent state-saving strategy* to further reduce memory consumption for saving past states and the cost of fossil collection, without increasing the overhead of rollback operations.
- Developed a *lightweight rollback mechanism* that is able to recover the lightweight LPs from causality errors without the need for sending anti-messages, mitigating rollback cost, limiting rollback propagation (both width and depth), and reducing the possibility of uncontrolled cascaded rollback.

7.2.2. Multicore Acceleration of DEVS Systems

In contrast to the LP-oriented model-decomposition approach, the MADS technique addresses the challenges of general-purpose parallel DEVS simulation on the heterogeneous Cell processor from a *data-flow* perspective, overcoming the performance bottlenecks in the simulation process while satisfying the requirements of the underlying hardware architecture. The key contributions of the MADS technique are summarized as follows.

- Identified and addressed two distinct types of *computational kernels* that are commonly found in demanding DEVS-based simulations, making the proposed technique applicable to simulations of a variety of different models.
- Developed *multi-grained parallelization strategies* that combine fine-grained data and event parallelism inherent in the simulation process with other forms of parallelism available at different levels of the system to accelerate the computational kernels, providing an alternative approach to PDES on emerging CMP architectures.
- Introduced the concept of *LP virtualization* for efficient mapping of LPs to a limited group of processing elements at runtime, allowing for improved processor utilization, reduced synchronization overhead, and fine-grained dynamic load balancing.
- Proposed varied methods for restructuring and optimizing simulation code and data, offering valuable insight and practical guidance for software development on CMP architectures.
- Provided the necessary support for general users to harness the potential of the Cell processor in P-DEVS and Cell-DESV simulations without being distracted by the

technical details of multicore programming, enhancing modeler productivity and lowering user learning curve with reduced M&S cost.

- Developed a modular, extensible software framework for meeting the needs of specific simulation requirements and for accommodating future expansion and development.

7.3. Suggestions for Future Research

There are a number of interesting topics for future research on DEVS-based high-performance parallel simulation with various extensions of the work presented in this dissertation. Some of the more prominent topics and open issues are suggested in this section.

7.3.1. Future Research on the LTW protocol

The following summarizes a list of issues that warrant further investigation in the context of the LTW protocol proposed in this dissertation.

- Integration of the LTW protocol with other TW optimization strategies to further improve the performance of P-DEVS and Cell-DEVS simulation on distributed-memory multiprocessor cluster systems. As the full-fledged and the interface TWLPs still rely on the persistent input, output and state queues, simulation performance could be enhanced using different state-saving strategies, event cancellation techniques, and event set implementations in the TW domains. Moreover, varied optimism control techniques could be applied to regulate overly optimistic execution at the cluster level.
- Incorporation of dynamic load balancing algorithms to support migration of lightweight LPs in DEVS-based TW simulations, taking advantage of the reduced overhead for transferring these LPs across cluster nodes. This research would allow for a quantitative evaluation of the impact of the LTW protocol on dynamic process migration. In addition, mechanisms for dynamic creation and deletion of lightweight LPs should be investigated to support runtime structural changes in optimistic DEVS systems.
- Application of the LTW concepts to other types of TW-based PDES systems. This research would investigate general methods for constructing a PDES system in a way that complies with the LTW assumptions as outlined in Chapter 4. If the LTW protocol could not be directly applied to a PDES system as a whole, algorithms would be required to detect the appropriate conditions under which certain parts of the system (or

certain periods of a single simulation) are able to execute in a lightweight manner in order to reduce TW operational overhead.

- Performance evaluation using an extended set of models with thorough sensitivity analysis. This sensitivity analysis would be useful to determine the effects that different model attributes (e.g., computational intensity, interconnectivity, and partition schemes), simulator configurations (e.g., event and state sizes, GVT estimation and fossil collection frequency, and checkpointing interval), and system parameters (e.g., size of available memory space, number of participating cluster nodes, inter-node communication characteristics, and background load fluctuation) have on simulation performance under the LTW protocol. Besides, additional performance metrics could be collected in the experiments to evaluate other aspects of the simulation performance. For instance, the number of active nodes (or LPs) at each virtual time and the percentage of useful work performed by the LPs (i.e., the proportion of events committed among the total number of events processed) would be useful to analyze the actual degree of parallelism available in a simulation under specific configurations and the effectiveness of the optimistic synchronization protocol.

7.3.2. Future Research on the MADS technique

Several issues with respect to the MADS technique need to be investigated in future research, as outlined below.

- Incorporation of additional computational kernels (e.g., random number generation and special-purpose libraries) into the MADS software architecture to meet the requirements of specific P-DEVS and Cell-DEVS simulations.
- Integration of the MADS technique with cluster-based conservative and optimistic synchronization algorithms to achieve high-performance parallel DEVS simulation on a cluster of Cell processors or on Cell-accelerated hybrid supercomputers.
- Sensitivity analysis of the varied MADS parameters (e.g., size of TA and IA blocks, size of event buffer entries, size of state buffer entries, size of pending job chunks, SEK notification frequency, and the number of SPEs allocated to each computational kernel) in order to evaluate their impact on simulation performance quantitatively. In addition, experiments could be conducted to evaluate the improvement of modeler productivity

as a result of the proposed methods for hiding multicore programming details.

- Investigation of new ways to exploiting the potential of the Cell processor in PDES systems. One possible direction would be to accelerate certain types of computation in TW simulations (e.g., rollback operation, GVT estimation, and fossil collection).
- Development of techniques for interfacing MADS-based parallel DEVS simulation with different graphical modeling and visualization tools [Wai09b] on multicore platforms towards the realization of interactive virtual reality and simulation-based serious games [Wai10].
- Extension of the key methods derived from the MADS technique, especially the data-flow oriented multi-grained parallelization strategy and the LP virtualization technique, to other homogeneous and heterogeneous CMP architectures.

References

[Aga08] Agarwal, V., L. K. Liu, and D. A. Bader, "Financial Modeling on the Cell Broadband Engine", *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, pp. 1-12, 2008.

[Agr91] Agre, J. R., and P. A. Tinker, "Useful Extensions to a Time Warp Simulation System", *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, Anaheim, CA, pp. 78-85, 1991.

[Aji08] Aji, A. M., W. Feng, F. Blagojevic, and D. S. Nikolopoulos, "Cell-SWat: Modeling and Scheduling Wavefront Computations on the Cell Broadband Engine", *Proceedings of the 5th Conference on Computing Frontiers*, Ischia, Italy, pp. 13-22, 2008.

[Aky93] Akyildiz, I. F., L. Chen, S. R. Das, R. M. Fujimoto, and R. F. Serfozo, "The Effect of Memory Capacity on Time Warp Performance", *Journal of Parallel and Distributed Computing*, 18(4), pp. 411-422, 1993.

[Ame01] Ameghino, J., A. Troccoli, and G. Wainer, "Models of Complex Physical Systems using Cell-DEVS", *Proceedings of the 34th Annual Simulation Symposium*, Seattle, WA, pp. 266-273, 2001.

[Ara09] Araya-Polo, M., F. Rubio, R. Cruz, M. Hanzich, J. M. Cela, and D. P. Scarpazza, "3D Seismic Imaging through Reverse-Time Migration on Homogeneous and Heterogeneous Multi-Core Processors", *Scientific Programming*, 17(1-2), pp. 185-198, 2009.

[Are08] Arevalo, A., R. M. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, C. Almond, *Progamming the Cell Broadband Engine Architecture: Examples and Best Practices*, 1st Edition, IBM Corporation, 2008.

[Art89] Artsy, Y., and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience", *IEEE Computer*, 22(9), pp. 47-56, 1989.

[Asa06] Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley", University of California at Berkeley, Berkeley, CA, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

[Avr95] Avril, H., and C. Tropper, "Clustered Time Warp and Logic Simulation", *Proceedings of the 9th International Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 112-119, 1995.

[Avr96] Avril, H., and C. Tropper, "The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation", *Proceedings of the 10th International Workshop on Parallel and Distributed Simulation*, Philadelphia, PA, pp. 20-27, 1996.

[Avr01] Avril, H., and C. Tropper, "On Rolling Back and Checkpointing in Time Warp", *IEEE Transactions on Parallel and Distributed Systems*, 12(11), pp. 1105-1121, 2001.

[Bad07a] Bader, D. A., and V. Agarwal, "FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine", *Proceedings of the 14th International Conference on High Performance Computing*, LNCS 4873, Goa, India, pp. 172-184, 2007.

[Bad07b] Bader, D. A., V. Agarwal, and K. Madduri, "On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking", *Proceedings of the 21st IEEE International Symposium on Parallel and Distributed Processing*, Long Beach, CA, pp. 1-10, 2007.

[Bad08] Bader, D. A., A. Chandramowlishwaran, and V. Agarwal, "On the Design of Fast Pseudo-Random Number Generators for the Cell Broadband Engine and an Application to Risk Analysis", *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, pp. 520-527, 2008.

[Bae92] Baezner, D., C. Rohs, and H. Jones, "U.S. Army ModSim on Jade's Time Warp", *Proceedings of the 1992 Winter Simulation Conference*, Arlington, VA, pp. 665-671, 1992.

[Bag00] Bagrodia, R. L., and M. Takai, "Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages", *IEEE Transactions on Parallel and Distributed Systems*, 11(4), pp. 395-411, 2000.

[Bar08] Barker, K. J., K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the Petaflop Era: The Architecture and Performance of Roadrunner", *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Austin, TX, Article No. 1, 2008.

[Bau93] Bauer, H., and C. Sporrer, "Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving", *Proceedings of the 26th Annual Simulation Symposium*, Arlington, VA, pp. 12-20, 1993.

[Bau07] Bauer, D. W., and E. H. Page, "An Approach for Incorporating Rollback Through Perfectly Reversible Computation in a Stream Simulator", *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, San Diego, CA, pp. 171-178, 2007.

[Bel06] Bellens, P., J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A Programming Model for the Cell BE Architecture", *Proceedings of the ACM/IEEE SC 2006 Conference*, Tampa, FL, 2006.

[Ber98] Beraldi, R., and L. Nigro, "Performance of a Time Warp Based Simulator of Large Scale PCS Networks", *Simulation Practice and Theory*, 6(2), pp. 149-163, 1998.

[Bev96] Bevins, C. D., *fireLib User Manual and Technical Reference*, Release 1, 1996, Available at: http://www.fire.org/downloads/fireLib/1.0.4/firelib.pdf.

[Bla07a] Blagojevic, F., D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos, "Dynamic Multigrain Parallelization on the Cell Broadband Engine", *Proceedings of the 12st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, pp. 90-100, 2007.

[Bla07b] Blagojevic, F., D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury, "Runtime Scheduling of Dynamic Parallelism on Accelerator-based Multi-core Systems", *Parallel Computing*, 33(10-11), pp. 700-719, 2007.

[Bla08] Blagojevic, F., X. Feng, K. W. Cameron, and D. S. Nikolopoulos, "Modeling Multigrain Parallelism on Heterogeneous Multi-core Processors: A Case Study of the Cell BE", *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, Goteborg, Sweden, LNCS 4917, pp. 38-52, 2008.

[Bou95] Boukerche, A., and C. Tropper, "Parallel Simulation on the Hypercube Multiprocessor", *Distributed Computing*, 8(4), pp. 181-190, 1995.

[Bou97] Boukerche, A., and S. K. Das, "Dynamic Load Balancing Strategies for Conservative Parallel Simulations", *Proceedings of the 11th International Workshop on Parallel and Distributed Simulation*, Lockenhaus, Austria, pp. 20-28, 1997.

[Bou00] Boukerche, A., "Conservative Circuit Simulation on Multiprocessor Machines", *Proceedings of the 7th International Conference on High Performance Computing*, Bangalore, India, LNCS 1970, pp. 415-424, 2000.

[Bou05] Boukerche, A., A. Mikkler, and A. Fabri, "Resource Control for Large-Scale Distributed Simulation System over Loosely Coupled Domains", *Journal of Parallel and Distributed Computing*, 65(10), pp. 1171-1189, 2005.

[Bro88] Brown, R., "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem", *Communications of the ACM*, 31(10), pp. 1220-1227, 1988.

[Bro09] Broutin, E., B. Paul, and J. Santucci, "Simulation of Heterogeneous DEVS Models: Application to the Study of Natural Systems", *Proceedings of the 2009 Spring Simulation Multiconference*, San Diego, CA, Article No. 148, 2009.

[Bry77] Bryant, R. E., "Simulation of Packet Communications Architecture Computer Systems", Massachusetts Institute of Technology, Cambridge, MA, Tech. Rep. MIT-LCS-TR-188, Nov. 1977.

[Buy99] Buyya, R., (ed.) *High Performance Cluster Computing: Architectures amd Systems*, New Jersey: Prentice Hall, 1999.

[Car95] Carothers, C. D., R. M. Fujimoto, and Y. B. Lin, "A Case Study in Simulating PCS Networks using Time Warp", *ACM SIGSIM Simulation Digest*, 25(1), pp. 87-94, 1995.

[Car99] Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto, "Efficient Optimistic Parallel Simulations Using Reverse Computation", *ACM Transactions on Modeling and Computer Simulation*, 9(3), pp. 224-253, 1999.

[Car00] Carothers, C. D., and R. M. Fujimoto, "Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms", *IEEE Transactions on Parallel and Distributed Systems*, 11(3), pp. 299-317, 2000.

[Car02] Carothers, C. D., D. Bauer, and S. Pearce, "ROSS: A High-Performance, Low-Memory, Modular Time Warp System", *Journal of Parallel and Distributed Computing*, 62(11), pp. 1648-1669, 2002.

[Cha79] Chandy, K. M., and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs", *IEEE Transactions on Software Engineering*, SE-5(5), pp. 440-452, 1979.

[Cha81] Chandy, K. M., and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", *Communications of the ACM*, 24(4), pp. 198-206, 1981.

[Cha83] Chandy, K. M., and J. Misra, "Distributed Deadlock Detection", *ACM Transactions on Computer Systems*, 1(2), pp. 144-156, 1983.

[Cha07] Chapman, B., G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, Cambridge: MIT Press, 2007.

[Che01] Chetlur, M., and P. A. Wilsey, "Causality Representation and Cancellation Mechanisms in Time Warp Simulations", *Proceedings of the 15th International Workshop on Parallel and Distributed Simulation*, Lake Arrowhead, CA, pp. 165-172, 2001.

[Che05] Chen, G., and B. K. Szymanski, "DSIM: Scaling Time Warp to 1,033 Processors", *Proceedings of the 2005 Winter Simulation Conference*, Orlando, FL, pp. 346-355, 2005.

[Che06] Chetlur, M., and P. A. Wilsey, "Causality Information and Fossil Collection in Time Warp Simulations", *Proceedings of the 2006 Winter Simulation Conference*, Monterey, CA, pp. 987-994, 2006.

[Che07] Chen, T., R. Raghavan, J. N. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its First Implementation – A Performance View", *IBM Journal of Research and Development*, 51(5), pp. 559-572, 2007.

[Che09a] Chetlur, M., and P. A. Wilsey, "Causality Information and Proactive Cancellation Mechanisms", *Concurrency and Computation: Practice and Experience*, 21(11), pp. 1483-2503, 2009.

[Che09b] Chellappa, S., F. Franchetti, and M. Puschel, "Computer Generation of Fast Fourier Transforms for the Cell Broadband Engine", *Proceedings of the 23rd International Conference on Supercomputing*, Yorktown Heights, NY, pp. 26-35, 2009.

[Chi76] Chicoix, C., J. Pedoussat, and N. Giambiasi, "An Accurate Time Delay Model for Large Digital Network Simulation", *Proceedings of the 13th ACM/IEEE Design Automation Conference*, San Francisco, CA, pp. 54-60, 1976.

[Chi07] Chidisiuc, C., and G. Wainer, "CD++Builder: An Eclipse-based IDE for DEVS Modeling", *Proceedings of the 2007 Spring Simulation Multiconference*, Norfolk, VA, pp. 235-240, 2007.

[Cho94] Chow, A. C., and B. P. Zeigler, "Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism", *Proceedings of the 1994 Winter Simulation Conference*, Orlando, FL, pp. 716-722, 1994.

[Cho03] Cho, Y. K., X. Hu, and B. P. Zeigler, "The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems", *SIMULATION*, 79(4), pp. 197-210, 2003.

[Chr90] Christensen, E. R., *Hierarchical Optimistic Distributed Simulation: Combining DEVS and Time Warp*, PhD Dissertation, University of Arizona, Tucson, AZ, 1990.

[Chu06] Chung, M. K., and C. M. Kyung, "Improving Lookahead in Parallel Multiprocessor Simulation Using Dynamic Execution Path Prediction", *Proceedings of the 20th International Workshop on Principles of Advanced and Distributed Simulation*, Singapore, pp. 11-18, 2006.

[Com79] Comfort, J. C., "A Taxonomy and Analysis of Event Set Management Algorithms for Discrete Event Simulation", *Proceedings of the 12th Annual Simulation Symposium*, Tampa, FL, pp. 115-146, 1979.

[Cor06] Correia, L., and T. Wehrle, "Beyond Cellular Automata, Towards More Realistic Traffic Simulators", *Cellular Automata*, LNCS 4173, pp. 690-693, 2006.

[Cra08] Crawford, C. H., P. Henning, M. Kistler, and C. Wright, "Accelerating Computing with the Cell Broadband Engine Processor", *Proceedings of the 5th Conference on Computing Frontiers*, Ischia, Italy, pp. 3-12, 2008.

[Das93] Das, S. R., and R. M. Fujimoto, "A Performance Study of the Cancelback Protocol for Time Warp", *Proceedings of the 7th International Workshop on Parallel and Distributed Simulation*, San Diego, CA, pp. 135-142, 1993.

[Das94] Das, S. R., R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: A Time Warp System for Shared Memory Multiprocessors", *Proceedings of the 1994 Winter Simulation Conference*, Orlando, FL, pp. 1332-1339, 1994.

[Das96] Das, S. R., "Adaptive Protocols for Parallel Discrete Event Simulation", *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA, pp. 186-193, 1996.

[DeF07] De Fabritiis, G., "Performance of the Cell Processor for Biomolecular Simulations", *Computer Physics Communications*, 176(11-12), pp. 660-664, 2007.

[DeV90] De Vries, R. C., "Reducing Null Messages in Misra's Distributed Discrete Event Simulation Method", *IEEE Transactions on Software Engineering*, 16(1), pp. 82-91, 1990.

[Doc09] Docan, C., M. Parashar, and C. Marty, "Advanced Risk Analytics on the Cell Broadband Engine", *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, pp. 1-8, 2009.

[Eic05] Eichenberger, A. E., J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing Compiler for the Cell Processor", *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, pp. 161-172, 2005.

[Eic06] Eichenberger, A. E., J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo, "Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture", *IBM Systems Journal*, 45(1), pp. 59-84, 2006.

[El-K99] El-Khatib, K., and C. Tropper, "On Metrics for the Dynamic Load Balancing of Optimistic Simulations", *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, Manoa, HI, Track 8, 1999.

[Fat06] Fatahalian, K., T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the Memory Hierarchy", *Proceedings of the ACM/IEEE SC 2006 Conference*, Tampa, FL, 2006.

[Fen06] Feng, T. H., and E. A. Lee, "Incremental Checkpointing with Application to Distributed Discrete Event Simulation", *Proceedings of the 2006 Winter Simulation Conference*, Monterey, CA, pp. 1004-1011, 2006.

[Fen08] Feng, B., Q. Liu, and G. Wainer, "Parallel Simulation of DEVS and Cell-DEVS Models on Windows-based PC Cluster Systems", *Proceedings of the 2008 Spring Simulation Multiconference: High Performance Computing Symposium*, Ottawa, Canada, pp. 439-446, 2008.

[Fer95] Ferscha, A., "Probabilistic Adaptive Direct Optimism Control in Time Warp", *Proceedings of the 9th International Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 120-129, 1995.

[Fil09] Filippi, J. B., F. Morandini, J. H. Balbi, and D. Hill, "Discrete Event Front-Tracking Simulation of a Physical Fire-Spread Model", *SIMULATION*, 2009.

[Fis73] Fishman, G. S., *Concepts and Methods in Discrete Event Digital Simulation*, New York: Wiley-Interscience, 1973.

[Fle95] Fleischmann, J., and P. A. Wilsey, "Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulations", *Proceedings of the 9th International Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 50-58, 1995.

[Fuj90] Fujimoto, R. M., "Parallel Discrete Event Simulation", *Communications of the ACM*, 33(10), pp. 30-53, 1990.

[Fuj97]Fujimoto, R. M., and M. Hybinette, "Computing Global Virtual Time in Shared-Memory Multiprocessors", *ACM Transactions on Modeling and Computer Simulatiuon*, 7(4), pp. 425-446, 1997.

[Fuj00] Fujimoto, R. M., *Parallel and Distributed Simulation Systems*, New York: Wiley-Interscience, 2000.

[Fuj01] Fujimoto, R. M., "Parallel and Distributed Simulation Systems", *Proceedings of the 2001 Winter Simulation Conference*, Arlington, VA, pp. 147-157, 2001.

[Fuj03] Fujimoto, R. M., "Distributed Simulation Systems", *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA, pp. 124-134, 2003.

[Gaf88] Gafni, A., "Rollback Mechanisms for Optimistic Distributed Simulation Systems", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, CA, pp. 61-67, 1988.

[Gar70] Gardner, M., "The Fantastic Combinations of John Conway's New Solitaire Game 'Life'", *Scientific American*, 223(4), pp. 120-123, 1970.

[Ged07] Gedik, B., P. S. Yu, and R. R. Bordawekar, "Executing Stream Joins on the Cell Processor", *Proceedings of the 33rd International Conference on Very Large Data Bases*, Vienna, Austria, pp. 363-374, 2007.

[Gla93] Glazer, D. W., and C. Tropper, "On Process Migration and Load Balancing in Time Warp", *IEEE Transactions on Parallel and Distributed Systems*, 4(3), pp. 318-327, 1993.

[Gli06] Glinsky, E., and G. Wainer, "New Parallel Simulation Techniques of DEVS and Cell-DEVS in CD++", *Proceedings of the 39th Annual Simulation Symposium*, Huntsville, AL, pp. 244-251, 2006.

[Gom97] Gomes, F., B. unger, J. Cleary, and S. Franks, "Multiplexed State Saving for Bounded Rollback", *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, GA, pp. 460-467, 1997.

[Gro91] Groselj, B., and C. Tropper, "The Distributed Simulation of Clustered Processes", *Distributed Computing*, 4(3), pp. 111-121, 1991.

[Gro99] Gropp, W., E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd Edition, Cambridge: MIT Press, 1999.

[Gro09] Gropp, W., E. Lusk, D. Ashton, P. Balaji, D. Buntinas, R. Butler, A. Chan, D. Goodell, J. Krishna, G. Mercier, R. Ross, R. Thakur, and B. Toonen, *MPICH2 User's Guide*, 2009, Available at: http://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-1.2.1-userguide.pdf.

[Gsc06] Gschwind, M., "Chip Multiprocessing and the Cell Broadband Engine", *Proceedings of the 3rd Conference on Computing Frontiers*, Ischia, Italy, pp. 1-8, 2006.

[Gsc07] Gschwind, M., "The Cell Broadband Engine: Exploiting Multiple Levels of Parallelism in a Chip Multiprocessor", *International Journal of Parallel Programming*, 35(3), pp. 233-262, 2007.

[Gum08] Gummaraju, J., J. Coburn, Y. Turner, and M. Rosenblum, "Streamware: Programming General-Purpose Multicore Processors Using Streams", *ACM SIGARCH Computer Architecture News*, 36(1), pp. 297-307, 2008.

[Har08] Harzallah, Y., V. Michel, Q. Liu, and G. Wainer, "Distributed Simulation and Web Map Mash-Up for Forest Fire Spread", *Proceedings of the 2008 IEEE Congress on Services – Part I*, Honolulu, HI, pp. 176-183, 2008.

[Ho89] Ho, Y. C., "Introduction to Special Issue on Dynamics of Discrete Event Systems", *Proceedings of the IEEE*, 77(1), pp. 3-6, 1989.

[Hu07] Hu, X., and Y. Sun, "Agent-Based Modeling and Simulation of Wildland Fire Suppression", *Proceedings of the 2007 Winter Simulation Conference*, San Diego, CA, pp. 1275-1283, 2007.

[IBM08a] IBM Corporation, *IBM XL C/C++ for Multicore Acceleration for Linux – Compiler Reference*, Version 10.1, 1st Edition, 2008.

[IBM08b] IBM Corporation, *Software Development Kit for Multicore Acceleration – IDE Tutorial and User's Guide*, Version 3.1, 2008.

[IBM08c] IBM Corporation, *Software Development Kit for Multicore Acceleration – Accelerated Library Framework: Programmer's Guide and API reference*, Version 3.1, 2008.

[IBM09] IBM Corporation, *Cell Broadband Engine Programming Handbook: Including the PowerXCell 8i Processor*, Version 1.12, 2009.

[IBM10a] IBM Corporation, *Cell Broadband Engine Resource Center*, Available on-line at: https://www.ibm.com/developerworks/power/cell/documents.html, accessed in July, 2010.

[IBM10b] IBM Corporation, *IBM BladeCenter QS22 Specification*, Available on-line at: http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs22/specs.html, accessed in July, 2010.

[Iko00] Ikonen, J., and J. Porras, "Automatic Load Distribution for Conservative Distributed Simulation", *Proceedings of the 14th European Simulation Multiconference on Simulation and Modelling: Enablers for a Better Quality of Life*, SCS Europe, pp. 49-53, 2000.

[Jef85] Jefferson, D. R., "Virtual Time", *ACM Transactions on Programming Languages and Systems*, 7(3), pp. 405-425, 1985.

[Jef90] Jefferson, D. R., "Virtual Time II: Storage Management in Distributed Simulation", *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, pp. 75-89, 1990.

[Jha00] Jha, V., and R. Bagrodia, "Simultaneous Events and Lookahead in Simulation Protocols", *ACM Transactions on Modeling and Computer Simulation*, 10(3), pp. 241-267, 2000.

[Jia94] Jiang, M. R., S. P. Shieh, and C. L. Liu, "Dynamic Load Balancing in Parallel Simulation Using Time Warp Mechanism", *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, Taiwan, China, pp. 222-227, 1994.

[Jon86] Jones, D. W., "An Empirical Comparison of Priority-Queue and Event-Set Implementations", *Communications of the ACM*, 29(4), pp. 300-311, 1986.

[Jon89] Jones, D. W., "Concurrent Operations on Priority Queues", *Communications of the ACM*, 32(1), pp. 132-137, 1989.

[Kan96] Kannikeswaran, B., R. Radhakrishnan, P. Frey, P. Alexander, and P. A. Wilsey, "Formal Specification and Verification of the pGVT Algorithm", *Proceedings of the 3rd International Symposium of Formal Methods Europe – Industrial Benefit and Advances in Formal Methods*, Oxford, UK, LNCS 1051, pp. 405-424, 1996.

[Kar05] Kari, J., "Theory of Cellular Automata: A Survey", *Theoretical Computer Science*, 334(1-3), pp. 3-33, 2005.

[Kha05] Khale, J. A., M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introducing to the Cell Multiprocessor", *IBM Journal of Research and Development*, 49(4), pp. 589-604, 2005.

[Kim96] Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park, "Distributed Simulation of Hierarchical DEVS Models: Hierarchical Scheduling Locally and Time Warp Globally", *TRANSACTIONS of The Society for Computer Simulation International*, 13(3), pp. 135-154, 1996.

[Kim97] Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park, "Ordering of Simultaneous Events in Distributed DEVS Simulation", *Simulation Practice and Theory*, 5(3), pp. 253-268, 1997.

[Kim98] Kim, K. H., T. G. Kim, and K. H. Park, "Hierarchical Partitioning Algorithm for Optimistic Distributed Simulation of DEVS Models", *Journal of Systems Architecture*, 44(6-7), pp. 433-455, 1998.

[Kim04] Kim, K. H., and W. S. Kang, "CORBA-Based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One", *Proceedings of the 2004 International Conference on Computational Science and Its Applications*, Assisi, Italy, pp. 167-176, 2004.

[Kis06] Kistler, M., M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed", *IEEE Micro*, 26(3), pp. 10-23, 2006.

[Kni07] Knight, T. J., J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan, "Compilation for Explicitly Managed Memory Hierarchies", *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, pp. 226-236, 2007.

[Kon05] Kongetira, P., K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor", *IEEE Micro*, 25(2), pp. 21-29, 2005.

[Kot05] Kota, R., and R. Oehler, "Horus: Large-scale Symmetric Multiprocessing for Opteron Systems", *IEEE Micro*, 25(2), pp. 30-40, 2005.

[Kud08] Kudlur, M., and S. Mahlke, "Orchestrating the Execution of Stream Programs on Multicore Platforms", *ACM SIGPLAN Notices*, 43(6), pp. 114-124, 2008.

[Kum05] Kumar, R., D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous Chip Multiprocessors", *Computer*, 38(11), pp. 32-38, 2005.

[Kum06] Kumar, R., D. M. Tullsen, and N. P. Jouppi, "Core Architecture Optimization for Heterogeneous Chip Multiprocessors", *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, Seattle, WA, pp. 23-32, 2006.

[Kum07] Kumar, A., G. Senthikumar, M. Krishna, N. Jayam, P. K. Baruah, R. Sarma, A. Srinivasan, and S. Kapoor, "Feasibility Study of MPI Implementation on the Heterogeneous Multi-Core Cell BE Architecture", *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, pp. 55-56, 2007.

[Leg96] Legedza, U., and W. E. Weihl, "Reducing Synchronization Overhead in Parallel Simulation", *Proceedings of the 10th International Workshop on Parallel and Distributed Simulation*, Philadelphia, PA, pp. 86-95, 1996.

[Lev07] Leverich, J., H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing Memory Systems for Chip Multiprocessors", *ACM SIGARCH Computer Architecture News*, 35(2), pp. 358-368, 2007.

[Li04] Li, L., and C. Tropper, "Event Reconstruction in Time Warp", *Proceedings of the 18th International Workshop on Principles of Advanced and Distributed Simulation*, Kufstein, Austria, pp. 37-44, 2004.

[Lin91a] Lin, Y. B., and E. D. Lazowska, "A Study of Time Warp Rollback Mechanisms", *ACM Transactions on Modeling and Computer Simulatiuon*, 1(1), pp. 51-72, 1991.

[Lin91b] Lin, Y. B., and B. R. Preiss, "Optimal Memory Management for Time Warp Parallel Simulation", *ACM Transactions on Modeling and Computer Simulatiuon*, 1(4), pp. 283-307, 1991.

[Lin93] Lin, Y. B., B. R. Preiss, W. M. Loucks, and E. D. Lazowska, "Selecting the Checkpoint Interval in Time Warp Simulation", *ACM SIGSIM Simulation Digest*, 23(1), pp. 3-10, 1993.

[Lin94] Lin, Y. B., "Memory Management Algorithms for Optimistic Parallel Simulation", *Information Sciences*, 77(1-2), pp. 119-140, 1994.

[Lin96] Lin, Y. B., and P. A. Fishwick, "Asynchronous Parallel Discrete Event Simulation", *IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans*, 26(4), pp. 397-412, 1996.

[Lin08] Lindholm, E., J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro*, 28(2), pp. 39-55, 2008.

[Liu02] Liu, J., and D. M. Nicol, "Lookahead Revisited in Wireless Network Simulations", *Proceedings of the 16th International Workshop on Parallel and Distributed Simulation*, Washington, D.C., pp. 79-88, 2002.

[Liu06] Liu, Q., *Distributed Optimistic Simulation of DEVS and Cell-DEVS Models with PCD++*, Master's Thesis, Carleton University, 2006.

[Liu07] Liu, Q., and G. Wainer, "Parallel Environment for DEVS and Cell-DEVS Models", *SIMULATION*, 83(6), pp. 449-471, 2007.

[Liu08] Liu, Q., and G. Wainer, "Lightweight Time Warp – A Novel Protocol for Parallel Optimistic Simulation of Large-Scale DEVS and Cell-DEVS Models", *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Vancouver, Canada, pp. 131-138, 2008.

[Liu09] Liu, Q., and G. Wainer, "A Performance Evaluation of the Lightweight Time Warp Protocol in Optimistic Parallel Simulation of DEVS-based Environmental Models", *Proceedings of the 23rd IEEE Workshop on Principles of Advanced and Distributed Simulation*, Lake Placid, NY, pp. 27-34, 2009.

[Liu10a] Liu, Q., and G. Wainer, "Accelerating Large-scale DEVS-based Simulation on the Cell Processor", *Proceedings of the 2010 Symposium on Theory of Modeling and Simulation – DEVS Integrative M&S Symposium*, Orlando, FL, pp. 191-198, 2010.

[Liu10b] Liu, Q., G. Wainer, L. Lu, and M. Perrone, "Novel Performance Optimization of Large-Scale Discrete-Event Simulation on the Cell Broadband Engine", *Proceedings of the 2010 International Conference on High Performance Computing & Simulation*, Caen, France, pp. 108-114, 2010.

[Liu10c] Liu, Q., and G. Wainer, "Exploring Multi-Grained Parallelism in Compute-Intensive DEVS Simulations", *Proceedings of the 24th IEEE Workshop on Principles of Advanced and Distributed Simulation*, Atlanta, GA, pp. 65-72, 2010.

[Low02] Low, M. Y. H., "Dynamic Load-Balancing for BSP Time Warp", *Proceedings of the 35th Annual Simulation Symposium*, San Diego, CA, pp. 267-274, 2002.

[Lub89] Lubachevsky, B. D., "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks", *Communications of the ACM*, 32(1), pp. 111-123, 1989.

[Mad88] Madisetti, V., J. Walrand, and D. Messerschmitt, "WOLF: A Rollback Algorithm for Optimistic Distributed Simulation Systems", *Proceedings of the 1988 Winter Simulation Conference*, San Diego, CA, pp. 296-305, 1988.

[Mar03] Martin, D. E., P. A. Wilsey, R. J. Hoekstra, E. R. Keiter, S. A. Hutchinson, T. V. Russo, and L. J. Waters, "Redesigning the WARPED Simulation Kernel for Analysis and Application Development", *Proceedings of the 36th Annual Simulation Symposium*, Orlando, FL, pp. 216-223, 2003.

[Mat93] Mattern, F., "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation", *Journal of Parallel and Distributed Computing*, 18(4), pp. 423-434, 1993.

[McC81] McCormack, W. M., and R. G. Sargent, "Analysis of Future Event Set Algorithms for Discrete Event Simulation", *Communications of the ACM*, 24(12), pp. 801-812, 1981.

[McC06] McCool, M. D., "Data-Parallel Programming on the Cell BE and the GPU Using the RapidMind Development Platform", *Proceedings of the GSPx Multicore Applications Conference*, Santa Clara, CA, 2006.

[McC08] McCool, M. D., "Scalable Programming Models for Massively Multicore Processors", *Proceedings of the IEEE*, 96(5), pp. 816-831, 2008.

[McK04] McKee, S. A., "Reflections on the Memory Wall", *Proceedings of the 1st Conference on Computing Frontiers*, Ischia, Italy, pp. 162-167, 2004.

[McL03] McLean, T., and R. M. Fujimoto, "Predictable Time Management for Real-Time Distributed Simulation", *Proceedings of the 17th International Workshop on Parallel and Distributed Simulation*, San Diego, CA, pp. 89-96, 2003.

[McN05] McNairy, C., and R. Bhatia, "Montecito: A Dual-Core, Dual-Thread Itanium Processor", *IEEE Micro*, 25(2), pp. 10-20, 2005.

[Meh92] Mehl, H., "A Deterministic Tie-Breaking Scheme for Sequential and Distributed Simulation", *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, Newport Beach, CA, 1992.

[Mer07] Meredith, J. S., S. R. Alam, and J. S. Vetter, "Analysis of a Computational Biology Simulation Technique on Emerging Processing Architectures", *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, pp. 1-8, 2007.

[Mey99] Meyer, R. A., and R. L. Bargrodia, "Path Lookahead: A Data Flow View of PDES Models", *Proceedings of the 13th International Workshop on Parallel and Distributed Simulation*, Atlanta, GA, pp. 12-19, 1999.

[Mit09] Mittal, S., J. L. Risco-Martin, and B. P. Zeigler, "DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process", *SIMULATION*, 85(7), pp. 419-450, 2009.

[Mis86] Misra, J., "Distributed Discrete-Event Simulation", *ACM Computing Surveys*, 18(1), pp. 39-65, 1986.

[Moo96] Moon, Y., B. P. Zeigler, G. Ball, and D. P. Guertin, "DEVS Representation of Spatially Distributed Systems: Validity, Complexity Reduction". *IEEE Transactions on Systems, Man and Cybernetics*, pp. 288-296, 1996.

[Mor06] Morad, T. Y., U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade, "Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors", *Computer Architecture Letters*, 5(1), pp. 14-17, 2006.

[Nab07] Naborskyy, A., and R. M. Fujimoto, "Using Reversible Computation Techniques in a Parallel Optimistic Simulation of a Multi-Processor Computing System", *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, San Diego, CA, pp. 179-188, 2007.

[Nan81] Nance, R. E., "The Time and State Relationships in Simulation Modeling", *Communications of the ACM*, 24(4), pp. 173-179, 1981.

[Nan07] Nanda, A. K., J. R. Moulic, R. E. Hanson, G. Goldrian, M. N. Day, B. D. D'Amora, and S. Kesavarapu, "Cell/B.E. Blades: Building Blocks for Scalable, Real-Time,

Interactive, and Digital Media Servers", *IBM Journal of Research and Development*, 51(5), pp. 573-582, 2007.

[Neu66] Neumann, J. V., and A. W. Burks, *Theory of Self-Reproducing Automata*, Champaign: University of Illinois Press, 1966.

[Nic93] Nicol, D. M., "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations", *Journal of the ACM*, 40(2), pp. 304-333, 1993.

[Nic95] Nicol, D. M., "Noncommittal Barrier Synchronization", *Parallel Computing*, 21(4), pp. 529-549, 1995.

[Nic96] Nichols, B., D. Buttlar, and J. P. Farrell, *Pthreads Programming*, 1st Edition, Sebastopol: O'Reilly & Associates, Inc., 1996.

[Nik93] Nikolopoulos, S. D., and R. MacLeod, "An Experimental Analysis of Event Set Algorithms for Discrete Event Simulation", *Microprocessing and Microprogramming*, 36(2), pp. 71-81, 1993.

[Nke01] Nketsa, A., and N. B. Khalifa, "Timed Petri Nets and Prediction to Improve the Chandy-Misra Conservative Distributed Simulation", *Applied Mathematics and Computation*, 120(1-3), pp. 235-254, 2001

[Nor02] Noronha, R., and N. B. Abu-Ghazaleh, "Early Cancellation: An Active NIC Optimization for Time-Warp", *Proceedings of the 16th International Workshop on Parallel and Distributed Simulation*, Washington, DC, pp. 43-50, 2002.

[Nta04] Ntaimo, L., B. P. Zeigler, M. J. Vasconcelos, and B. Khargharia, "Forest Fire Spread and Suppression in DEVS", *SIMULATION*, 80(10), pp. 479-500, 2004.

[Nta08] Ntaimo, L., X. Hu, and Y. Sun, "DEVS-FIRE: Towards an Integrated Simulation Environment for Surface Wildfire Spread and Containment", *SIMULATION*, 84(4), pp. 137-155, 2008.

[Nut04] Nutaro, J., "Risk-Free Optimistic Simulation of DEVS Models", *Proceedings of the 2004 Advanced Simulation Technologies Conference – Military, Government, and Aerospace Simulation*, Arlington, VA, 2004.

[Nut08] Nutaro, J., "On Constructing Optimistic Simulation Algorithms for the Discrete Event System Specification", *ACM Transactions on Modeling and Computer Simulatiuon*, 19(1), Article 1, 2008.

[Oh97] Oh, S. H., and J. S. Ahn, "Dynamic Lazy Calendar Queue: An Event List for Network Simulation", *Proceedings of High Performance Computing on the Information Superhighway (HPC Asia'97)*, Seoul, South Korea, pp. 254-259, 1997.

[Oh99] Oh, S. H., and J. S. Ahn, "Dynamic Calendar Queue", *Proceedings of the 32nd Annual Simulation Symposium*, San Diego, CA, pp. 20-25, 1999.

[Oha06] Ohara, M., H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "MPI Microtask for Programming the Cell Broadband Engine Processor", *IBM Systems Journal*, 45(1), pp. 85- 102, 2006.

[Oli07] Olivier, S., J. Prins, J. Derby, and K. Vu, "Porting the GROMACS Molecular Dynamics Code to the Cell Processor", *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, pp. 1-8, 2007.

[Olu05] Olukotun, K., and L. Hammond, "The Future of Microprocessors", *ACM Queue: Multiprocessors*, 3(7), pp. 26-34. 2005.

[Olu07] Olukotun, K., L. Hammond, and J. Laudon, *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*, 1st Edition. San Rafael: Morgan & Claypool Publishers, 2007.

[Pag94] Page, E., *Simulation Modeling Methodology: Principles and Etiology of Decision Support*, PhD Dissertation, Virgina Polytechnic Institute and State University, 1994.

[Pan97] Panesar, K. S., and R. M. Fujimoto, "Adaptive Flow Control in Time Warp", *Proceedings of the 11th International Workshop on Parallel and Distributed Simulation*, Lockenhaus, Austria, pp. 108-115, 1997.

[Par04] Park, A., R. M. Fujimoto, and K. S. Perumalla, "Conservative Synchronization of Large-Scale Network Simulations", *Proceedings of the 18th International Workshop on Parallel and Distributed Simulation*, Kufstein, Austria, pp. 153-161, 2004.

[Pen08] Peng, L., K. Nomura, T. Oyakawa, R. K. Kalia, A. Nakano, and P. Vashishta, "Parallel Lattice Boltzmann Flow Simulation on Emerging Multi-core Platforms", *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Las Palmas de Gran Canaria, Spain, LNCS 5168, pp. 763-777, 2008.

[Per01] Perumalla, K. S., and R. M. Fujimoto, "Virtual Time Synchronization over Unreliable Network Transport", *Proceedings of the 15th International Workshop on Parallel and Distributed Simulation*, Lake Arrowhead, CA, pp. 129-136, 2001.

[Per05] Perumalla, K. S., "μsik – A Micro-Kernel for Parallel/Distributed Simulation Systems", *Proceedings of the 19th International Workshop on Principles of Advanced and Distributed Simulation*, Monterey, CA, pp. 59-68, 2005.

[Per06] Perumalla, K. S., "Parallel and Distributed Simulation: Traditional Techniques and Recent Advances", *Proceedings of the 2006 Winter Simulation Conference*, Monterey, CA, pp. 84-95, 2006.

[Per07] Perez, J. M., P. Bellens, R. M. Badia, and J. Labarta, "CellSs: Making It Easier to Program the Cell Broadband Engine Processor", *IBM Journal of Research and Development*, 51(5), pp. 593-604, 2007.

[Pes07a] Peschlow, P., and P. Martini, "Efficient Analysis of Simultaneous Events in Distributed Simulation", *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, Chania, Greece, pp. 244-251, 2007.

[Pes07b] Peschlow, P., T. Honecker, and P. Martini, "A Flexible Dynamic Partitioning Algorithm for Optimistic Distributed Simulation", *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, San Diego, CA, pp. 219-228, 2007.

[Pet07] Petrini, F., G. Fossum, J. Fernandez, A. L. Varbanescu, M. Kistler, and M. Perrone, "Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine", *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, pp. 1-10, 2007.

[Pra94] Praehofer, H., and B. P. Zeigler, "On the Expressibility of Discrete Event Specified Systems", *Proceedings of the 4th International Workshop on Computer Aided Systems Theory*, LNCS 1105, Ottawa, Canada, pp. 65-79, 1994.

[Pre90] Presley, M. T., P. L. Reiher, and S. F. Bellenot, "A Time Warp Implementation of Sharks World", *Proceedings of the 1990 Winter Simulation Conference*, New Orleans, LA, pp. 199-203, 1990.

[Pre94] Preiss, B. R., W. M. Loucks, and I. D. Macintyre, "Effects of the Checkpoint Interval on Time and Space in Time Warp", *ACM Transactions on Modeling and Computer Simulatiuon*, 4(3), pp. 223-253, 1994.

[Pre95] Preiss, B. R., and W. M. Loucks, "Memory Management Techniques for Time Warp on a Distributed Memory Machine", *Proceedings of the 9th International Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 30-39, 1995.

[Qua98] Quaglia, F., "Event History Based Sparse State Saving in Time Warp", *ACM SIGSIM Simulation Digest*, 28(1), pp. 72-79, 1998.

[Qua01a] Quaglia, F., "A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation", *IEEE Transactions on Parallel and Distributed Systems*, 12(4), pp. 346-362, 2001.

[Qua01b] Quaglia, F., "A Scaled Version of the Elastic Time Algorithm", *Proceedings of the 15th International Workshop on Parallel and Distributed Simulation*, Lake Arrowhead, CA, pp. 157-164, 2001.

[Rad98] Radhakrishnan, R., D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey, "An Object-Oriented Time Warp Simulation Kernel", *Proceedings of the 2nd International Symposium on Computing in Object-Oriented Parallel Environments*, Santa Fe, NM, LNCS 1505, pp. 13-23, 1998.

[Raj93] Rajaei, H., R. Ayani, and L. E. Thorelli, "The Local Time Warp Approach to Parallel Simulation", *ACM SIGSIM Simulation Digest*, 23(1), pp. 119-126, 1993.

[Raj07] Rajaei, H., "Local Time Warp: An Implementation and Performance Analysis", *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, San Diego, CA, pp. 163-170, 2007.

[Rei90] Reiher, P. L., and D. R. Jefferson, "Virtual Time Based Dynamic Load Management in the Time Warp Operating System", *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, San Diego, CA, pp. 103-111, 1990.

[Ron91] Ronngren, R., J. Riboe, and R. Ayani, "Lazy Queue: An Efficient Implementation of the Pending-event Set", *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, LA, pp. 194-204, 1991.

[Ron93] Ronngren, R., R. Ayani, R. M. Fujimoto, and S. R. Das, "Efficient Implementation of Event Sets in Time Warp", *ACM SIGSIM Simulation Digest*, 23(1), pp. 101-108, 1993.

[Ron94] Ronngren, R., and R. Ayani, "Adaptive Checkpointing in Time Warp", *Proceedings of the 8th International Workshop on Parallel and Distributed Simulation*, Edinburgh, UK, pp. 110-117, 1994.

[Ron96] Ronngren, R., M. Liljenstam, R. Ayani, and J. Montagnat, "Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation", *ACM SIGSIM Simulastion Digest*, 26(1), pp. 70-77, 1996.

[Ron99] Ronngren, R., M. Liljenstam, "On Event Ordering in Parallel Discrete Event Simulation", *Proceedings of the 13th International Workshop on Parallel and Distributed Simulation*, Atlanta, GA, pp. 38-45, 1999.

[Rot72] Rothermel, R. C., "A Mathematical Model for Predicting Fire Spread in Wild-Land Fuels", USDA Forest Service, Research Paper INT-115, Intermountain Forest and Range Experiment Station, Ogden, UT, 1972.

[Rot04] Rothman, D. H., and S. Zaleski, *Lattice-Gas Cellular Automata: Simple Models of Complex Hydrodynamics*, 1st Edition, Cambridge: Cambridge University Press, 2004.

[Roz93] Rozenblit, J. W., and B. P. Zeigler, "Representing and Constructing System Specifications Using the System Entity Structure Concepts", *Proceedings of the 1993 Winter Simulation Conference*, Los Angeles, CA, pp. 604-611, 1993.

[Sac04] Sachdev, V., M. Hybinette, and E. Kraemer, "Controlling Over-Optimism in Time-Warp via CPU-based Flow Control", *Proceedings of the 2004 Winter Simulation Conference*, Washington, DC, pp. 402-410, 2004.

[Sai07] Saidani, T., S. Piskorski, L. Lacassagne, and S. Bouaziz, "Parallelization Schemes for Memory Optimization on the Cell Processor: A case Study of Image Processing

Algorithm", *Proceedings of the 2007 Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, Brasov, Romania, pp. 9-16, 2007.

[Sca09a] Scarpazza, D. P., and G. W. Braudaway, "Workload Characterization and Optimization of High-Performance Text Indexing on the Cell Broadband Engine (Cell/B.E.)", *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, Austin, TX, pp. 13-23, 2009.

[Sca09b] Scarpazza, D. P., and G. F. Russell, "High-Performance Regular Expression Scanning on the Cell/B.E. Processor", *Proceedings of the 23rd International Conference on Supercomputing*, Yorktown Heights, NY, pp. 14-25, 2009.

[Sch95] Schlagenhaft, R., M. Ruhwandl, C. Sporrer, and H. Bauer, "Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations", *Proceedings of the 9th International Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, pp. 175-180, 1995.

[Seo95] Seong, Y. R., S. H. Jung, T. G. Kim, and K. H. Park, "Parallel Simulation of Hierarchical Modular DEVS Models: A Modified Time Warp Approach", *International Journal in Computer Simulation*, 5(3), pp. 263-285, 1995.

[Shi08] Shi, G., and V. Kindratenko, "Implementation of NAMD Molecular Dynamics Non-Bonded Force-Field on the Cell Broadband Engine Processor", *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, pp. 1-8, 2008.

[Ski98] Skillicorn, D. B., and D. Talia, "Models and Languages for Parallel Computation", *ACM Computing Surveys*, 30(2), pp. 123-169, 1998.

[Sko96] Skold, S., and R. Ronngren, "Event Sentitive State Saving in Time Warp Parallel Discrete Event Simulations", *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA, pp. 653-660, 1996.

[Sli09] Slimi, R., S. E. Yacoubi, E. Dumonteil, and S. Gourbiere, "A Cellular Automata Model for Chagas Disease", *Applied Mathematical Modelling*, 33(2), pp. 1072-1085, 2009.

[Sok91] Sokol, L. M., J. B. Weissman, and P. A. Mutchler, "MTW: An Empirical Performance Study", *Proceedings of the 1991 Winter Simulation Conference*, Phoenix, AZ, pp. 557-563, 1991.

[Sol96] Soliman, H. M., and A. S. Elmaghraby, "An Efficient Clustered Adaptive-Risk Technique for Distributed Simulation", *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, pp. 383-391, 1996.

[Sol97] Soliman, H. M., and A. S. Elmaghraby, "Performance Evaluation of the Aggressive Adaptive-Risk Approach for Parallel and Distributed Simulation", *Proceedings of the 30th Annual Simulation Symposium*, Atlanta, GA, pp. 50-55, 1997.

[Sol08] Soliman, H. M., "Throttled Lazy Cancellation in Time Warp Parallel Simulation", *SIMULATION*, 84(2-3), pp. 149-160, 2008.

[SPEC10] SPEC, *NAMD SPEC CPU2006 Benchmark Description*, Available on-line at: http://www.spec.org/cpu2006/Docs/444.namd.html, accessed in July, 2010.

[Sri98] Srinivasan, S., and P. F. Reynolds, "Elastic Time", *ACM Transactions on Modeling and Computer Simulatiuon*, 8(2), pp. 103-139, 1998.

[Sta08] Stamatakis, A., and M. Ott, "Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study", *Proceedings of the 3rd International Conference on Pattern Recognition in Bioinformatics*, Melbourne, Australia, LNCS 5265, pp. 424-435, 2008.

[Ste91] Steinman, J. S., "Interactive SPEEDES", *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, LA, pp. 149-158, 1991.

[Ste93] Steinman, J. S., "Breathing Time Warp", *ACM SIGSIM Simulation Digest*, 23(1), pp. 109-118, 1993.

[Ste96] Steinman, J. S., "Discrete-Event Simulation and the Event Horizon Part 2: Event List Management", *Proceedings of the 10th International Workshop on Parallel and Distributed Simulation*, Philadelphia, PA, pp. 170-178, 1996.

[Ste05] Steinman, J. S., "The WarpIV Simulation Kernel", *Proceedings of the 19th International Workshop on Principles of Advanced and Distributed Simulation*, Monterey, CA, pp. 161-170, 2005.

[Sun08] Sun, Y., and J. Nutaro, "Performance Improvement Using Parallel Simulation Protocol and Time Warp for DEVS Based Applications", *Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, Vancouver, Canada, pp. 277-284, 2008.

[Sup00] Suppi, R., F. Cores, and E. Luque, "An Efficient Method for Improving Large Optimistic PDES", *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, San Francisco, CA, pp. 351-357, 2000.

[Tan00] Tan, K. L., and L. J. Thng, "Snoopy Calendar Queue", *Proceedings of the 2000 Winter Simulation Conference*, Orlando, FL, pp. 487-495, 2000.

[Tan05a] Tang, W. T., R. S. M. Goh, and L. J. Thng, "Ladder Queue: An O(1) Priority Queue Structure for Large-Scale Discrete Event SImulation", *ACM Transactions on Modeling and Computer Simulatiuon*, 15(3), pp. 175-204, 2005.

[Tan05b] Tang, Y., K. S. Perumalla, R. M. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko, "Optimistic Parallel Discrete Event Simulations of Physical Systems Using Reverse Computation", *Proceedings of the 19th International Workshop on Principles of Advanced and Distributed Simulation*, Monterey, CA, pp. 26-35, 2005.

[Tan06] Tang, Y., K. S. Perumalla, R. M. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko, "Optimistic Simulations of Physical Systems Using Reverse Computation", *SIMULATION*, 82(1), pp. 61-73, 2006.

[Tay97] Tay, S. C., Y. M. Teo, and S. T. Kong, "Speculative Parallel Simulation with an Adaptive Throttle Scheme", *Proceedings of the 11th International Workshop on Parallel and Distributed Simulation*, Lockenhaus, Austria, pp. 116-123, 1997.

[Tay00] Tay, S. C., and Y. M. Teo, "Probabilistic Checkpointing in Time Warp Parallel Simulation", *Proceedings of the 8th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, San Francisco, CA, pp. 366-373, 2000.

[Tay01] Tay, S. C., and Y. M. Teo, "Performance Optimization of Throttled Time-Warp Simulation", *Proceedings of the 34th Annual Simulation Symposium*, Seattle, WA, pp. 211-218, 2001.

[Thi02] Thies, W., M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications", *Proceedings of the 11th International Conference on Compiler Construction – Joint European Conferences on Theory and Practice of Software*, Grenoble, France, LNCS 2304, pp. 49-84, 2002.

[Tro02] Tropper, C., "Parallel Discrete-Event Simulation Applications", *Journal of Parallel and Distributed Computing*, 62(2), pp. 327-335, 2002.

[Tro03] Troccoli, A., and G. Wainer, "Implementing Parallel Cell-DEVS", *Proceedings of the 36th Annual Simulation Symposium*, Orlando, FL, pp. 273-280, 2003.

[Var07] Varbanescu, A. L., H. Sips, K. A. Ross, Q. Liu, L. K. Liu, A. Natsev, and J. R. Smith, "An Effective Strategy for Porting C++ Application on Cell", *Proceedings of the 2007 International Conference on Parallel Processing*, Xi'an, China, pp. 59-68, 2007.

[Vee02] Vee, V. Y., and W. J. Hsu, "Pal: A New Fossil Collector for Time Warp", *Proceedings of the 16th International Workshop on Parallel and Distributed Simulation*, Washington, DC, pp. 35-42, 2002.

[Wai00] Wainer, G., "Improved Cellular Models with Parallel Cell-DEVS", *Transactions of the Society for Computer Simulation International*, 17(2), pp. 73-88, 2000.

[Wai01] Wainer, G., and N. Giambiasi, "Application of the Cell-DEVS Paradigm for Cell Spaces Modelling and Simulation", *SIMULATION*, 76(1), pp. 22-39, 2001.

[Wai02a] Wainer, G., and N. Giambiasi, "N-dimensional Cell-DEVS Models", *Discrete Event Dynamic Systems*, 12(2), pp. 135-157, 2002.

[Wai02b] Wainer, G., "CD++: A Toolkit to Develop DEVS Models", *Software – Practice and Experience*, 32(13), pp. 1261-1306, 2002.

[Wai06] Wainer, G., "Applying Cell-DEVS Methodology for Modeling the Environment", *SIMULATION*, 82(10), pp. 635-660. 2006.

[Wai08a] Wainer, G., Q. Liu, J. Chazal, L. Quinet, and M. K. Traore, "Performance Analysis of Web-based Distributed Simulation in DCD++: A Case Study across the Atlantic Ocean", *Proceedings of the 2008 Spring Simulation Multiconference: High Performance Computing Symposium*, Ottawa, Canada, pp. 413-420, 2008.

[Wai08b] Wainer, G., R. Madhoun, and K. Al-Zoubi, "Distributed Simulation of DEVS and Cell-DEVS Models in CD++ using Web-Services", *Simulation Modelling Practice and Theory*, 16(9), pp. 1266-1292, 2008.

[Wai09a] Wainer, G., *Discrete-Event Modeling and Simulation: a Practitioner's Approach*, Boca Raton: CRC Press, 2009.

[Wai09b] Wainer, G., and Q. Liu, "Tools for Graphical Specification and Visualization of DEVS Models", *SIMULATION*, 85(3), pp. 131-158, 2009.

[Wai10] Wainer, G., Q. Liu, O. Dalle, and B. P. Zeigler, "Applying DEVS and Cellular Automata Methodologies to Serious Games", *Simulation & Gaming: An Interdisciplinary Journal of Theory, Practice and Research*, 2010, in press.

[Wan06] Wang, X., S. J. Turner, and S. J. E. Taylor, "COTS Simulation Package (CSP) Interoperability – A Solution to Synchronous Entity Passing", *Proceedings of the 20th International Workshop on Principles of Advanced and Distributed Simulation*, Sigapore, pp. 201-210, 2006.

[Wan07] Wang, J., and C. Tropper, "Optimizing Time Warp Simulation with Reinforcement Learning Techniques", *Proceedings of the 2007 Winter Simulation Conference*, Washington, DC, pp. 577-584, 2007.

[Wan09] Wang, J., and C. Tropper, "Using Genetic Algorithms to Limit the Optimism in Time Warp", *Proceedings of the 2009 Winter Simulation Conference*, Austin, TX, pp. 1180-1188, 2009.

[Wes96] West, D., and K. Panesar, "Automatic Incremental State Saving", *ACM SIGSIM Simulation Digest*, 26(1), pp. 78-85, 1996.

[Wie97] Wieland, F., "The Threshold of Event Simultaneity", *Proceedings of the 11th International Workshop on Parallel and Distributed Simulation*, Lockenhaus, Austria, pp. 56-59, 1997.

[Wie06] Wieland, F., L. Hawley, A. Feinberg, M. D. Loreto, L. Blume, J. Ruffles, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson, "The Performance of a Distributed Combat Simulation with the Time Warp Operating System", *Consurrency: Practice and Experience*, 1(1), pp. 35-50, 2006.

[Wil93] Willebeek-Lemair, M. H., and A. P. Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers", *IEEE Transactions on Parallel and Distributed Systems*, 4(9), pp. 979-993, 1993.

[Wil98] Wilson, L. F., and W. Shen, "Experiments in Load Migration and Dynamic Load Balancing in SPEEDS", *Proceedings of the 1998 Winter Simulation Conference*, Washington, DC, pp. 483-490, 1998.

[Wil06] Williams, S., J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The Potential of the Cell Processor for Scientific Computing", *Proceedings of the 3rd Conference on Computing Frontiers*, Ischia, Italy, pp. 9-20, 2006.

[Wil07] Williams, S., J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "Scientific Computing Kernels on the Cell Processor", *International Journal of Parallel Programming*, 35(3), pp. 263-298, 2007.

[Wil08] Williams, S., J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms", *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, pp. 1-14, 2008.

[Wol02] Wolfram, S., *A New Kind of Science*, Champaign: Wolfram Media Inc., 2002.

[Woo94] Wood, K. R., and S. J. Turner, "A Generalized Carrier-Null Method for Conservative Parallel Simulation", *Proceedings of the 8th International Workshop on Parallel and Distributed Simulation*, Edinburgh, UK, pp. 50-57, 1994.

[Xia99] Xiao, Z., B. Unger, R. Simmonds, and J. Cleary, "Scheduling Critical Channels in Conservative Parallel Discrete Event Simulastion", *Proceedings of the 13th International Workshop on Parallel and Distributed Simulation*, Atlanta, GA, pp. 20-28, 1999.

[Yan03] Yang, L., W. Fang, and W. Fan, "Modeling Occupant Evacuation Using Cellular Automata – Effect of Human Behavior and Building Characteristics on Evacuation", *Journal of Fire Sciences*, 21(3), pp. 227-240, 2003.

[Yau03] Yaun, G., C. D. Carothers, S. Adali, and D. Spooner, "Optimistic Parallel Simulation of a Large-Scale View Storage System", *Future Generation Computer Systems*, 19(4), pp. 479-492, 2003.

[You96] Young, C. H., and P. A. Wilsey, "A Distributed Method to Bound Rollback Lengths for Fossil Collection in Time Warp Simulators", *Information Processing Letters*, 59(4), pp. 191-196, 1996.

[You98] Young, C. H., N. B. Abu-Ghazaleh, and P. A. Wilsey, "OFC: A Distributed Fossil Collection Algorithm for Time Warp", *Proceedings of the 12th International Symposium on Distributed Computing*, Andros, Greece, LNCS 1499, pp. 408-418, 1998.

[You99] Young, C. H., R. Radhakrishnan, and P. A. Wilsey, "Optimism: Not Just for Event Execution Anymore", *Proceedings of the 13th International Workshop on Principles of Advanced and Distributed Simulation*, Atlanta, GA, pp. 136-143, 1999.

[Zei76] Zeigler, B. P., *Theory of Modeling and Simulation*, 1st Edition, New York: Wiley-Interscience, 1976.

[Zei84] Zeigler, B. P., *Multifacetted Modelling and Discrete Event Simulation*, Orlando: Academic Press, 1984.

[Zei90a] Zeigler, B. P., *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Orlando: Academic Press, 1990.

[Zei90b] Zeigler, B. P., and G. Zhang, "Mapping Hierarchical Discrete Event Models to Multiprocessor Systems: Concepts, Algorithm, and Simulation", *Journal of Parallel and Distributed Computing*, 9(3), pp. 271-281, 1990.

[Zei93] Zeigler, B. P., and S. Vahie, "DEVS Formalism and Methodology: Unity of Conception/Diversity of Application", *Proceedings of the 1993 Winter Simulation Conference*, Los Angeles, CA, pp. 573-579, 1993.

[Zei96] Zeigler, B. P., Y. Moon, D. Kim, and J. G. Kim, "DEVS-C++: A High Performance Modelling and Simulation Environment", *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, Maui, HI, pp. 350-359, 1996.

[Zei97] Zeigler, B. P., Y. Moon, D. Kim, and G. Ball, "The DEVS Environment for High-Performance Modeling and Simulation", *IEEE Computational Science & Engineering*, 4(3), pp. 61-71, 1997.

[Zei00] Zeigler, B. P., H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, 2nd Edition, London: Academic Press, 2000.

[Zei03] Zeigler, B. P., "DEVS Today: Recent Advances in Discrete Event-Based Information Technology", *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, Orlando, FL, pp. 148-161, 2003.

[Zen04] Zeng, Y., W. Cai, and S. J. Turner, "Batch Based Cancellation: A Rollback Optimal Cancellation Scheme in Time Warp Simulations", *Proceedings of the 18th International Workshop on Principles of Advanced and Distributed Simulation*, Kufstein, Austria, pp. 78-86, 2004.