DISTRIBUTED OPTIMISTIC SIMULATION OF DEVS AND CELL-DEVS MODELS WITH PCD++

By

Qi Liu, B. Eng.

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

Canada

© Copyright 2006, Qi Liu

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

Acceptance of the thesis

Distributed Optimistic Simulation of DEVS and Cell-DEVS Models with PCD++

Submitted by Qi Liu

In partial fulfillment of the requirements for the degree of

Master of Applied Science

Thesis Supervisor

Dr. Gabriel A. Wainer

Chair, Department of Systems and Computer Engineering

Dr. Victor C. Aitken

Carleton University

2006

ABSTRACT

DEVS is a sound formal modeling and simulation (M&S) framework based on generic dynamic system concepts. Cell-DEVS is a DEVS-based formalism intended to model complex physical systems as cell spaces. Time Warp is the most well-known optimistic synchronization protocol for parallel and distributed simulations. This work is devoted to developing new techniques for executing DEVS and Cell-DEVS models in parallel and distributed environments based on the WARPED kernel, an implementation of the Time Warp protocol. The resultant optimistic simulator, called as PCD++, is built as a new simulation engine for CD++, an M&S toolkit that implements the DEVS and Cell-DEVS formalisms. Algorithms in CD++ and the WARPED kernel are redesigned to carry out optimistic simulations using a non-hierarchical approach that reduces the communication overhead. The message-passing paradigm is analyzed using a high-level abstraction called *wall clock time slice*. A two-level *user-controlled state-saving* mechanism is proposed to achieve efficient and flexible state saving at runtime. This mechanism is integrated with both the *copy state-saving* and *periodic state-saving* strategies to realize a hybrid technique that gives simulator developers the full power to dynamically choose the best possible combination of state-saving strategies at runtime. An optimization strategy called *one log file per node* is provided to break the bottleneck caused by file I/O operations. The number of file descriptors consumed in the simulation is upper-bounded and the operational overhead is reduced substantially under this strategy. Different Time Warp optimizations are integrated into PCD++, and their effects are analyzed quantitatively. It is shown that PCD++ markedly outperforms other alternatives, and considerable speedups can be achieved in parallel and distributed simulations, indicating that PCD++ is well-suited for simulating large and complex models.

ACKNOWLEDGEMENTS

I would like to gratefully acknowledge the enthusiastic supervision and constant support of Dr. Gabriel Wainer during this work.

I want to thank specially to Narendra Mehta for his assistance with all types of technical problems – at all times.

Finally, I am forever indebted to my parents and Sara for their understanding, endless patience and encouragement throughout the course of my studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

CSS	Copy State Saving
DEVS	Discrete Event System Specification
FC	Flat Coordinator
GVT	Global Virtual Time
LP	Logical Process
LTSF	Least-Time-Stamp-First scheduling
LVT	Local Virtual Time
M&S	Modeling and Simulation
MPI	Message Passing Interface
MTSS	Message Type-based State Saving
NC	Node Coordinator
P-DEVS	Parallel Discrete Event System Specification
PSS	Periodic State Saving
UCSS	User Controlled State Saving
WCTS	Wall Clock Time Slice

CHAPTER 1 INTRODUCTION

Computer-based modeling and simulation (M&S) have become important tools for analyzing and designing a broad array of complex systems where a mathematical analysis is intractable. A simulation study is often conducted in order to understand the behavior of a system, or to evaluate the effects of various parameters or operating policies. A general framework for M&S [Zei00] is established to define the basic entities and their relationships that are central to the M&S process. The basic entities of the framework include *source system*, *experimental frame*, *model*, and *simulator*. The source system is the real or virtual environment under analysis. It is viewed as the source of data gathered through experimental frames of interest to the modeler. An experimental frame defines the type of data acquired in the system and the specific conditions under which the system is observed or experimented with. A model is an abstract representation of the construction and working of the system of interest. In general, a simulation model is a set of instructions, rules, mathematical equations, or constraints to approximate the behavior of the actual system. A simulator is any computation system that is capable of executing a model to generate its behavior.

There are two primary relations among the basic entities, namely the *modeling relation* (or *validity*) and the *simulation relation* (or *simulator correctness*). The modeling relation links the system under study, the experimental frame in use, and the model for that system. It is concerned with how well the model-generated behavior agrees with the system behavior observed under the conditions as specified by the experimental frame. The simulation relation lies between a model and a simulator. This relation ensures that the simulator executes the model instructions correctly.

Separating the model and simulator concepts brings two major benefits to the framework [Zei00]. First, the same model can be executed with different simulators, allowing portability and interoperability at a high level of abstraction. Secondly, the well-defined separation of concerns allows models and simulators to be independently verified and reused in later combinations with minimal re-verification.

Different modeling techniques have been used to model and simulate different types of systems. The discrete time modeling approach adopts a stepwise execution mode where the

states of all the components are updated synchronously based on the states of the previous time step and the inputs. Continuous modeling and simulation is the classical approach of the natural sciences that often involves difference or differential equations. Finally, discrete-event simulation refers to the modeling technique in which changes to the state of the system can occur only at countable points in time. In this work, we are primarily concerned with the discrete-event M&S approach and the **DEVS** (Discrete Event System Specification) formalism [Zei76, Zei00] that has been proven to be a universal common modeling mechanism for discrete event dynamic systems.

As a sound formal M&S framework based on generic dynamic system concepts, DEVS allows hierarchical and modular construction of models. Tested models can be reused, enhancing reliability and reducing the effort for model development and testing. Since its first formalization, DEVS has been extended into various directions. **Parallel DEVS** or **P-DEVS** [Cho94] is an extension of DEVS that facilitates the handling of simultaneous events. It eliminates the serialization constraints existed in the original DEVS definition and exhibits increased parallelism in parallel and distributed simulations. The **Timed Cell-DEVS** formalism [Wai98] is a combination of the DEVS and Cellular Automata [Wol86] formalisms with explicit timing delays. It defines a way to describe n-dimensional cell spaces as discrete event models, where each cell is represented as a DEVS basic model that can be delayed using different timing constructions.

Parallel and distributed simulation technologies have received increasing interest as simulations become more time consuming and geographically distributed. They address the issues of executing simulations on a computing system containing multiple processors interconnected by a communication network. A parallel or distributed simulation typically consists of a collection of concurrent processes, each modeling a different part of the physical system and executing on a dedicated processor in a sequential fashion. They interact with each other by exchanging time-stamped event messages. The execution of the processes needs to be synchronized to guarantee that correct results will be produced from the concurrent execution of events.

Synchronization is the key to parallel and distributed simulation. It ensures that each process complies with the *local causality constraint* [Fuj00], which requires that events are processed in time stamp order. Errors resulting from out-of-order event execution are referred to

as *causality errors*. Two major classes of synchronization approaches exist: *conservative* (or *pessimistic*) approaches strictly avoid processing events out of time stamp order; *optimistic* approaches detect causality errors during the execution, and provide mechanisms to recover from them via an operation known as *rollback*. Optimistic synchronization allows higher degree of parallelism to be exploited in parallel and distributed simulations. Also, it does not rely on application-specific data to achieve good performance, which is usually the case in conservative synchronization.

Jefferson's Time Warp mechanism [Jef85] is by far the most well-known optimistic synchronization protocol. The WARPED simulation kernel [Rad98] is a configurable object-oriented middleware written in C++ that incorporates the Time Warp mechanism and a variety of optimization algorithms. In our research, the WARPED kernel is used as a middleware abstraction layer for distributed optimistic simulation.

CD++ [Wai01a, Wai02a] is an M&S toolkit that implements Parallel DEVS and Cell-DEVS formalisms. It currently supports both standalone [Rod99] and parallel conservative simulations [Tro03]. CD++ has been used to model and simulate different complex systems in a variety of fields. Based on previous research [Gli04], our work aims to develop new techniques for executing Parallel DEVS and Cell-DEVS models in parallel and distributed environments using the Time Warp mechanism. The resultant optimistic simulator, called as PCD++, is built as a new extension to the CD++ toolkit. Various optimization strategies are proposed and integrated into the PCD++ simulator. It is shown that PCD++ markedly outperforms the previous conservative simulator, and considerable speedups can be achieved in parallel and distributed simulations, indicating that PCD++ is well-suited for simulating large and complex models.

1.1. CONTRIBUTION

This thesis provides a variety of new techniques that allow PCD++ to be used as a high-performance toolkit for distributed optimistic simulation of complex DEVS and Cell-DEVS models. The following is a list of the main contributions of the present work:

- The algorithms for the DEVS processors are redesigned to allow optimistic simulation in parallel and distributed environments. A non-hierarchical approach is employed to reduce the communication overhead and improve the performance.

- It is shown that the message-passing paradigm in PCD++ is different from that in the previous versions of CD++. The algorithms for Cell-DEVS models with transport and inertial delays are adapted to the new asynchronous state transition paradigm.
- Mechanisms are provided for handling rollbacks at virtual time 0 and messaging anomalies that may occur during the simulation.
- The notion of *wall clock time slice* (WCTS) is proposed to provide a high-level abstraction of the simulation process. It greatly simplifies the task of analyzing the complex message exchanges between the DEVS processors involved in the simulation.
- A two-level *user-controlled state saving* (UCSS) mechanism is proposed to achieve efficient and flexible state saving at runtime. It is integrated with the copy state saving to implement a risk-free *message type-based state-saving* strategy that can significantly reduce the number of states saved during the simulation. The UCSS mechanism is also combined with the periodic state saving to realize a hybrid technique that allows dynamic integration of different state-saving strategies at runtime.
- An optimization strategy called *one log file per node* is provided to break the bottleneck caused by file I/O operations. The number of file descriptors consumed in the simulation is upper-bounded and the operational overhead is reduced substantially under this strategy.
- Several enhancements are added to the WARPED kernel [Rad98] to address a number of issues. The kernel state-saving algorithm is modified to implement the UCSS mechanism. The concept of *breakpoint* state is introduced and the kernel state restoration algorithm is revised to deal with messaging anomalies. The fossil collection algorithm is modified to integrate the periodic state-saving strategy into the PCD++ simulator. The kernel rollback operations are enhanced to allow direct handling of variables defined in the processes. Finally, the problem found in the kernel rollback algorithm is fixed to correctly perform secondary rollbacks (a crucial operation for recovering from causality errors).
- Several Time Warp optimizations are integrated into the PCD++ toolkit, including the *one anti-message per rollback* strategy for reducing the overhead of rollback

operations, the *periodic state-saving* strategy for reducing state-saving overhead, and the *lazy cancellation* strategy for exploiting parallelism available within a LP.

1.2. THESIS OVERVIEW

The rest of the thesis is organized as follows:

Chapter 2 introduces the DEVS and Cell-DEVS formalisms and their extensions, and reviews the general concepts in parallel and distributed simulation. A brief survey of existing DEVS-based simulation toolkits is given as well.

Chapter 3 presents the layered software architecture, followed by a description of the main functionalities implemented in the PCD++ toolkit.

Chapter 4 covers the major algorithms employed by the WARPED kernel based on the standard Time Warp mechanism.

Chapter 5 discusses the redesign of the algorithms for the DEVS processors and Cell-DEVS models in PCD++. The new message-passing paradigm is illustrated. Different methods for saving and restoring modifiable variables in PCD++ are presented.

Chapter 6 is concerned with the essential enhancements to the PCD++ and the WARPED kernel to ensure correct and efficient execution of simulations. The notion of WCTS is proposed as an abstraction of the simulation process. Mechanisms are provided to address the problem of asynchronous execution of the processes and to deal with rollbacks at virtual time 0. The UCSS mechanism is proposed to achieve efficient and flexible state saving at runtime. The *one log file per node* strategy is put forward to remove the bottleneck found in the simulations. Algorithms of the WARPED kernel and DEVS processors are enhanced to handle messaging anomalies that may occur during the simulation.

Chapter 7 covers the integration of several Time Warp optimization algorithms into the PCD++ toolkit to improve the performance. The UCSS mechanism is further extended to work with the periodic state-saving strategy in order to reduce the state-saving overhead.

Chapter 8 illustrates the experimental results for measuring the performance of the PCD++ toolkit. The effects of different optimization strategies are discussed quantitatively.

Chapter 9 presents the main conclusions of the thesis and outlines possible future work.

CHAPTER 2 REVIEW OF THE STATE OF THE ART

This chapter provides a review of the state-of-the-art in the field of discrete event modeling and simulation, particularly the techniques for parallel and distributed simulation systems. The DEVS and Cell-DEVS formalisms and their extensions are presented in Section 2.1 and 2.2. Section 2.3 covers the two major synchronization approaches for distributed simulation, namely conservative approaches and optimistic approaches. Finally, a brief survey of existing DEVS-based simulation toolkits is given in Section 2.4.

2.1. DEVS AND PARALLEL DEVS FORMALISMS

In a discrete event simulation, the system being simulated changes state only at discrete points in time, upon the occurrence of an event. Based on general systems theory, the **DEVS** (Discrete Event System specification) formalism [Zei00] provides a framework for the definition of hierarchical models in a modular way. A real system modeled using DEVS can be described as a composition of behavioral (atomic) and structural (coupled) components. A DEVS *atomic model* is defined by:

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

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 sequential states;

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

$\delta_{ext}: Q \times X \rightarrow S$ is the external state transition function, where

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

e is the time elapsed since the last state transition;

$\lambda: S \rightarrow Y$ is the output function;

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

At any time, a DEVS atomic model is in some state s. If no external event occurs, it will remain in state s for ta(s), the lifetime of state s. When the state lifetime ta(s) expires, i.e. the

elapsed time e = ta(s), the atomic model outputs the value $\lambda(s)$ and does an internal state transition to a new state given by $\delta_{int}(s)$. Notice that the output only happens just before the internal state transition. If an external event $x \in X$ occurs before the time ta(s), i.e. the model is in total state (s,e) with $e \leq ta(s)$, it performs an external state transition to state $\delta_{ext}(s,e,x)$. That is, the internal state transition function dictates the model's new state when no events occurred since the last transition, while the external state transition decides the model's new state due to the reception of an external event.

The time advance function can take on any real value including 0 and ∞. A state with zero ta(s) is called a transitory state, whereas a state with ta(s) equal to infinity is a passive state, in which case the system will stay in state s forever unless it is reactivated by an external event.

The DEVS formalism has a well defined concept of system modularity and component coupling to form composite models. A DEVS *coupled model* is formally defined by:

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

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, and the following requirements are imposed on the components, which must also be DEVS models:

For each $d \in D$, $M_d = (X_d, Y_d, S_d, \delta_{int}, \delta_{ext}, \lambda, ta)$ is a DEVS 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 couplings 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 for imminent components.

Direct feedback loops are not allowed, i.e., no output port of a component may be connected to an input port of the same component: $((d, op_d), (e, ip_d)) \in IC$ implies $d \neq e$.

Also, the values sent from a source port must be within the range of accepted values of a destination port, formally expressed as:

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

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

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

A coupled DEVS model can be expressed as an equivalent basic model in the DEVS formalism due to the *closure under coupling* property. Expressing a coupled model as an equivalent basic model captures the means by which the components interact to yield the overall behavior [Zei99a]. Such a basic model can itself be employed in a larger coupled model as required for hierarchical model construction.

Since multiple imminent components can exist at the same time in a coupled model, ambiguity may arise. If an imminent component executes its internal transition and produces an output that is received by another imminent component as an external event, then it is not clear which transition should be done by the receiving component. There are two possible scenarios: executing the external transition first with e = ta(s) and then the scheduled internal transition, or executing the scheduled internal transition first followed by the external transition with e = 0. The DEVS formalism solves this potential ambiguity with the *Select* function, which defines an order over the components so that only one component in the group of imminent models is allowed to have e = 0. The other imminent components are divided into two groups: receivers of the external event from this model, and the rest. Components in the former group will execute their external transition functions with e = ta(s), and those in the latter group will be imminent in the next simulation cycle and may need to use the *Select* function again to decide the execution sequence. This rigid tie-breaking strategy introduces serialization of execution, a potential bottleneck in the simulation system.

The **Parallel DEVS** or **P-DEVS** formalism [Cho94] was proposed to eliminate the restrictions that forced the original DEVS definition to sequential execution. It defines an additional function, called as *confluent transition function*, in the atomic models to handle transition collisions and thus removes the sequential *Select* function for resolving simultaneous events. Hence, all imminent components are allowed to be activated and to send their output to other components. The receiver is responsible for examining the input event and properly interpreting it. Higher degree of parallelism can be exploited in parallel and distributed

simulations with the P-DEVS formalism. As a result, it was chosen as the theoretical foundation for our research.

A P-DEVS atomic model is defined as:

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

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;

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

$\delta_{ext}: Q \times X^b \rightarrow S$ is the external state transition function, where X^b is a set of bags over elements in X;

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

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

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

$Q = \{(s,e) \mid s \in S, \ 0 \le e \le ta(s)\}$ is the total state set, and e is the time elapsed since the last state transition.

Instead of having a single input, basic P-DEVS models employ a bag of inputs to allow the execution of multiple concurrent events. By simultaneously executing the events in the input bag X^b, the external transition function can combine the functionality of a number of external transitions into a single one. A second difference is the addition of a confluent transition function (δ_{con}) in the model definition. This function decides the next state of the model in cases of collision between external and internal functions [Zei00]. In virtue of the confluent transition function, modelers have a flexible way to define the appropriate behavior for each atomic model in the simulation system.

Consequently, the *Select* function is removed from the definition of P-DEVS coupled models, resulting in a model definition as follows:

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

The specifications for the set of input and output events (X and Y) and couplings (EIC, EOC, and IC) follow the definitions of DEVS coupled models as presented earlier. The basic components (D and M_d) are specified by the P-DEVS atomic model definition.

As we can see, P-DEVS coupled models are specified in the same way as in classic DEVS except that the *Select* function is omitted. While this is an innocent-looking change, its semantics are much different. They differ significantly in how imminent components are handled. In P-DEVS, there is no serialization among the imminent computations – all imminent components generate their outputs which are then distributed to their destinations using the coupling information [Zei00].

2.2. TIMED CELL-DEVS AND PARALLEL CELL-DEVS FORMALISMS

Cellular Automata formalism [Wol86] has been widely used to describe real systems that can be represented as cell spaces. A Cellular Automata is an infinite regular n-dimensional lattice whose cells can take one finite value. These cells evolve by executing a global transition function that updates the state of every cell in the space. The behavior of this global function depends on the results of a local function executed in each cell in discrete time steps. Conceptually, these local functions are computed synchronously and in parallel, using the state values of the present cell and a finite set of neighboring cells (called as the *neighborhood* of the cell). However, this discrete time paradigm constrains the precision and efficiency of the simulated models. Furthermore, it is usual that several cells do not need to be updated in every step, wasting computation time [Wai01b].

The **Timed Cell-DEVS** formalism [Wai98] solves these problems by using the DEVS paradigm to define a cell space where each cell is represented as a DEVS atomic model. Moreover, it adopts delay constructions and defines them as a functional component of the model defining each cell. Hence, it is possible to define a discrete event cell space with explicit delays. Each cell can use one of two kinds of delay constructions with different semantics, namely *transport* and *inertial* [Gia76]. Transport delay allows one to model a variable commuting time for each cell with anticipatory semantics, while inertial delay introduces preemptive semantics to generate more complex temporal behaviors.

A Cell-DEVS atomic model can be formally defined as [Wai01b]:

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

where

X is the set of external input events;

Y is the set of external output events;

I represents the definition of modular model interface;

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};

d is the delay for the cell;

δ_{int} is the internal transition function;

δ_{ext} is the external transition function;

τ is the local computation function;

λ is the output function; and

D is the state's duration function.

The cell's modular interface (I) is composed of a fixed number of ports, each connected with a neighbor. A cell can use the input and output ports to interchange data with other neighboring cells as well as models outside the cell space. The input values are used to compute the future state of the cell by evaluating the local computation function τ. If the resultant future state is different from the cell's present value, the new state value will be sent to all the neighboring cells. Otherwise, the cell remains quiescent and no output will be scheduled. Also, the new state value is transmitted only after the completion of the delay time given by the delay function associated with the cell. Finally, the DEVS transition (δ_{int}, δ_{ext}) and output (λ) functions are included in each cell.

The Cell-DEVS atomic models can be coupled with others, forming a cell space that consists of multiple cells interconnected by the neighborhood relationship. Further, the cell space itself can be integrated with other Cell-DEVS or DEVS models. A cell space is constructed by defining a coupled Cell-DEVS model as follows:

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

where

X_{list} is the list of input coupling;

Y_{list} is the list of output coupling;

I represents the definition of 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; and

Select is the tie-breaking function for simultaneous events.

The cell space (C) is a coupled model defined as an array of Cell-DEVS atomic models of fixed size ($t_1 \times \ldots \times t_n$). The neighborhood set (N) is defined as a set of n-tuples giving the relative position between the origin cell and the surrounding neighbors. The border of the cell space is specified by the border cells (B). If $B = \{\varnothing\}$, i.e. the border is *wrapped*, every cell in the space will have the same behavior. The cells in one border are connected with those in the opposite one using the inverse neighborhood relationship. Otherwise, the border set is not empty and the cells in it will have a different behavior from the others in the cell space. The Z function allows definition of the coupling between cells in the model. It translates the outputs of i^{th} output port in cell C_a into values for the i^{th} input port in cell C_b. The *Select* function serves the same purpose as in the classic DEVS formalism. It defines an order of execution for the case where simultaneous events occur.

As in the DEVS formalism, the use of the *Select* function can lead to serialization and incorrect execution when models are considered to be executed in parallel [Wai99]. Moreover, only one input is allowed for each input port in the Timed Cell-DEVS paradigm, disallowing zero-delay transitions and multiple simultaneous events from external DEVS models. The **Parallel Cell-DEVS** formalism [Wai00b] is an extension of the Timed Cell-DEVS formalism to remove these restrictions. Several important propositions are presented in [Wai00b], as summarized below:

(1) Parallel Cell-DEVS models are equivalent to Parallel DEVS models.

(2) Closure under coupling for Parallel Cell-DEVS models also holds. That is, a coupled Parallel Cell-DEVS model is equivalent to a basic Parallel Cell-DEVS model.

An implementation of the Parallel Cell-DEVS was presented in [Tro03], in which the author extended the CD++ toolkit [Rod99] to execute Parallel DEVS and Cell-DEVS models in distributed environments based on conservative synchronization mechanisms.

2.3. PARALLEL AND DISTRIBUTED SIMULATION

In parallel and distributed simulations, the whole simulation task is divided into a set of smaller subtasks with each executed on a different processor or node. Hence, the simulation system is viewed as a collection of concurrent processes, each modeling a different part of the physical system and executing on a dedicated processor in a sequential fashion. These processes communicate with each other by exchanging time-stamped event messages. The subtask executed by each process consists of a sequence of event computations, where each computation may modify the state of the process and/or schedule new events that need to be executed on the present process or on other processes. Unlike sequential simulations, which ensure that all events generated in the whole simulation are executed in time stamp order, parallel and distributed simulations need some mechanism to guarantee that the same results as the sequential execution will be produced from the concurrent execution of events.

Fujimoto defines conditions for correct simulation as follows [Fuj00]:

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

One of the most challenging problems of parallel and distributed simulation is synchronization, which ensures the local causality constraint to be satisfied in the simulation system. Two major schools of thought have been shaped to address the synchronization problem: conservative schemes and optimistic schemes. Conservative schemes adopt a *block-resume* strategy to keep processes synchronized. Under such schemes, the synchronization is done by globally controlling the execution order and run lengths of the individual processes to strictly avoid executing events out of time stamp order. However, conservative schemes sacrifice parallelism to a degree due to the continuous avoidance of time ambiguities. On the other hand, optimistic schemes take a *lookahead-rollback* strategy where causality errors are detected during the execution, and mechanisms are provided to recover from them.

2.3.1. Conservative parallel discrete event simulation

Conservative synchronization approaches were introduced in the late 1970s by R. E. Bryant [Bry77], K. M. Chandy and J. Misra [Cha78]. Since then several variations, improvements, and optimizations have been developed. All of them are based on the principal idea that causality violations are strictly avoided.

The notion of *lookahead* is essential to conservative synchronization mechanisms. It gives the smallest time stamp of the potential new events that a process can schedule in the future. The lookahead information is exchanged among the LPs via null messages. Based on the lookahead collected from all the processes, each LP can derive a lower bound on the time stamp (LBTS) of messages that it may later receive. Armed with this information, the LP can then determine which events can be safely processed. However, the resulting cycles of null messages could severely degrade simulation performance.

Although conservative synchronization algorithms have advanced to a state where they are viable for use in real-world application, optimistic approaches offer two important advantages over conservative techniques [Fuj03]:

(1) The optimistic approaches can exploit higher degree of parallelism available in the simulation. Usually, the conservative approaches tend to be overly pessimistic, and force sequential execution when it is not necessary.

(2) The conservative approaches generally rely on application-specific information to determine which events are safe to process. While optimistic mechanisms can execute more efficiently if they exploit such information, they are less reliant on the application for correct execution, allowing more transparent synchronization and simplifying software development.

On the other hand, optimistic approaches may require computations with higher overhead than conservative ones, degrading the system performance to a certain extent.

2.3.2. Optimistic parallel discrete event simulation

Jefferson's Time Warp mechanism [Jef85] is the first and remains the most well-known optimistic synchronization protocol that uses *Virtual Time* to model the passage of time in the simulation. The simulation is executed via several ***Time Warp processes*** interacting with each other by exchanging time-stamped event messages. Each process maintains a *Local Virtual Time*

(LVT) that advances in discrete steps as each event is executed on the process. Time Warp processes execute their own part of the simulation speculatively without explicit synchronization. A causality error arises if a process receives an event with timestamp less than its LVT. Such events are referred to as *straggler events*. Upon the arrival of a straggler event, the process recovers from the causality error by undoing the effects of those events speculatively executed during previous computations, an operation called as *rollback*. Due to the nature of optimistic execution, erroneous computations on a Time Warp process can spread to other processes via false messages. These false messages are cancelled during rollbacks by virtue of *anti-messages*. When a process sends a message, an anti-message is created and kept separately. The anti-message has exactly the same format and content as the positive (original) message except in one field, a negative flag. Whenever an anti-message meets its counterpart positive message, they immediately annihilate one another, hence cancelling the positive one.

The Time Warp protocol consists of two parts: the *local control mechanism* and the *global control mechanism*. The local control mechanism is provided in each Time Warp process to implement the rollback operations. In order to make rollback possible, three structures are maintain in each process: an *input queue* containing all recently arrived messages sorted in virtual receive time order, an *output queue* containing negative copies (i.e. anti-messages) of the messages the process has recently sent in virtual send time order, and a *state queue* containing saved copies of the process's recent states. Two major actions are performed in case of a rollback. First, the state of the process is restored to the last state saved before the virtual time indicated by the straggler's timestamp. Secondly, the process sends anti-messages in its output queue to their receivers to cancel the positive ones generated in previous false computations. An anti-message causes a rollback at its destination if its timestamp is less than the LVT of the receiving process, just as a positive straggler would. During this rollback, more anti-messages may be sent to other processes, resulting in a cascade of rollbacks in the simulation system.

The global control mechanism is concerned with such global issues as space management, I/O operations, and termination detection. It requires a distributed computation involving all of the processes in the system. The central concept of the global control mechanism is *Global Virtual Time* (GVT), a property of an instantaneous global snapshot of the system at wall clock time T, which is defined as follows [Fuj00]:

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

One characteristic of GVT is that it never decreases, despite the fact that individual local virtual clocks roll back frequently [Fre02]. Hence, GVT serves as a floor for the virtual time of any future rollback that might occur. Any event occurred prior to GVT cannot be rolled back and may be safely committed. Therefore, messages in the input and output queues whose timestamp is less than GVT can be discarded. Similarly, all but the last saved state older than GVT can be reclaimed for each process. Destroying information older than GVT is done via an operation known as *fossil collection*. Furthermore, I/O operations with virtual time less than GVT can be irrevocably committed with safety.

In Time Warp systems, the global control mechanism must estimate GVT every so often. How frequent the estimation should be is a trade-off: high frequency allows faster response time and better space utilization, but it also imposes an overhead on the processor and communication network, slowing down the simulation system.

Many refinements have been proposed to enhance Jefferson's original Time Warp mechanism, either for reducing the operational overhead or for exploiting more parallelism than is available in the basic protocol. Details on the Time Warp protocol are covered in Chapter 4, while several optimization algorithms are discussed later in Chapter 7. Other advanced optimistic techniques can be found in [Fuj00].

2.4. DEVS-BASED SIMULATION TOOLKITS

Based on previous studies [Gli04], we give a brief survey on the existing DEVS-based toolkits that have been implemented by different researchers as follows:

- ADEVS [Nut06] supports the construction of discrete event models based on a variant of the P-DEVS formalism. It includes support for dynamic structure models based on the Dynamic DEVS formalism [Uhr01a].
- DEVS-C++ [Zei96] is a high performance simulation environment that allows portability of models across platforms at a high level of abstraction. It uses a set of C++ classes, called as containers, to realize serial and parallel simulations.

- DEVS-Scheme [Zei93] is a knowledge-based environment implemented in Scheme for discrete-event model construction and simulation. It allows combining symbolic and hierarchical, modular discrete-event modeling approaches.
- DEVS/CORBA [Zei99a] is a runtime infrastructure on top of CORBA middleware to support distributed simulation of DEVS components. It is possible to embed DEVS/CORBA in a larger network-centric environment to provide a combination of graphical process modeling, discrete-event simulation, animation, activity-based costing, and optimization functions.
- DEVS/HLA [Zei99b] is an HLA-compliant M&S environment implemented in C++ that supports high level model construction. It greatly simplifies the underlying programming details required to establish and participate in an HLA federation.
- DEVS/Grid [Seo04] is an M&S framework implemented using Java and Globus toolkit for Grid computing infrastructure.
- DEVSCluster [Kim04] is a CORBA-based, multi-threaded distributed simulator implemented in Visual C++. It transforms a hierarchical DEVS model into a non-hierarchical one to ease the synchronization of the distributed simulation.
- DEVSJAVA [Sar98] is a DEVS-based simulator that supports high-level modeling.
- GALATEA [Dav00] is offered as a family of languages to model multi-agent systems to be simulated in a DEVS, multi-agent platform.
- JDEVS [Fil02] is an M&S environment that enables discrete-event, general purpose, object-oriented, component-based, GIS (Geographic Information System) connected, collaborative, visual simulation model development and execution.
- JAMES [Uhr01b] is a Java-based simulation environment that allows the modeler to describe agents and their environment as situated automata.
- PyDEVS is a simulator developed in ATOM3 [Del02], a tool for multi-paradigm modeling. DEVS models are constructed using the ATOM3-DEVS tool, which generates Python code to be executed with the PyDEVS simulator.
- PowerDEVS [Kof03] is an M&S toolkit developed in C++ for hybrid systems. Atomic DEVS models can be graphically coupled in hierarchical block diagrams to create complex systems.

- SimBeans [Pra99] is a discrete-event simulation framework based on DEVS and the JavaBean component model.
- DEVS/P2P [Che04] is an M&S framework based on P-DEVS formalism and Peer-to-Peer message communication protocol. It uses a customized DEVS simulation protocol to achieve decentralized inter-node communication. Simulators are synchronized by themselves without involving a coordinator.
- DEVS/RMI [Zha06] is a DEVS-based system that provides a fully dynamic and re-configurable runtime infrastructure for handling load balancing and fault tolerance in distributed simulations. It reduces the overhead associated with common middleware solutions by using the native support of Java RMI to achieve the synchronization of local and remote simulators.
- CD++ [Rod99, Wai02a, Tro03] is an M&S toolkit developed in C++ that implements the original and Parallel DEVS and Cell-DEVS formalisms. It supports both standalone and parallel conservative simulations. This toolkit has been extended in our research to realize distributed optimistic discrete-event simulations based on the Time Warp mechanism.

CHAPTER 3 SOFTWARE ARCHITECTURE

Aiming at running simulations in parallel and distributed environments using the Time Warp protocol, our simulator, PCD++, was developed based on previous work presented in [Gli04]. The PCD++ simulator adhered to the same layered design as used in the parallel conservative simulator [Tro01]. The software architecture is first presented in Section 3.1, followed by a more detailed discussion of the layers that are the focus of our research in Section 3.2.

3.1. LAYERED ARCHITECTURE

As shown in Figure 1, the PCD++ simulator employs a layered architecture, where each layer only depends on the layers below it and not above. The following is a brief introduction to each of the layers.

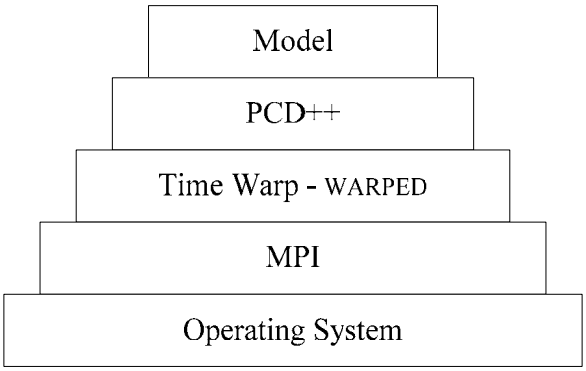


Figure 1. Layered architecture of the PCD++ optimistic simulator [Gli04]

At the bottom of the architecture is the operating system. Since the Linux Operating System is rapidly becoming a de facto standard platform for high-performance parallel and distributed computing, it is used as the underlying platform on which our simulator runs.

Above the Operating System lies the Message Passing Interface (MPI), a standard specification of message-passing library for high-performance communications on both massively parallel machines and on workstation clusters. MPI, along with the operating system, provides the communication infrastructure for the PCD++ simulator. There are both freely available and vendor-supplied MPI implementations for use: MPICH [Gro96] is an open-source portable implementation of MPI that provides a vehicle for MPI implementation research and for

developing parallel and distributed applications, while Scali MPI Connect™ is a fully integrated MPI solution that enables users to take advantage of the leading interconnect technologies, such as the Myrinet™ technology for clusters, to build high performance applications.

Originally designed and developed at the University of Cincinnati, the WARPED simulation kernel [Rad98] is a configurable middleware that implements the Time Warp mechanism and a variety of optimization algorithms.

On top of the WARPED kernel, the PCD++ simulator implements the Parallel DEVS and Cell-DEVS formalisms and provides the framework for building and executing DEVS and Cell-DEVS models in distributed environments using the Time Warp protocol.

The topmost layer represents the DEVS and Cell-DEVS models built in the CD++ simulation environment and executed by the PCD++ simulator.

3.2. MAJOR FUNCTIONALITIES OF THE PCD++ AND WARPED LAYERS

As our research is targeted at the PCD++ and the WARPED layers, this section highlights the major functionalities at these layers to give a broad overview of the capabilities and important algorithms implemented in the PCD++ toolkit. Figure 2 shows a closer look at these two layers based on previous researches as presented in [Rad98, Mar99, Wai02a]. More detailed description on these modules is provided in the following subsections.

<table>
<tr><td rowspan="3">PCD++ Layer</td><td>Utilities</td><td>Specification Language</td><td>Quantization Facility</td><td>I/O Facility</td></tr>
<tr><td rowspan="2">Simulation Administration</td><td colspan="2">Modeling Framework</td><td>Logging Facility</td></tr>
<tr><td colspan="2">Simulation Framework</td><td>Partition Facility</td></tr>
<tr><td rowspan="5">WARPED Layer</td><td colspan="4">Application Interface</td></tr>
<tr><td colspan="4">Scheduling</td></tr>
<tr><td>Rollback Facility</td><td>GVT and Fossil Collection</td><td>Memory Management</td><td>Time Warp Optomizations</td></tr>
<tr><td>Event Management</td><td>State Management</td><td>File Management</td><td>Time Management</td></tr>
<tr><td colspan="4">Communication Management</td></tr>
</table>

Figure 2. Major functionalities of the PCD++ and the **WARPED** layers

Following is a brief summary of the majority of our work in terms of these modules:

(1) At the PCD++ layer, the message-processing algorithms provided in the Simulation Framework are redesigned to carry out distributed optimistic simulations. The state transition logic in the Modeling Framework is modified to ensure correct computation in Cell-DEVS models. Also, the Logging Facility is optimized to reduce the overhead of file I/O operations.

(2) At the WARPED layer, a flexible user-controlled state-saving mechanism is implemented in the State Management module. Also, the concept of *breakpoint* state is introduced and the state restoration algorithm is modified to handle messaging anomalies that can happen during the simulation. The fossil collection algorithm is revised in the GVT and Fossil Collection module to integrate the periodic state-saving strategy into the PCD++ simulator. The Rollback Facility is enhanced to allow direct handling of variables defined in the processes during rollbacks. Finally, the Time Warp Optimizations module is modified to incorporate three different optimization strategies into the PCD++ simulator.

3.2.1. The WARPED layer

The WARPED kernel provides services to the application above it for building Time Warp processes (called as *simulation objects*) based on Jefferson's definition. In the WARPED kernel, simulation objects are organized into groups called "clusters" [Rad98]. Adding the extra *cluster* or *partition* level is the result of partitioning the simulation objects amongst the available physical processors [Low99]. The clustering levels used in the WARPED kernel are shown in Figure 3.

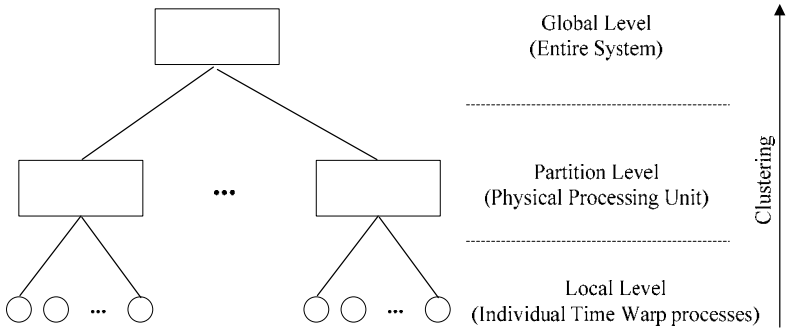


Figure 3. The clustering levels in the WARPED kernel

Individual simulation objects are located at the lowest level, implementing the Time Warp local control mechanism. Above it is the partition level associated with each physical processor. Simulation objects mapped on a physical processor are grouped by an entity called as *logical process* (LP). Note that the simulation objects within a LP operate as Time Warp processes, even though they are grouped together, they are not coerced into synchronizing with each other [Rad98]. The top level is the entire system consisting of multiple partitions that operate collectively to implement the Time Warp global mechanism.

The major functionality modules in the WARPED kernel are described as follows:

- **Application Interface**

The WARPED kernel presents an abstract definition of events, states, and simulation objects to the applications [Mar99]. Basic functions are provided for sending and receiving events between simulation objects. Different types of simulation objects with unique definitions of states can be constructed by deriving from the WARPED kernel. Control is passed between the application and the kernel through cooperative function calls. The application is responsible for initializing the simulation objects and defining the activities of each simulation object. The Time Warp mechanism and other facilities are made available through inheritance, which allows transparent access and is restrictive enough to hide kernel internal operations from the user.

- **Scheduling**

The issue of scheduling is not given recognition in the Time Warp mechanism, and the choice of policy to govern scheduling is left entirely to an implementation that requires it. The WARPED kernel [Mar99] takes on the most straight-forward and intuitive approach to scheduling simulation objects based on their LVT, an approach called *least time stamp first* (LTSF) scheduling. A *LTSF scheduler* is created on each processor at the partition level for scheduling the simulation objects mapped on the corresponding LP.

- **Rollback Facility**

The kernel rollback facility [Mar99] operates transparently to the application. Causality errors are detected between event executions for each simulation object. Once a causality error is found, normal execution is suspended and rollback operations are performed immediately in the kernel. After the rollbacks, the erroneous data resulting from speculative computations is recovered and forward execution is resumed. Implementing the rollback facility is demanded by the Time Warp local control mechanism.

- **GVT and Fossil Collection**

An entity called *GVT manager* [Mar99] is created on each LP to implement the Time Warp global control mechanism, namely detecting the termination of the simulation, performing I/O operations, and measuring the progress of the computation so that the memory for states and events with timestamps older than GVT can be reclaimed. Two different algorithms are provided in the GVT and Fossil Collection facility: the Mattern's GVT estimation algorithm [Mat93] and the passive response GVT or pGVT algorithm [Dso94]. One attractive feature of the pGVT algorithm is that the loss of a control message does not have a significant impact on the GVT calculation and the decision-making process is completely distributed [Low99].

- **Memory Management**

Five alternatives for dynamic memory management are available [Mar99], including the operating system's default memory allocator, the Global Memory Manager that implements the CustoMalloc [Gru93] algorithm, the Buddy Memory Manager based on Knuth's buddy system [Knu73], the Segregated Storage Allocator that combines the basic ideas of buddy system and the first fit algorithm, and the Brent's implementation of the first fit allocation strategy.

- **Event Management**

The basic properties of the abstract event definition (*BasicEvent*) include the virtual send and receive time of the event, the identities of the sender and the receiver, a sequence counter, a flag to mark the event as an anti-message, and a flag to label the event as processed or not [Mar99]. The application can define different types of events by deriving from the *BasicEvent*. Events are organized in the input and output queues in the kernel. While an output queue is created for each simulation object, a single input queue is shared by all the simulation objects mapped on a LP. Event management is greatly simplified by organizing all incoming events in a single input queue that is under the control of the LTSF scheduler on that processor.

- **State Management**

The abstract state definition (*BasicState*) has three basic properties [Mar99]: the virtual time when the state is saved for a simulation object; a pointer, called as *inPos*, to the input event executed just before saving the state; another pointer, called as *outPos*, to the output event most recently sent by the simulation object. Each simulation object has its own current state that is susceptible to modifications during the execution of events. An object's current state is saved regularly in its state queue that is managed by the associated *state manager*. The state manager

provides functionalities for saving and restoring states, and governs the state-saving policy with respect to when and how the states should be saved for the simulation object. Two types of state managers are provided that implement the *copy state saving* (CSS) and *periodic state saving* (PSS) strategies respectively. The latter type further includes: fixed-sized checkpoint interval strategy and Lin's [Lin93], Palaniswamy's [Pal93], Fleischmann's [Fle95], and Ronngren's [Ron94] adaptive state-saving algorithms. Users can select one of these types for use at compile time.

- **File Management**

Files may be opened for output during the simulation. Output data is wrapped in objects of type *FileData* and saved temporarily in a structure called *file queue* [Mar99]. A *FileData* object contains three values: the actual output data, the length of the data, and the virtual time at which the data should be output. A file queue is created for each physical file in the owning simulation object. *FileData* objects with virtual time older than GVT are committed automatically by the kernel GVT and Fossil Collection facility. The kernel also provides functions for removing erroneous data from the file queues during rollbacks.

- **Time Management**

By default, WARPED has a simple notation of integer time written in the *hours:minutes:seconds:milliseconds* format [Mar99]. There is no concept of negative virtual time in the kernel, and any virtual time less than zero is deemed as invalid.

- **Communication Management**

There are two types of communications in the simulation [Mar99]: message-passing between simulation objects residing on different processors (*remote* or *inter-LP communications*), and message-passing between simulation objects on the same processor (*local* or *intra-LP communications*). Inter-LP communications are realized using a group of *communication managers* over MPI. A communication manager is created on each LP at the beginning of the simulation. Intra-LP communications are done via direct function invocations, which is much faster than MPI communications. Since both the sender and the receiver reside in the same address space, the event is directly inserted into the receiver's input queue. Therefore, simulation objects that communicate frequently should be placed within the same partition.

- **Time Warp Optimizations**

A variety of Time Warp optimizations are provided by WARPED to optimize almost every aspect of operations in the kernel [Mar99], including fixed-sized and dynamic message aggregation [Che98] algorithms for minimizing inter-LP communication overhead; static and adaptive polling [Sha99] algorithms for optimizing the message reception behavior; one anti-message per rollback strategy [Mar99] for reducing the number of anti-messages during rollbacks; lazy and dynamic cancellation algorithms [Lin91] for exploiting parallelism available within a Time Warp process; and algorithms for adjustment of runtime parameters using external agents [Rad97] to reduce the operational overhead.

3.2.2. The PCD++ layer

The major functionality modules at the PCD++ layer are described as follows:

- **Modeling Framework**

The modeling framework represents the behavior of the DEVS and Cell-DEVS models [Wai02a]. A hierarchy of classes, rooted at *Model*, is defined to implement the model theoretical definitions. Modelers can define their models by deriving from the modeling framework. For P-DEVS models, the model logic needs to be provided in classes inherited from the abstract atomic model definition in the framework. After defining the atomic models, the coupled models can be specified using the built-in specification language. For Cell-DEVS models, the modeler can rely solely on the capabilities provided by the specification language (no programming is needed). Currently, the framework supports the definition of Cell-DEVS models with transport and inertial delays. The properties of the cell space can be fully specified by the language as well.

- **Simulation Framework**

The simulation framework implements the optimistic simulation mechanisms in line with the DEVS theory [Wai02a]. It consists of a hierarchy of classes, rooted at *Processor*, defining different types of simulation objects. That is, the ***PCD++ processors*** are concrete implementations of simulation objects to realize the abstract DEVS processors. The simulation framework is loosely coupled with the modeling framework. The model logic defined in entities of the modeling framework is executed in a standardized fashion by their counterparts of the simulation framework according to the DEVS formalism.

- **Simulation Administration**

The simulation is managed by several administrators, including [Wai02a]: a *main administrator* that takes care of the bootstrap operations of the simulation; a *model administrator* that keeps a registry for all the models defined in the simulation and provides a lookup service to retrieve model information at runtime; a *processor administrator* that manages all the PCD++ processors created in the simulation; and a *local transition administrator* that registers and evaluates the local transition and port-in rules defined for Cell-DEVS models.

- **Specification Language**

A built-in specification language is provided for defining DEVS and Cell-DEVS models [Wai02a]. The model coupling information and all properties of the cell space can be coded in simple rules with a few parameters. Besides, the language provides numerous operations, functions, and constants, allowing complex models to be defined through a very simple set of procedures and greatly facilitating the model development process.

- **Utilities**

Various utilities can to be used internally by the toolkit itself and externally by the modelers [Wai02a]. The internal utilities include a parser for the specification language and tools for model verification; the external utilities comprises such tools as file format converters, log file analyzers, rule and partition debuggers, and visualization tools to show the simulation results.

- **Partition Facility**

Partition of models (and the corresponding PCD++ processors) is achieved using a simple text file that defines the mapping of atomic models to the machines [Wai02a]. Models are divided at the lowest level of the model hierarchy, allowing flexible and fine-grained partitions.

- **Logging Facility**

Log files are created during the simulation [Wai02a]. Each PCD++ processor can log the messages received during the simulation in a human readable format. The log files can be use by a variety of tools for debugging and visualization purposes. Users can also choose to log only a subset of messages, allowing less storage consumption, faster execution, and greater flexibility.

- **I/O Facility**

A number of files are opened during the simulation for defining the models and initial values, declaring the external events, specifying the partitions, sending output events to the

environment, and generating debugging information [Wai02a]. Although these I/O operations are mainly done in the file system, the toolkit can be adapted to permit I/O via other interfaces like serial ports, network, and USB connections in future versions.

- **Quantization Facility**

Based on the theory of quantized DEVS models [Zei98a, Zei98b], PCD++ provides quantization facility for Cell-DEVS models. A quantized version of CD++ toolkit was introduced in [Rod99] and experimental results were presented in [Wai00a]. Two types of quantization techniques were provided, including the uniform and the non-uniform (intervals) quantizer [Dab03]. Quantization allows faster execution with decreased number of active cells and message exchanges at the cost of introducing errors in the simulation results.

CHAPTER 4 BASIC CONTROL MECHANISMS IN THE WARPED KERNEL

This chapter covers the kernel mechanisms based on a set of standard settings, including LTSF scheduling, copy state saving, passive response GVT (pGVT) algorithm, and aggressive cancellation. The assumptions made by the WARPED kernel are first presented in Section 4.1, followed by a discussion of the kernel control mechanisms in Section 4.2. Several flaws in the kernel algorithms are discussed in Section 4.3.

4.1. ASSUMPTIONS OF THE WARPED KERNEL

The WARPED kernel makes a number of assumptions with respect to its execution environment. These assumptions are summarized based on the documentation of the WARPED kernel [Mar99]:

(1) Reliable communications over First-in, First-out (FIFO) channels. Jefferson's definition [Jef85] did not assume this order preservation in the communication medium. However, the WARPED kernel relies on this property to simplify the implementation of the scheduling, rollback, and GVT and fossil collection facilities.

(2) Predefined ordering of simultaneous events. The kernel orders input events with the same timestamp based on the identities of their receivers. Input events to the same receiver at the same virtual receive time are ordered by their *arriving order* (i.e. the sequence they are received by the receiver). Output events from the same sender at the same virtual send time are ordered by their *sending order* (i.e. the sequence they are sent out by the sender).

(3) The virtual send time of each message must be *less than or equal to* its virtual receive time. An event with the same virtual send and receive time are executed *instantaneously* in virtual time by the receiver.

(4) The timestamp of each event in a process must be *less than or equal to* the timestamp of the next event in that process. Simultaneous events are ordered by the rules specified in Assumption 2 as described above.

(5) There is no concept of negative simulation time in the kernel.

(6) Currently, each LP is associated with a UNIX heavy-weight process and is assigned

to a dedicated processor. This may change in the future to allow a multithreaded implementation where each LP is associated with a light-weight thread [Mar99].

(7) Rollback is completely transparent to the process being rolled back. To fulfill this requirement, the kernel carries out rollbacks *between* event executions. That is, rollbacks can happen only after the execution of an event has finished and before the execution of the next event is commenced.

4.2. KERNEL CONTROL MECHANISMS

The kernel control mechanisms include two parts: (1) rollback mechanisms realized by individual simulation objects at the local level, and (2) GVT calculation and fossil collection mechanisms implemented by the LPs at the global level. These mechanisms are briefly discussed in the following subsections.

4.2.1. Rollback mechanisms and cascaded rollback process

As required by the Time Warp local control mechanism, rollbacks are performed by the simulation objects in the WARPED kernel. Rollback operations are triggered by an incoming straggler or anti-message when it is inserted into a simulation object's input queue. There are two types of rollbacks: *primary rollback* triggered by a straggler message and *secondary rollback* as the result of receiving an anti-message.

- **Primary rollback**

The runtime representation of a simulation object before primary rollback is shown in Figure 4, where input events are depicted as blocks with *receive time*; output events are shown as blocks with *send time*. States are shown as circles with three values: the recorded LVT of the simulation object, a pointer (*inPos*) to the input event just executed, and another pointer (*outPos*) to the last message sent by the simulation object. Let's denote the states as S(12), S(21), and S(35). The diagram shows that this simulation object has executed events with receive time 12, 21, and 35, notated as E(12), E(21), and E(35). Accordingly, the simulation object's LVT is set to 35. Now a straggler E(18) arrives, resulting in a primary rollback on this simulation object. The receive time of the straggler is referred to as *rollback time*. Here, the rollback time is 18.

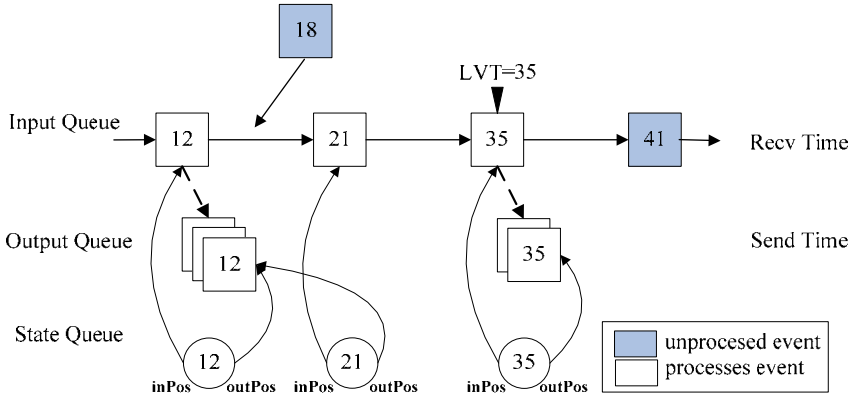


Figure 4. Runtime representation of a simulation object

Shown in Figure 5, the kernel operations for primary rollback are described as follows:

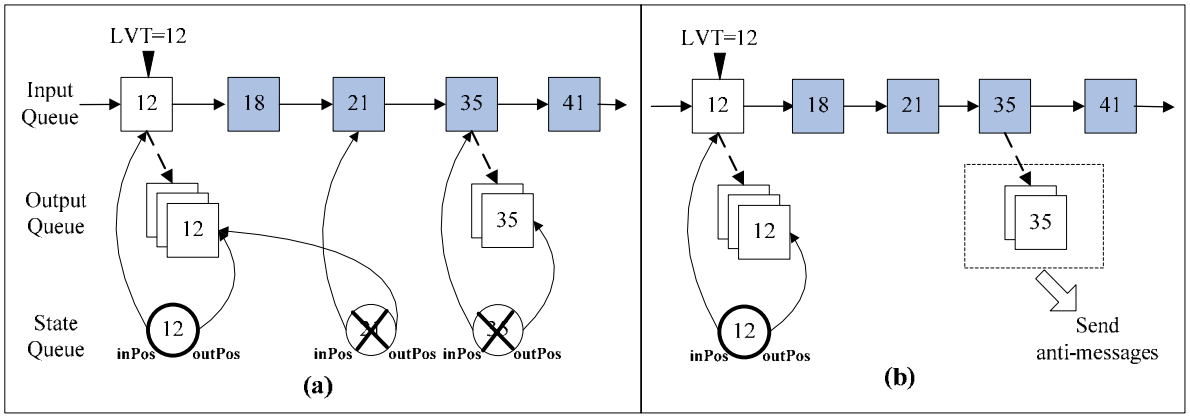


Figure 5. Kernel operations for primary rollback

(1) Insert the straggler, i.e. E(18), into the input queue.

(2) Undo the input events after the straggler, i.e. E(21) and E(35) in Figure 5(a).

(3) Restore the simulation object's current state to the last state with LVT less than the rollback time. Hence, the object's current state is an exact duplicate of S(12).

(4) Remove all saved states after S(12) from the state queue.

(5) Reset the object's LVT to the LVT in its current state, i.e. 12.

(6) Rollback the simulation object's file queues, if any. This is done by removing all data with virtual time greater than or equal to the rollback time from the queues.

(7) Send output messages with send time greater than or equal to the rollback time as anti-messages to their receivers, as shown in Figure 5(b).

After these operations, the kernel resumes normal execution forward again.

- **Secondary rollback**

Depending on whether the counterpart positive message is processed or not, two different scenarios can happen during secondary rollbacks: The first scenario is shown in Figure 6 where the positive event has already been processed. The simulation object receives an anti-message, denoted as E(-21), which is the counterpart to E(21). The kernel operations are described below.

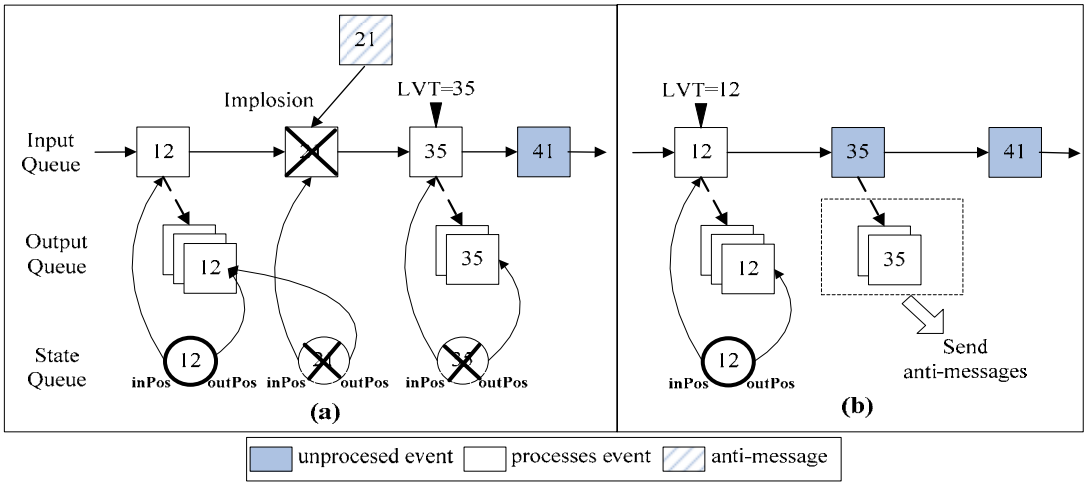


Figure 6. Kernel operations for secondary rollback (positive event already processed)

(1) Perform a message implosion to delete both E(21) and E(-21).

(2) Follow step 2 to step 7 of the primary rollback operations as presented earlier but using the timestamp of the anti-message (i.e. 21) as the rollback time.

We can see that the operations largely remain the same as those for primary rollbacks except that a message implosion replaces the previous enqueue operation.

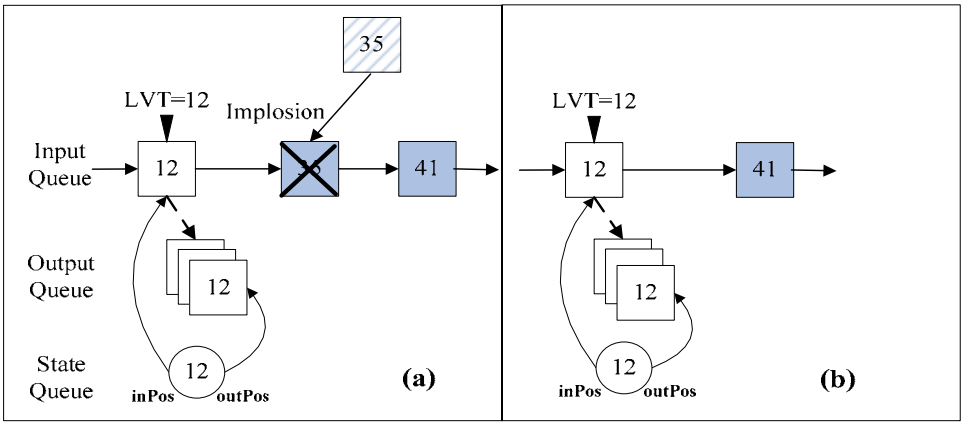


Figure 7. Kernel operations for secondary rollback (positive event not yet processed)

Another scenario is shown in Figure 7 where the positive event has not yet been processed. The only action that needs to be done is a message implosion. The simulation object continues to execute the next available event E(41) after the implosion as shown in Figure 7(b).

- **Cascaded rollback process**

From the preceding discussion, we can see that the primary rollback triggered by a straggler message is the root cause of rollbacks in Time Warp systems. Secondary rollbacks are performed *immediately* upon the arrival of anti-messages at the destinations. Hence, rollback propagation consists of one primary rollback and, optionally, multiple rounds of secondary rollbacks spreading out across the simulation system from the hosting simulation object of the primary rollback. The hosting simulation object of the primary rollback is called as *rollback originator*, and the original primary rollback of the propagation is called the *root of the propagation*. The levels of secondary rollbacks may be to any depth, and there may even be circularity in the graph of anti-message paths, but the propagation eventually terminates [Eln02]. The rollback propagation can be depicted using a tree structure, as shown in Figure 8, where rollbacks are denoted as circles and anti-messages are represented as edges.

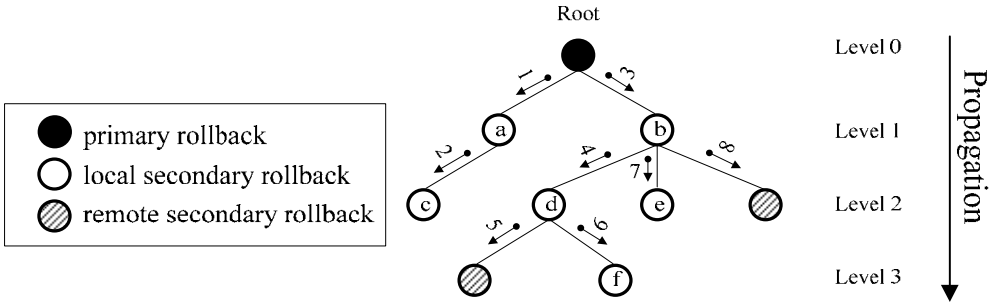


Figure 8. Tree structure of rollback propagation on a processor

Three types of rollbacks are illustrated in the diagram: the primary rollback that is the root of the propagation (black circle); secondary rollbacks happened locally on the same processor as the primary rollback (empty circle); and secondary rollbacks occurred remotely on other processors (shaded circle). Each shaded circle, in fact, represents a sub-tree of secondary rollbacks on remote processors. In the diagram, the primary rollback sends out two anti-messages that trigger two secondary rollbacks (i.e. *a* and *b*) at level 1. While *a* triggers only one further rollback referred to as *c*, *b* causes two local rollbacks (i.e. *d* and *e*) and a remote rollback. Rollback *d*, in turn, triggers one local rollback *f* as well as a remote one.

The propagation process can be described as traversing the tree from the primary rollback, following the sequence of the numbers marked in the diagram. This operation backtracks, by returning from the present rollback operation, to the most recent node it hadn't finished exploring if it hits a node that has no children (i.e. no further anti-message from that

rollback) or a node that represents a remote sub-tree (shaded circle). We observed that this traversing process exactly follows the *Depth-first search* algorithm. There is no blocking for remote rollbacks, i.e. the operation returns immediately once a shaded circle is touched. When the traversing process returns to the root of the tree, the rollbacks finish and normal execution starts on that processor.

Understanding this process can help us in implementing the kernel algorithms. For example, most of the dynamic state-saving and cancellation strategies need to measure the time spent on rollbacks for each simulation object. However, we now know that simply starting a watch at the beginning of a simulation object's rollback function and stopping the watch at the end will not do the trick. In Figure 8, the time measured in this way for the simulation object where the primary rollback takes place includes not only the time for the primary rollback itself, but also the time for all the other local secondary rollbacks in the tree.

4.2.2. GVT calculation and fossil collection

The pGVT algorithm [Dso94] is implemented by the GVT managers to realize the Time Warp global control mechanism, and users can set the frequency of GVT computation as a kernel parameter at compile time. Fossil collection is done locally within each partition. The GVT manager walks through all local simulation objects, removing all but one saved state older than GVT and all the input/output messages whose timestamps is less than the GVT (timestamp means virtual receive time for input messages and virtual send time for output messages).

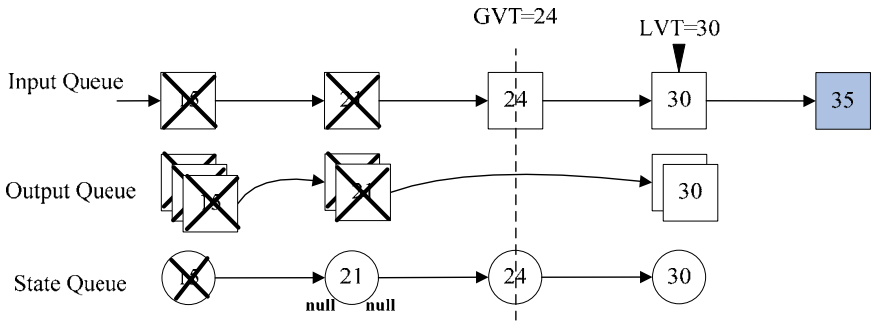


Figure 9. Status of the queues during fossil collection

Figure 9 shows the queues of a simulation object during a typical fossil collection scenario. As we will see in Chapter 7, this fossil collection scheme needs to be enhanced to work with the periodic state-saving strategy.

4.3. PROBLEMS AND FIXES

The original kernel algorithms have several flaws that cause runtime crashes of the simulation system. These problems have been identified and fixed in our research, and two of them are briefly summarized as follows:

(1) The secondary rollback mechanism was nonexistent in the kernel. Specifically, the kernel did not take the necessary actions to perform the secondary rollback after the message implosion. During secondary rollbacks, different operations should be performed based on whether the imploded positive message is processed or not. Hence, the processing status of the positive message must be recorded by the rollback facility and used later to select the appropriate operations.

(2) The insertion functions defined for the input and output queues were not correctly implemented. New events were inserted to the front of other existing simultaneous events in the queues, which can cause serious problems such as sending out wrong anti-messages during rollbacks.

We also implemented an extra step in the rollback operations to handle the data defined in the simulation objects rather than in their states. The Time Warp protocol requires a clear cut between a process and its state. That is, all modifiable data must be put into the state of the process and saved regularly in the state queue. During rollbacks, the data is recovered solely by restoring to a previously saved state. This approach has some limitations. One example is that the process may operate on dynamically allocated objects that are referenced by pointers. During the simulation, new objects are created and old ones deleted when necessary. If these pointers are saved in the state of the process, we may have trouble when the state is restored to a previously saved one but the actual objects referred by the pointers in that state have been deleted. In such case, the recovered state contains invalid pointers, resulting in runtime failure. In short, there are occasions where part of the process's internal data is inappropriate to be saved in the state queue and managed automatically by the Time Warp protocol. On the other hand, simulator developers have the knowledge as to how to handle the data in a consistent manner during rollbacks.

The following mechanism has been implemented in the kernel to solve this problem:

(1) Define an empty function, referred to as *rollbackProcessData*, in each simulation object. By default, this function does nothing.

(2) Invoke the *rollbackProcessData* function during the kernel rollback operations between step 6 and step 7 as presented in Section 4.2.1. Thus, this function acts as a placeholder in the rollback operations and allows simulator developers to define application-specific logic that needs to be performed during kernel rollbacks.

(3) Normally, simulator developers can leave this function alone and let the kernel handle the state restoration for the simulation object during rollbacks.

(4) If necessary, simulator developers can define the data that is inappropriate to be managed by the Time Warp protocol (e.g. the pointers in the previous example) directly in the simulation object rather than in its state, and provide application-specific implementation for the *rollbackProcessData* function so that the data is maintained consistently should rollbacks happen.

This solution provides programmers the required flexibility for handling some problems that may arise in the simulation. Unfortunately, it also exposes some of the rollback operations to simulator developers and relies on, at least in part, their knowledge for correctly implementing the Time Warp local control mechanism. Therefore, it should be used with care and only in situations where no other solution is available for the problem.

CHAPTER 5 DISTRIBUTED OPTIMISTIC SIMULATION IN PCD++

Research was carried out to enable the CD++ toolkit to run simulations in distributed environments using the Time Warp protocol [Gli04]. However, due to the flaws described in the previous chapter and other issues we will discuss in the following sections, the tool failed to run advanced simulations. In this work, we have redesigned most part of the PCD++ modules to achieve our goals. This chapter discusses the basic algorithms implemented in PCD++, while the enhancements and optimizations are covered in the following two chapters. The flattened structure of the simulation framework is first introduced in Section 5.1 followed by a description of the message definitions in Section 5.2. The current status of the CD++ toolkit is covered in Section 5.3. The redesigned simulation framework is presented in Section 5.4 to 5.8, while modifications to the modeling framework are provided in Section 5.9.

5.1. FLATTENED STRUCTURE FOR THE SIMULATION FRAMEWORK

As discussed in Chapter 3, the CD++ toolkit provides the modeling and the simulation frameworks to implement the behavior of DEVS and Cell-DEVS models and the simulation mechanisms respectively. In [Gli04], two new types of CD++ processors, called as *Flat Coordinator* (FC) and *Node Coordinator* (NC), are introduced to realize more efficient distributed simulations. This approach tries to reduce the communication overhead by flattening the structure of the simulation framework, while keeping the modeling framework unchanged. In this work, we adopted the flattened structure of the simulation framework. The class hierarchies in the modeling and the simulation frameworks are shown in Figure 10.

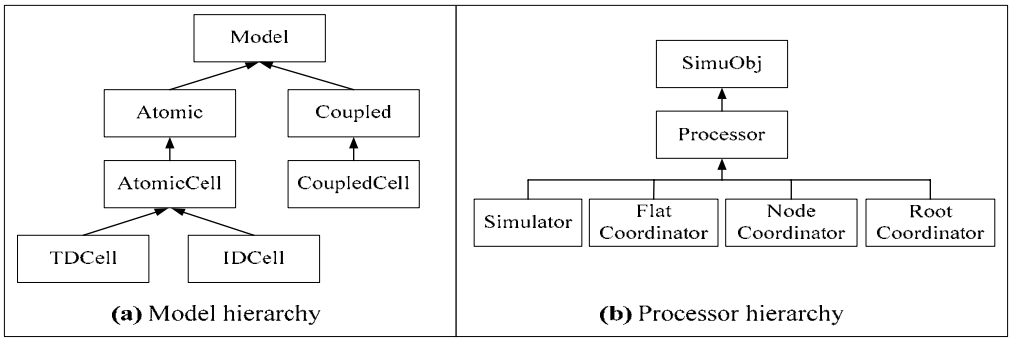


Figure 10. Model and processor hierarchies in PCD++

There are only four types of PCD++ processors existed in the simulation: *Simulator*, *FC*, *NC*, and *Root*. In the case of executing DEVS and Cell-DEVS models over multiple machines, a distributed processor structure is constructed in PCD++ to carry out the simulation. Figure 11 gives an example model and partition definition. Four atomic models (A1, A2, A3, and A4) are defined, where A1 and A2 are grouped into a coupled model C1. The TOP model represents the coupled model at the top (system) level. The example partition scheme maps the atomic models onto 2 machines: A1 and A2 on machine 0, while A3 and A4 on machine 1.

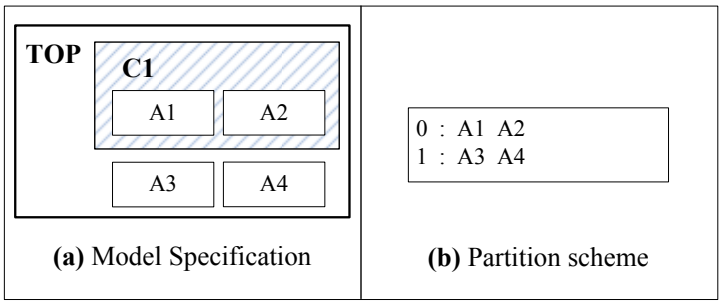


Figure 11. Example model and partition definition

The distributed processor structure corresponding to the example model and partition definition is shown in Figure 12.

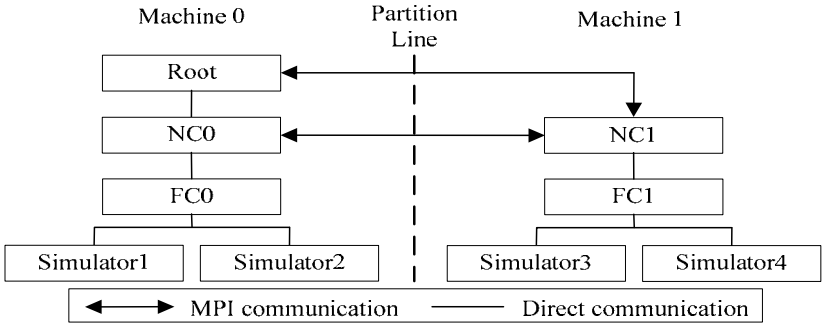


Figure 12. Distributed processor structure for the example model

Two LPs are created in this example, LP0 on machine 0 and LP1 on machine 1, each groups together the PCD++ processors on that machine. Only one Root is created on machine 0. Messages may be exchanged, locally and remotely, between the Root and the NCs. The Root starts the simulation and performs I/O operations between the simulation system and the surrounding environment. The NC created on each machine is the local central controller on its hosting LP and the end point of inter-LP communications. The FC sits between the NC and the Simulators, synchronizing the execution of its child Simulators underneath. All messages exchanged between the local Simulators are routed directly by the FC. A Simulator is

responsible for executing the DEVS abstract functions defined in its associated atomic model. If a Simulator sends a message to another Simulator running on a different machine, the message is forwarded by the FC to the local NC, which further relays the message to the remote NC that resides on the same machine as the destination Simulator. On the receiving end, the remote NC will then route the message to the target Simulator via its child FC.

5.2. MESSAGE DEFINITIONS

PCD++ processors exchange messages that can be classified into two categories: *content messages* and *control messages*. The former includes the external message (*x*) and the output message (*y*), and the latter includes the initialization message (*I*), the collect message (@), the internal message (*), and the done message (*D*). The simulation is executed in a message-driven fashion. Each type of the processors defines its own receive functions for different types of messages. The algorithms for these receive functions will be described in detail in Section 5.5.

External and output messages are used to exchange simulation data between the models. Initialization messages start the simulation. Collect and internal messages trigger the output and the state transition functions respectively in the atomic models according to the DEVS formalism. Done messages carry the model timing information for synchronizations.

PCD++ also defines six types of wrapper objects that are derived from the abstract event definition in the kernel. Each type can be used to wrap the corresponding type of PCD++ messages so that these messages can be treated as kernel events and transmitted between the processors. That is, the PCD++ messages are the actual information generated and consumed by the PCD++ processors to carry out the simulation, and the kernel events (or wrapper objects) act as the vessel by which the information is transmitted.

5.3. CURRENT STATUS OF THE CD++ TOOLKIT

CD++ simulation techniques based on optimistic synchronization protocol have been studied to extend the conservative approach used in Parallel CD++ [Tro01, Tro03]. In [Gli04], the author implemented a distributed version of CD++ using the optimistic synchronization protocol provided by the WARPED kernel. However, the original implementation of the WARPED kernel has some problems that cause runtime crashes of the simulation system, which, in turn, prevent

further examination of other issues in the CD++ toolkit. The kernel problems and their solutions have been discussed in Chapter 4. Using the optimistic synchronization protocol has changed the simulation environment so dramatically that most of the mechanisms in the toolkit need to be modified or enhanced to adapt to the new settings.

Some of the issues that need further investigation include:

(1) Mechanism for starting and terminating the simulation: In the conservative Parallel CD++, the simulation is synchronized by a central controller, the Root. Thus, it is straightforward to use the Root to perform such tasks like handling external events, advancing the simulation time, and terminating the simulation when the stop time comes. In PCD++, we have a group of NCs, each managing the sequential simulation under its control and interacting with each other in a fully asynchronous way. Hence, the simulation must be managed in a totally distributed fashion.

(2) Mechanism for saving and restoring state variables: Although the state saving and restoration mechanisms are provided by the kernel and should be transparent to the application, as described in Section 4.3, there are circumstances where simulator developers need to take care of some of the variables that cannot be handled in the standard way.

(3) Inter-LP communications: In PCD++, it is possible that different LPs have different local virtual times. The asynchronous execution of the LPs complicates the inter-LP communications since we have to handle messages with different timestamps and the potential out-of-order execution of these messages.

(4) Enhancements to the simulation framework: As we will discuss in the following sections as well as in the next chapter, algorithms for the PCD++ processors, especially for the NC, need to be enhanced significantly to address a variety of issues in distributed optimistic simulations.

(5) Modifications to the modeling framework: As we will see in Section 5.9, the modeling framework needs to be modified to adapt to the new message-passing paradigm in optimistic simulations.

Furthermore, various optimization strategies for the Time Warp mechanism as well as for the CD++ toolkit itself have not yet been exploited in the previous study. We will address several of them to reduce the overhead incurred in the simulation.

5.4. STRUCTURE FOR INTER-LP COMMUNICATIONS

In the previous versions of the CD++ toolkit, a structure called *message bag* is used by all kinds of DEVS processors to handle simultaneous events according to the P-DEVS formalism. In PCD++, there are two types of communications: *synchronous* intra-LP communications performed by all types of PCD++ processors, and *asynchronous* inter-LP communications carried out by the NCs. Due to the asynchronous nature of the inter-LP communications, the NCs need to use a special structure, hereinafter called as *NC Message Bag*, to handle messages exchanged between LPs that may have different local virtual times. The *NC Message Bag* has the following properties:

(1) A *NC Message Bag* may contain messages with different timestamps.

(2) The time of a *NC Message Bag* is defined as the minimum timestamp among the messages contained in it. An empty *NC Message Bag* has a time of infinity.

(3) Messages in the *NC Message Bag* are processed in batches in increasing timestamp order. Thus, messages with timestamp equal to the time of the *NC Message Bag* are always processed first. These messages are removed from the bag after being processed, and the time of the bag advances to the next minimum value among the timestamps of the remaining messages, if any.

Therefore, there are two types of structures used in the PCD++ simulator: the *message bag* employed by the FC and the Simulator for processing synchronized messages exchanged in intra-LP communications; and the *NC Message Bag* used only by the NC for processing asynchronous messages transmitted in inter-LP communications.

Furthermore, the PCD++ messages inserted into and removed from the bags (both types) are actually data objects dynamically allocated (generated) and deleted (consumed) by the PCD++ processors. As explained in Section 4.3, the bag structures should be defined directly in the PCD++ processors rather than in their states; and we need to provide the algorithm for the *rollbackProcessData* function to maintain the messages in the bags in a consistent manner during rollbacks, which will be covered in Section 5.8 when we discuss the mechanism for saving and restoring state variables.

5.5. MESSAGE-PROCESSING ALGORITHMS FOR PCD++ PROCESSORS

In this section, we present the simulation mechanism implemented in the PCD++ processors, including the *Simulator*, *FC*, *NC*, and *Root*. In the following discussion, a message of *type* that has a timestamp (virtual receive time) of *t* is denoted as (*type, t*). The initialization, external, output, collect, internal, and done message are symbolized as *I*, *x*, *y*, *@*,*, and *D* respectively.

5.5.1. Simulator

The algorithms for the Simulator are implemented as in [Gli04], with minor changes.

```
1. when a (I, 0) is received from the parent FC
2.         t_L = 0; ta = infinity
3.         initialize variables in the atomic model
4.         send (D, 0) to the parent FC
5. end when
```

Figure 13. Simulator algorithm for (*I, 0*)

Two variables are used in the Simulator to record its current simulation time (t_L) and the value of *sigma* (ta) as defined in the DEVS formalism. Hence, the time of the next state transition is scheduled at time (t_L + ta), so-called *absolute next time* that is denoted as t_N. Upon receiving a (*I, 0*), the Simulator resets t_L to the timestamp of the message (which is now the Simulator's current virtual time) and ta to infinity (line 2). Thus, the Simulator will remain in the passive state unless it is reactivated by a further message. The Simulator also initializes the variables defined in its associated atomic model, and then it informs its parent FC of the value of ta via a (*D, 0*) (line 4). Notice that the (*I, 0*) can only arrive at virtual time 0.

```
1. when a (@, t) is received from the parent FC
2.         if t = t_N then
3.                 t_L = t, ta = 0
4.                 y = λ(s)
5.                 send (y, t) to the parent FC
6.                 send (D, t) to the parent FC
7.         end if
8. end when
```

Figure 14. Simulator algorithm for (*@, t*)

Upon the arrival of a (*@, t*), the Simulator invokes the output function (λ) defined in the atomic model, and the resulting output is sent to the FC as a (*y, t*) (line 4 and 5). Then, it sends a (*D, t*) to the FC with ta = 0, indicating that it is imminent (line 6).

```
1. when a (*, t) is received from the parent FC
2.          if t_L ≤ t < t_N then
3.                    e = t – t_L; ta = t_N - t
4.                    s = δext(s, e, bag)
5.                    empty bag
6.          else if t = t_N and bag is empty then
7.                    s = δint(s)
8.          else if t = t_N and bag is not empty then
9.                    s = δcon(s, bag)
10.                   empty bag
11.         end if
12.         t_L = t
13.         send (D, t) to the parent FC
14. end when
```

Figure 15. Simulator algorithm for (**, t*)

Following the collect message, a (**, t*) will arrive to trigger the internal/external/confluent function defined in the atomic model depending on the timing of the message and the status of the Simulator's message bag. In the first case (line 2), the (**, t*) arrives before t_N (i.e. the simulator is not imminent yet). Thus, the Simulator must have a non-empty message bag. The external transition function (δ_{ext}) of the atomic model is called (line 4), and the messages in the message bag are removed afterwards (line 5). In the second case (line 6), the Simulator is imminent and its message bag is empty when the (**, t*) arrives. Hence, it is time to execute the internal transition function (δ_{int}) of the atomic model. In the last case (line 8), the imminent Simulator has a non-empty message bag. Therefore, a conflict between the internal and external transitions is found, and the confluent function (δ_{con}) is called accordingly (line 9). The message bag is also empties thereafter. Finally, the Simulator sends a (*D, t*) to its parent FC. While ta is updated in this algorithm in case 1 (line 3), it is modified by the user-defined logic for state transitions (line 7 and 9) in the other two cases. The resulting ta is carried in the (*D, t*) (line 13).

```
1. when a (x, t) is received from the parent FC
2.          insert message x to the bag
3. end when
```

Figure 16. Simulator algorithm for (*x, t*)

The last message that may arrive at the simulator is the (*x, t*). The received message is simply inserted into the Simulator's message bag. All external messages in the bag will be processed when the following (**, t*) arrives as shown in Figure 15. Only external messages with identical timestamp can be inserted into the message bag at any given simulation time, and a (**, t*) will always arrive in between any two consecutive batches of external messages.

5.5.2. Flat Coordinator

The FC synchronizes its child Simulators, routes messages among them, and forwards to the NC those messages sending from its children to the environment or to other remote Simulators. Simulators ready for a state transition are cached in *synchronize set*. The number of done messages that the FC should receive from its children is recorded in *doneCount* for synchronization purposes. The FC only passes control to the NC after its children (the number is given by *doneCount*) have finished their previous computation. Most of the algorithms given here are similar to those as presented in [Gli04]. However, the FC algorithm for (*y, t*) has been redesigned to address the defects in the previous version.

```
1. when a (I, 0) is received from the parent NC
2.          t_L = 0
3.          doneCount = the number of children
4.          send (I, 0) to all children
5. end when
```

Figure 17. FC algorithm for (*I, 0*)

When a (*I, 0*) arrives, the FC records the total number of its children in *doneCount* (line 3) and forwards the (*I, 0*) to each child (line 4). After this, the FC waits for a (*D, 0*) from each of its children.

```
1. when a (@, t) is received from the parent NC
2.          t_L = t; ta = 0
3.          for each imminent child Ci with t_N = t do
4.                    cache i in the synchronize set
5.                    doneCount++
6.                    send (@, t) to Ci
7.          end for each
8. end when
```

Figure 18. FC algorithm for (*@, t*)

Upon receiving a (*@, t*), the FC forwards the message to all the imminent Simulators (line 6), and records them in the *synchronize set* so that later it knows which children need to do state transitions when the (**, t*) comes.

When a (*y, t*) is received, the FC searches the model coupling information to find its ultimate destinations. A destination is ultimate if it is an input port on an atomic model or an output port on the topmost coupled model. Be careful that these two cases are not mutually exclusive. A (*y, t*) may be sent to multiple atomic models and the environment simultaneously. Furthermore, there are two scenarios in the former case: the receiving Simulators may reside

locally on the same machine as the sender or they may locate on remote machines. If the (*y, t*) is sent eventually to remote Simulators or to the environment, the FC simply forwards the (*y, t*) itself to the parent NC (line 3). Otherwise, the FC translates the (*y, t*) into a (*x, t*) using the $Z_{i,j}$ translation function (line 6) and directly sends the (*x, t*) to the local receivers (line 8). Also, the local receivers are recorded in the *synchronize set* for later state transitions.

```
1. when a (y, t) is received from a child Simulator Ci
2.         if y ultimately influences remote Simulators or the environment then
3.                 send a single (y, t) to the parent NC
4.         end if
5.         for each child Cj influenced by y do
6.                 x = Zi,j (y)
7.                 cache j in the synchronize set
8.                 send (x, t) to Cj
9.         end for each
10. end when
```

Figure 19. FC algorithm for (*y, t*)

Two major problems in the previous version are addressed here: in [Gli04], the author mistakenly assumed that a (*y, t*) sending to the environment will not influence other Simulators, which is clearly a false assumption. Moreover, in the previous algorithm, the FC translates the (*y, t*) into a (*x, t*), and forwards the resulting (*x, t*) to the NC, for remote receivers. While this is not wrong as long as the NC can handle the received (*x, t*) correctly, it has some undesirable consequences. Firstly, it blurs the different roles of the FC and NC. It is the NC that handles inter-LP communications. As the hub for intra-LP communications, the FC should not do the message translation for remote receivers, which is part of the task of inter-LP messaging. Secondly, this unnecessarily complicates the NC algorithm for (*x, t*), which, as we will see later, should be dedicated to processing inter-LP messages received from other NCs and is already complex enough due to the asynchronous nature of inter-LP communications. The algorithm presented here allows a clearer separation of roles between the NC and FC and a more reasonable division of functionalities.

As shown in the following algorithm, the received external messages are simply inserted into the FC's message bag.

```
1. when a (x, t) is received from the parent NC
2.         insert message x to the bag
3. end when
```

Figure 20. FC algorithm for (*x, t*)

As shown in Figure 21, the external messages in the FC's message bag are flushed to the local receiving Simulators upon the arrival of a (*, t) (line 6). All the receivers are recorded in the *synchronize set*. Therefore, the *synchronize set* contains the local imminent Simulators (Figure 18), if any, and/or those Simulators that have received external messages in the previous computation (Figures 19 and 21). These are the Simulators ready for a state transition, which will be triggered by the (*, t) forwarded by the FC (line 12).

```
1. when a (*, t) is received from the parent NC
2.         if doneCount = 0 then
3.                 t_L = t
4.                 for each x in bag do
5.                         for each local receiver C_i of x do
6.                                 send (x, t) to C_i
7.                                 cache i in the synchronize set
8.                         end for each
9.                 end for each
10.                empty bag
11.                for each i in the synchronize set do
12.                        send (*, t) to i
13.                        doneCount++
14.                end for each
15.                clear the synchronize set
16.        else
17.                raise error
18.        end if
19. end when
```

Figure 21. FC algorithm for (*, t)

Shown in Figure 22, for each (*D, t*) received from a child Simulator, the FC decreases the *doneCount*, and updates the child's t_N to the sum of the current simulation time and the *sigma* value carried in the received (*D, t*) (line 3). When the *doneCount* is reduced to zero, the FC calculates the closest state transition time among its children (line 6), and sends this time to the parent NC via a (*D, t*) (line 7).

```
1. when a (D, t) is received from child C_i
2.         doneCount--
3.         update C_i's t_N = t + D.ta
4.         if doneCount = 0 then
5.                 t_L = t
6.                 ta = MIN(all children's t_N) – t
7.                 send (D, t) to parent NC
8.         end if
9. end when
```

Figure 22. FC algorithm for (*D, t*)

5.5.3. Node Coordinator

As the local central controller, the NC performs a number of important operations as follows:

(1) Inter-LP communications. Messages exchanged between the NCs are handled using the *NC Message Bag*.

(2) Handling external events from the environment. The external events are scheduled by the modeler at the beginning of the simulation using a text file, so-called *EV file*. They are loaded into the NCs during the bootstrap operations. An *Event List* is used to hold the external events that the NC needs to handle during the simulation. Events in the *Event List* are processed in increasing timestamp order. The NC uses an *event-pointer* to refer to the first event in its *Event List* that has not yet been processed. Initially, this pointer points to the first event in the list.

(3) Driving the simulation on the hosting LP. The NC advances the local simulation time to the minimum among: (i) the timestamp of the external event pointed by the *event-pointer*, (ii) the time of the *NC Message Bag*, and (iii) the closest state transition time given by the FC in the received done message.

(4) Managing the flow of control messages in line with the P-DEVS formalism. The NC uses the *next-message-type* flag to keep track of the type of the control message (either @ or *) that will be sent to the FC in the next simulation cycle. The initial value of the flag is set to @.

(5) Handling a variety of problems to shield the other processors, i.e. the FC and Simulators, from the complexity of distributed optimistic simulations, as we will discuss in Chapter 6.

The NC algorithms reflect the major redesign we have done to the previous version. Some of the algorithms presented here are simplified versions, while the enhancements are delayed to Chapter 6 when we discuss the solutions for a few specific problems.

Upon receiving a (*I, 0*), the NC simply forwards it to the child FC.

```
1. when a (I, 0) is received from the Root
2.          send (I, 0) to the child FC
3. end when
```

Figure 23. NC algorithm for (*I, 0*)

The following algorithm is a simplified version for processing external messages from other remote NCs, while the enhanced version will be presented in Section 6.2. As usual, the (*x,*

t) is inserted into the *NC Message Bag*. In our design, an NC can only receive external messages from other remote NCs. These external messages carry the values sending from remote Simulators to the local ones.

```
1. when a (x, t) is received from a remote NC
2.         insert message x to the NC Message Bag
3. end when
```

Figure 24. Simplified NC algorithm for (*x, t*)

In Figure 25, if the received (*y, t*) is sent to the environment, the NC simply forwards it to the Root (line 2 to 4). Also, the NC finds out the remote machines on which the ultimate receiving Simulators locate based on the model coupling and partition information. Then, the NC translates the (*y, t*) into a (*x, t*) and sends it to the NC on each of those machines (line 6 and 7). Notice that only one (*x, t*) is sent to each of these machines, reducing the communication overhead to the minimum. On the receiving end, the (*x, t*) will be eventually delivered to the receiving Simulators on that machine.

```
1. when a (y, t) is received from the child FC
2.         if y ultimately sends to the environment then
3.                 send (y, t) to the Root
4.         end if
5.         for each remote machine i hosting destination Simulators of y do
6.                 x = Z_{i,j} (y)
7.                 send (x, t) to NC_i
8.         end for each
9. end when
```

Figure 25. NC algorithm for (*y, t*)

A (*D, t*) is the response of a control message previously sent out by the NC. It carries the synchronization information as the closest state transition time collected by the child FC. Figure 26 shows the simplified NC algorithm for (*D, t*). The first (*D, t*) received by the NC is the response to the (*I, 0*) sent to the FC to start the simulation. Since *next-message-type* is initialized to @, the NC follows the second half part of the algorithm (line 6 to 33). For a start, the NC calculates the next simulation time, *min-time*, based on the three factors as presented earlier (line 7 to 9). If the calculated *min-time* is larger than the user-specified stop time, the NC simply sets a flag (line 11) and exits the algorithm. The usage of this flag will be discussed in Section 6.2. For now, we only need to know that the NC will not send any message to the FC beyond the stop time. On the other hand, if the *min-time* is smaller or equal to the stop time, the NC performs the following operations:

(1) Send all external events scheduled at the *min-time*, if any, as external messages to the FC (line 13 to 18);

(2) Send the received external messages with timestamp equal to the *min-time*, if any, to the FC and remove them from the *NC Message Bag* (line 19 to 24);

(3) Send a control message to the FC and reset the *next-message-type* accordingly (line 25 to 31). That is, if there are imminent Simulators on the local machine, the NC sends out a (*@, t*); otherwise, it sends out a (**, t*).

```
1. when a (D, t) is received from the child FC
2.          t_L = t; t_N = t_L + D.ta
3.          if next-message-type = * then
4.                      send (*, t) to the child FC
5.                      next-message-type = @
6.          else
7.                      min-time = MIN( timestamp of the event pointed by event-pointer,
8.                                      time of the NC Message Bag,
9.                                      t_N )
10.                     if min-time > stop-time then
11.                             set a flag
12.                     else
13.                             if min-time = the timestamp of the event pointed by event-pointer then
14.                                     for each x in the Event List with min-time do
15.                                             send (x, t) to the child FC
16.                                             move event-pointer to the next event
17.                                     end for each
18.                             end if
19.                             if min-time = the time of the NC Message Bag then
20.                                     for each x in the NC Message Bag with min-time do
21.                                             send (x, t) to the child FC
22.                                     end for each
23.                             end if
24.                             remove all x in the NC Message Bag with min-time
25.                             if t_N = min-time then
26.                                     send (@, t) to the child FC
27.                                     next-message-type = *
28.                             else
29.                                     send (*, t) to the child FC
30.                                     next-message-type = @
31.                             end if
32.                     end if
33.         end if
34. end when
```

Figure 26. Simplified NC algorithm for (*D, t*)

There are two important differences between the external events (from the environment) in the *Event List* and the external messages (from other remote NCs) in the *NC Message Bag*: First, all the external events are known prior to the start of the simulation, while the external

messages are known only when they arrive at runtime. This has a significant impact on the calculation of the *min-time*. Among the three determinant factors, the timestamp of the first not-yet-processed external event and the closest state transition time, i.e. factor (i) and (iii), are assured at the time of the calculation. However, the time of the *NC Message Bag*, i.e. factor (ii), is far from certain. After the calculation of the *min-time* based on the current *NC Message Bag*, more external messages with less timestamp may arrive at the NC, invalidating the previously calculated *min-time*. The uncertainty of the external messages in the *NC Message Bag* makes the min-time calculation a *speculative* one. Secondly, the external events exist in the *Event List* throughout the simulation whereas the external messages in the *NC Message Bag* are removed after processing (line 24). In other words, the event objects in the *Event List* are static, while those in the *NC Message Bag* are dynamic. As a result, they are treated differently during state saving operations, as we will discuss in Section 5.7.

Let's go back to the NC algorithm for (*D, t*). The *next-message-type* is set to * only after the NC sends out a (*@, t*) (line 27), in which case there must be imminent Simulators on the LP and their output functions will be invoked upon receiving the (*@, t*). These imminent Simulators need to do internal transitions immediately after the output operations. Therefore, the NC triggers the internal transitions by sending out a (**, t*) (line 4). On the other hand, if there is no imminent Simulator at this time, the NC always sends a (**, t*) whenever external messages are flushed out (line 29). Thus, external transitions will be performed in the non-imminent Simulators, consuming the external messages.

5.5.4. Root Coordinator

The role of the Root Coordinator is weakened significantly in our design. It only handles environment-oriented output messages during the simulation.

```
1. when a (y, t) is received from a NC
2.         if createOutput = true then
3.                 for each output port on the TOP model influenced by y do
4.                         create a FileData object fd for y
5.                         insert fd into the file queue corresponding to the OUT file
6.                 end for each
7.         end if
8. end when
```

Figure 27. Root algorithm for (*y, t*)

As shown in Figure 27, Output to the environment is done through a text file called as output file or OUT file. Internally, the Root uses a flag, *create-output*, to record whether such a file has been created or not. If the OUT file is ready (line 2), the Root finds out all output ports on the TOP model to which the (*y, t*) will be ultimately sent. Then, it creates a *FileData* object from the (*y, t*) for each of these ports and inserts it into the file queue corresponding to the OUT file (line 4 and 5). Eventually, the data in the file queue will be written to the physical file by the kernel when GVT advances.

5.6. A MESSAGE-PASSING SCENARIO

Based on the message-processing algorithms described in the previous section, we now illustrate the message flow in PCD++ using an *event precedence graph*, where a vertex (black dot) represents a message, and an edge (black arrow) represents the action of sending a message with the message type placed nearby. A line with a solid arrowhead denotes a (synchronous) intra-LP message and a line with a stick arrowhead denotes an (asynchronous) inter-LP message. A lifeline (dashed line) is drawn for each PCD++ processor. The sequence of message execution is marked by the numbers following the message type.

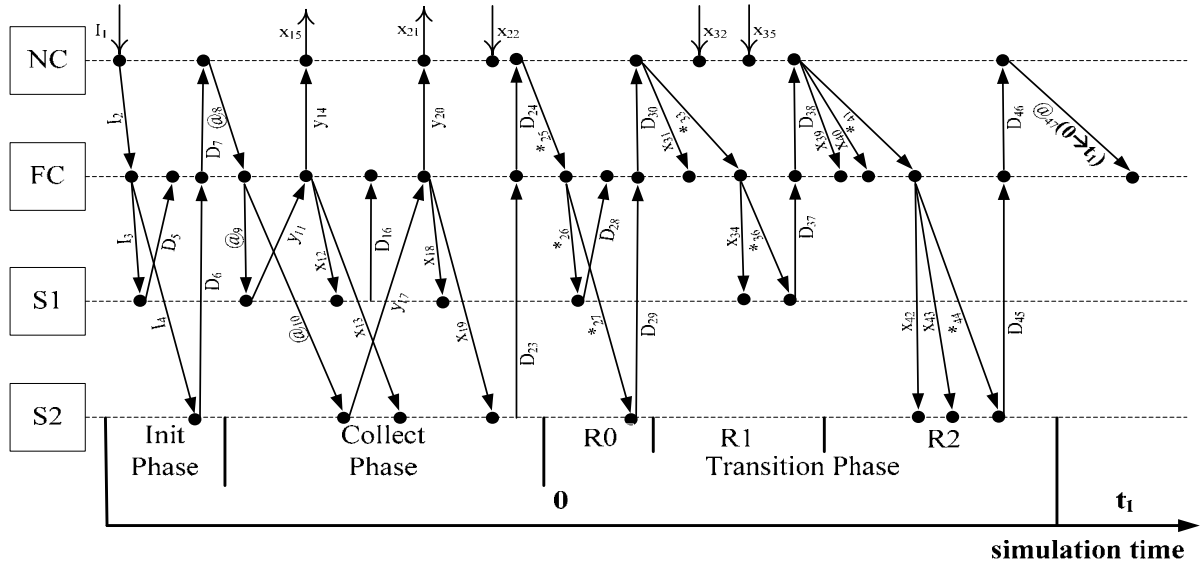


Figure 28. Example message-passing scenario

Without loss of generality, we focus our analysis on a single LP. Since all the LPs operate in a similar way, a general idea of the message-passing paradigm can be obtained. Figure 28 shows a LP with four PCD++ processors, a NC, a FC, and two Simulators (S1 and S2). We do

not include the Root in order to keep the diagram as concise as possible. Also, external events from the environment are not considered in the example. A further simplification is the absence of the potential out-of-order execution. Since the rollback operations are performed automatically and transparently in the kernel, we can largely ignore them when we consider the message flow at the PCD++ layer. Although there may be messaging anomalies that cannot be handled by the kernel rollback facility alone, which will be covered in Section 6.5, it is sufficient to leave them alone for now.

The simulation on this LP starts at simulation time 0, upon the arrival of an initialization message (I_1) at the NC. The NC forwards the message to the FC (I_2), which further forwards it to the Simulators (I_3, I_4). The Simulators respond with done messages (D_5, D_6) after initializing their associated atomic models. The FC informs the NC about the closest state transition time after receiving all the done messages from its children (D_7).

At this time, all Simulators are imminent. Thus, the NC sends a collect message ($@_8$) to the FC, which again forwards the message to each of the Simulators ($@_9$, $@_{10}$). Upon receiving the collect message, imminent Simulators execute their output functions and send output messages to the FC. S1 processes $@_9$ first and sends an output message (y_{11}) to the FC. Suppose that this output message is sent to all local Simulators as well as to remote ones. The FC first translates the output message into an external message and sends the message to each of its children (x_{12}, x_{13}), and then forwards the output message itself to the NC (y_{14}), which, in turn, translates the output message into an external one (x_{15}) and sends it remotely to all destination NCs. In our example, the external message is sent to only one remote NC. After processing y_{11}, the FC turns to process the done message (D_{16}) sent from S1 right after y_{11}. This done message represents the end of the output operation at S1. Then, the FC continues to process the output message (y_{17}) from S2 just like in the previous case (x_{18}, x_{19}, y_{20}, x_{21}). Notice that before the execution of D_{23} at the FC, a remote external message sending to S1 arrives at the NC (x_{22}). Hence, this message is inserted into the *NC Message Bag*. Now, it is the FC's turn to process the postponed D_{23} and a new done message (D_{24}) is sent to the NC.

As a response, the NC sends an internal message ($*_{25}$) to the FC immediately. This message is then delivered to all Simulators that have just finished their output operations ($*_{26}$,$*_{27}$). Internal transitions are triggered at these Simulators followed by done messages emitted to the FC (D_{28}, D_{29}). The FC sends the closest state transition time to the NC via a done message

(D_{30}). In processing D_{30}, the NC calculates the *min-time* and notices that a message, derived from x_{22}, with that *min-time* exists in its *NC Message Bag*. Therefore, it sends this external message to the FC followed by another internal message (x_{31}, $*_{33}$). The external message (x_{31}) is added to the FC's bag and will be sent to the receiving Simulators when $*_{33}$ is executed by the FC. Meanwhile, another remote external message (x_{32}) sending to S2 arrives. The execution of $*_{33}$ is thus delayed until after x_{32} is processed by the NC. Then, the FC executes $*_{33}$, flushing x_{34} to S1 followed by $*_{36}$. The external message x_{34} is added into S1's bag, thereby accepting the value previously transmitted by x_{22} from a remote sender. In the mean time, one more remote external message (x_{35}) sending to S2 arrives and gets executed by the NC. After that, the internal message $*_{36}$ invokes S1's external transition, consuming the value wrapped in x_{34}. The resulting done message (D_{37}) is sent to the FC. When D_{38} is executed by the NC, it sends all external messages with *min-time* existed in its *NC Message Bag* to the FC (x_{39}, x_{40}), again followed by an internal message ($*_{41}$). These messages are executed by the FC and then forwarded to the destination Simulator S2 (x_{42}, x_{43}, $*_{44}$). Now, the values derived from x_{32} and x_{35} are consumed in S2's external transition function, and a done message (D_{45}) is sent to the FC.

When D_{46} is processed at the NC, there is no message in its *NC Message Bag*, and the closest state transitions are scheduled at time t_1. Hence, the NC advances the local simulation time from 0 to t_1 and sends to the FC a collect message ($@_{47}$) that has a send time of 0 and a receive time of t_1, thereby starting a new cycle of simulation similar to that initiated by $@_8$.

Some characteristics of the message flow are summarized as follows:

(1) The execution of messages at any given simulation time on a LP can be classified into at most three distinct phases, namely *initialization phase*, *collect phase*, and *transition phase*. Only one initialization phase exists at the beginning of the simulation (time 0), consisting of messages in the range of [I_1, D_7]. The collect phase at a specific simulation time starts with a collect message sending from the NC to the FC and ends with the following done message received by the NC. In the diagram, the collect phase at time 0 comprises messages in range [$@_8$, D_{24}]. This phase is optional, it happens if, and only if, there are imminent Simulators on the LP at that time. Finally, the transition phase at a specific simulation time begins with the first internal message sending from the NC to the FC and ends at the last done message received by the NC at that time. In our example, messages in the

range of [$*_{25}$, D_{46}] belong to the transition phase at time 0. The transition phase is mandatory for each individual simulation time.

(2) The variables defined in the atomic models are initialized in the initialization phase. The output functions in the imminent atomic models are invoked during the collect phases. The state transitions are performed for the atomic models in the transition phases. These phases are arranged in line with the P-DEVS formalism.

(3) Outgoing inter-LP communication happens only in the collect phases, whereas incoming inter-LP communication can occur in any phase. Since the output functions of imminent models are invoked only in the collect phases, it is clear that at any given simulation time, all external messages going to remote NCs are sent out by the end of the collect phase of that time. On the other hand, an external message from a remote source can arrive at the destination NC when the simulation is executed in any phase.

(4) Although these phases also exist in other versions of the CD++ toolkit, the optimistic version differs from the standalone and conservative ones in the structure of the transition phase. In the previous versions, the message exchanges are synchronized and the state transition is performed only once for the Simulator that needs to change its state at a specific time. In PCD++, state transitions in the Simulators may be performed repeatedly at any given simulation time as additional remote external messages arrive at the NC, resulting in a multi-round transition phase. A transition phase consisting of (n+1) rounds is denoted as **[R_0…R_n]**. Each round starts with zero/one/more external messages followed by an internal message sending from the NC to the FC and ends with a done message returned back to the NC. In Figure 28, the transition phase at time 0 has three rounds: R_0 includes messages in range [$*_{25}$, D_{30}], R_1 involves messages in [x_{31}, D_{38}], and R_2 contains messages in [x_{39}, D_{46}]. During each round, state transitions are performed *incrementally* with additional external messages and/or for potentially extra Simulators.

These characteristics of the message flow, especially the multi-round transition phases, have a significant impact on the computation of the models. Accordingly, the new algorithms for Cell-DEVS models with transport and inertial delays are presented in Section 5.9. Based on the

above discussions, Section 6.1 gives a new abstraction for optimistic simulations in PCD++.

5.7. STARTING AND TERMINATING SIMULATIONS

The mechanisms for starting and terminating the simulation in PCD++ are dramatically different from those in the previous versions because of the optimistic and decentralized approach to distributed simulation. The major modifications to the bootstrap algorithm are summarized as follows:

The first modification concerns with handling external events from the environment. In PCD++, the Root is not longer the central controller and the simulation is carried out under the control of a group of NCs. Accordingly, we have to distribute the task of managing external events among these NCs. Each NC uses an *Event List* to hold the external events it needs to handle during the simulation. The events given in the EV file are *purged* before they are loaded into a NC's *Event List*. That is, an event is loaded into a NC's *Event List* if and only if that event will ultimately influence some of the Simulators controlled by that NC. As a result, a NC can solely depend on its own *Event List* to process the external events when the local virtual time advances to the event's timestamp. Furthermore, this arrangement is crucial for the NC to calculate the *min-time* as discussed at the end of Section 5.5.3.

The second modification is that the Root now sends initialization messages to all the NCs to start the simulation in a distributed way. Previously, the Root sends only one initialization message to the coordinator associated with the TOP model. As the intermediary coordinators are removed in our flattened structure, using this approach in PCD++ causes runtime failure.

The last modification is to handle the user-specified stop time. This issue is naturally related to the simulation termination mechanism. In the standalone and conservative versions, the stop time is loaded into the Root and the termination of the simulation is totally controlled by the Root. As we know, with Time Warp the detection of termination is one of the several global issues handled in terms of GVT [Jef85]. In PCD++, the stop time is loaded into each NC so that the NC will not send out any message with timestamp beyond the stop time (line 10 and 11 in Figure 26). At the same time, the stop time is also passed into the LP as the parameter of the *simulate* function, which defines the main event processing loop. At the end of each simulation cycle, the LP compares the stop time with the current GVT. The simulation terminates when the

GVT exceeds the user-defined stop time.

Since the GVT computation is CPU and communication intensive, too frequent GVT calculation can degrade the performance. Usually, we set the number of simulation cycles between two consecutive GVT calculations to tens of thousand. Therefore, the LP may keep executing events with timestamp larger than the stop time while the GVT is lagging behind the actual simulation time. In the worst case when the last GVT calculation was done just before the stop time, the LP may keep executing tens of thousand of extra events with timestamp actually larger than the stop time until the next GVT computation. This is why we need to load the stop time into the NC so that no message with timestamp larger than the stop time will be emitted. In this case, the LP will run empty loops when the simulation time exceeds the stop time, while waiting the GVT value to eventually catch up as illustrated in Figure 29.

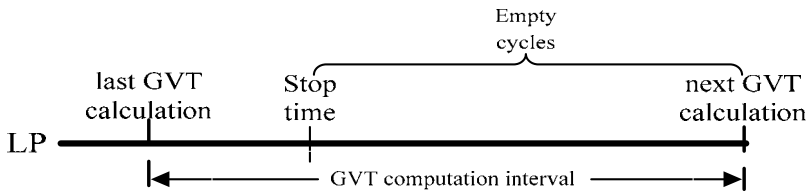


Figure 29. Terminating the simulation on a LP

This approach has two advantages: (1) the empty cycles after the stop time are consumed faster than normal event execution, allowing more efficient termination and better performance; (2) we can have a clear cut of the simulation results at the stop time.

5.8. SAVING AND RESTORING STATE VARIABLES

In Time Warp parallel simulation, each process must periodically save its local state such that, in the event of a causality error, a rollback to a correct state is possible [Fre02]. The Time Warp protocol requires that all the modifiable variables of a process are saved in its state and restored during rollbacks by the Time Warp executive. However, this approach has some disadvantages when applied to objects. We have to perform a *deep copy* for the objects in order to fully recover them later from a previously saved state, resulting in large states and long operation time for state saving and restoration.

To reduce the overhead, we want to save only the pointers to these objects in the state of a process, and perform a *shallow copy* during state saving. That is, only the values of the pointers are copied rather than the objects themselves.

In other situations, the simulator developer has the knowledge as to how to maintain the data objects in a consistent way during rollbacks. Hence, this kind of data objects can be removed entirely from the state of the process. Even the pointers to these objects need not to be saved in the state, further reducing the state copying overhead. Instead, these objects are defined in the process itself and will not be saved during the state-saving operation. In Section 4.3, we defined an empty function (*rollbackProcessData*) in each simulation object that will be invoked during kernel rollback operations. The simulator developer can provide the implementation of this function to take the responsibility of recovering the data objects during rollbacks.

Therefore, we have three different methods for saving and restoring the modifiable variables defined in PCD++ processors and models: *deep copy*, *shallow copy*, and *no copy at all*. The following rule of thumb gives the criteria for selecting the appropriate method for different types of variables.

(1) All primitive data types, or a collection of them, can be saved using the deep copy method. Examples of such variables include the *doneCount* and *synchronize set* in the FC, and the *next-message-type* in the NC. Notice that the *synchronize set* is a container that holds primitive integers (processor ids). As the processors have invariable ids, the data contained in the *synchronize set* will never be invalidated.

(2) Objects that exist throughout the simulation can be saved using the shallow copy method. The NC's *Event List* is a perfect candidate for this method. Once loaded into the *Event List* during the bootstrap operation, the event objects remain in the list until the end of the simulation. We only need to use a pointer, *event-pointer,* to maintain our current position in the list. The *event-pointer* is saved in the NC's state, whereas the event objects are not.

(3) Objects that are dynamically allocated and deleted during the simulation can be saved using the deep copy method or, if the simulator developer has enough knowledge about them, can be removed from the state of the process all together. The message bags (including the *NC Message Bag*) are defined directly in the PCD++ processors, and the message objects in the bags are handled in virtue of the

rollbackProcessData function during rollbacks. Figure 30 gives the algorithm for the *rollbackProcessData* function, which is common for all kinds of PCD++ processors.

```
1. when function rollbackProcessData(rollback-time) is invoked
2.         for each message m in the current message bag do
3.                 if m.timestamp >= rollback-time then
4.                         remove m from the message bag
5.                         delete m
6.                 end if
7.         end for each
8. end when
```

Figure 30. Algorithm for function *rollbackProcessData*

In PCD++, both processors and models define their modifiable variables. A connection is made between a processor's state and the corresponding model's state so that both are saved in the state queue of the processor. Modelers can define variables as they want and put them in the states of the models. These user-defined model variables will automatically participate in the Time Warp synchronization protocol.

5.9. ASYNCHRONOUS STATE TRANSITIONS IN CELL-DEVS MODELS

As discussed in Section 5.6, the PCD++ message-passing paradigm is considerably different from that in the standalone and conservative versions. At any given simulation time, the state transitions may be performed incrementally in the Simulators with additional external messages, resulting in a multi-round transition phase. Therefore, the functions for Cell-DEVS atomic models must be adapted to this *asynchronous state transition paradigm* to obtain the same simulation results in PCD++ as in the previous versions for any given model.

A brief description of the new computation model under the asynchronous state transition paradigm is given as follows:

(1) Applying preemptive semantics to the state transition logic. For a transition phase $[R_0 \ldots R_n]$, the state transitions in all but the last round (R_n) are based on incomplete information and, hence, false transitions. Only R_n has the best chance to perform the correct transition (This is the case if no rollback happens later on. Otherwise, the whole transition phase will be reprocessed after the rollbacks.) Since the state transition in a later round involves additional external messages, it has a better

chance to perform the correct computation and, thus, generate the correct results. Therefore, the state transition logic should be implemented so that the computation of the later round preempts that of the previous round. In the end, the potentially correct results obtained in R_n preempt those erroneously generated in R_{n-1}, and the simulation advances to the next virtual time afterwards. Both the value and the state of the cell must follow this preemptive logic during the multi-round state transitions. To do so, the cell needs to record its *previous value* and *previous state* passed in from the previous virtual time at the beginning of R_0 in the transition phase at each individual simulation time. For time 0, the previous value and state are the cell's initial value and state defined by the modeler. Except the R_0 at time 0, the entry point of the first round is identified by a change in the simulation time. Hence, a cell can safely record its previous value and state once a time change is detected at the beginning of the state transition algorithm. For time 0, this job can be done in the initialization phase.

(2) Handling user-defined state variables. User-defined state variables may be involved in the evaluation of local rules. With the multi-round transition phase, this computation becomes much more complex. During each round, a potentially different rule is evaluated and the state variables referenced in the rule are computed. As a result, potentially wrong values are assigned to the variables and passed to the next round. The computation errors accumulate throughout the rounds, and finally, the wrong values are passed to the simulation at the next virtual time. To ensure correct computation of the state variables, a cell needs to record the values of the state variables at the beginning of R_0. These recorded values are inherited from the computation in the last round at the previous simulation time and, thus, they are potentially correct. In each of the following rounds, the variables are first restored to the recorded values. Only after this restoration operation, can a new computation be performed. Therefore, the cell always uses the potentially correct values as the basis for a new computation.

(3) Handling external events. In CD++, port-in transition function (for evaluating external events) is given a higher priority than the local transition rules. Under the new asynchronous state transition paradigm, the computation results of the port-in

transition function can be modified by the local transition rules in later rounds. In order to preserve the effect of external events throughout the multi-round transition phase, we define a flag, called as *event-flag*, in each cell. Whenever the cell's value is influenced by an external event or events at a given simulation time, this flag is set so that no further changes can be done to the value during the following rounds at this time. This flag will be reset once the preserved value has been output to other cells and the R_0 at a new simulation time begins. In this case, the influence of the external event has spread out in the cell space as expected, and the cell's value is again under the control of its local transition rules.

Based on the above analysis, we now present the enhanced algorithms for Cell-DEVS models. The algorithms presented in the following subsections only include the core simulation logic. Other auxiliary logic untouched in our revision such as that for model quantization is excluded for clarity. Another simplification is that we only show a single output port, hence a single cell value, in the algorithms. Actually, a cell can define multiple inter-cell ports to communicate with its neighbors. It is easy to obtain the complete version by replacing the single cell value with a list of values, each corresponding to an extra output port.

5.9.1. Cell-DEVS models with transport delay

Shown in Chapter 2, four abstract functions are defined in the P-DEVS formalism, namely the *output function* (λ), *internal transition function* (δ_{int}), *external transition function* (δ_{ext}), and *confluent transition function* (δ_{con}). An *initialization function* is also defined to initialize the variables used by each atomic model. In CD++, a default algorithm for the δ_{con} function has been provided that gives a higher priority to internal events. This default δ_{con} function is inherited in PCD++ and hence not included in the following discussions. Users can always define their own δ_{con} function by deriving from the modeling framework.

```
1. when the initialization function is invoked
2.          get the cell's initial value v
3.          event-flag = false
4.          transient-value = v
5.          time-record = 0
6.          state-variable-record = state-variable
7.          push <0 / out = v> into the queue
8.          holdIn(active, 0)
9. end when
```

Figure 31. Initialization algorithm in Cell-DEVS models with transport delay

Figure 31 gives the algorithm for the initialization function, which is invoked by the associated Simulator during the initialization phase. First, the cell retrieves its initial value *v* given by the modeler (line 2). The *event-flag* for identifying external events is initialized to false (line 3). The *transient-value* for recording the tentative value changes during the multi-round state transitions at a given simulation time is initialized to the cell's initial value (line 4). The *time-record* used to detect the entry point of R_0 is initialized to time 0 (line 5). The cell also records the initial value of the user-defined state variable in *state-variable-record* (line 6). Again, we only show a single user-defined state variable in the algorithm for simplicity. Actually, a structure is used to hold all the state variables and another similar structure to keep their records. Then, the cell creates an element *<0 / out = v>* and inserts it into the queue so that its initial value *v* can be sent to all its neighbors via its output port, *out*, during the collect phase at time 0 (line 7). Finally, function *holdIn* is invoked, activating the cell at time 0 (line 8). As a result, the cell is ready to output its initial value once the collect phase begins.

The λ and δ_{int} functions are implemented as in the previous conservative version [Tro03].

```
1. when the λ function is invoked
2.          for each element <t / port = value> in the queue do
3.                    if t = current-time then
4.                              send value to the port
5.                    end if
6.          end for each
7. end when
```

Figure 32. Algorithm for the λ function in Cell-DEVS models with transport delay

Figure 32 shows the algorithm for the λ function, which is invoked during each collect phase. The cell simply walks through the queue, and sends the value to the specified port if the time of the element is equal to the current simulation time (*current-time*) as indicated by the timestamp of the collect message that has triggered this function.

```
1. when the δint function is invoked
2.          for each element <t / port = value> in the queue do
3.                    if t = current-time then
4.                              remove the element from the queue
5.                    end if
6.          end for each
7.          if queue is empty then
8.                    passivate()
9.          else
10.                   holdIn(active, time to the next output)
11.         end if
12. end when
```

Figure 33. Algorithm for the δ_{int} function in Cell-DEVS models with transport delay

As shown in Figure 33, when the δ_{int} function is called, the cell first removes from the queue all the elements that have been sent out in the preceding λ function (line 2 to 6). Then, it resets its state based on the current status of the queue. If no further output is scheduled, its state is set to passive (line 8). Otherwise, the cell calculates the remaining time to the next scheduled output time, and remains active until that time (line 10).

The new state transition logic is realized in the δ_{ext} function, which is invoked repeatedly throughout the multi-round transition phases. The algorithm is shown in Figure 34.

```
1. when the δext function is invoked
2.          set the values of neighboring cells based on the current message bag
3.          time-change = false
4.          if time-record != current-time then
5.                      time-record = current-time
6.                      time-change = true
7.                      get the previous-value from the input port
8.                      transient-value = previous-value
9.                      event-flag = false
10.                     state-variable-record = state-variable
11.         end if
12          state-variable = state-variable-record
13.         if there are external events in the current message bag then
14.                     new-value = port-in-function()
15.         else
16.                     new-value = local-transition-function()
17.         end if
18.         output-time = current-time + delay
19.         if event-flag = false then
20.                     if new-value is derived from external events then
21.                                 event-flag = true
22.                     end if
23.                     if (new-value != previous-value) & (transient-value = previous-value) then
24.                                 transient-value = new-value
25.                                 push <output-time / out = new-value> into the queue
26.                                 holdIn(active, time to the next output)
27.                     else if (new-value = previous-value) & (transient-value != previous-value) then
28.                                 transient-value = new-value
29.                                 if time-change = false then
30.                                             remove the previous element from the queue
31.                                             if queue is empty then
32.                                                         passivate()
33.                                             else
34.                                                         holdIn(active, time to the next output)
35.                                             end if
36.                                 end if
37.                     else if (new-value != previous-value) & (transient-value != previous-value) & (new-value != transient-value) then
38.                                 transient-value = new-value
39.                                 if time-change = false then
40.                                             replace  the previous element in the queue
41.                                             holdIn(active, time to the next output)
42.                                 end if
43.                     end if
44.         end if
45. end when
```

Figure 34. Algorithm for the δ_{ext} function in Cell-DEVS models with transport delay

First of all, the cell sets the values of its neighbors based on the existing external messages in the message bag (line 2), thereby consuming these external messages. Then, it compares the *time-record* with the *current-time* to detect a possible change of time (line 4), which indicates the entry point of R_0 at the current simulation time. Once found, a series of operations are performed (line 5 to 10): the *time-record* is updated to the current virtual time so that no more time change will be detected in the following rounds, and a flag called *time-change* is set accordingly; the cell's value passed in from the previous time, *previous-value*, is retrieved, and the *transient-value* is initialized to this *previous-value* for use in the later rounds; the *event-flag* is reset to false in case external events have been processed during the computation of the previous time; and the current value of the user-defined state variable inherited from R_n of the previous time is recorded in the *state-variable-record*. These housekeeping operations are done only at the beginning of R_0 for each individual simulation time.

The remaining logic (line 12 to 44) is common for all the rounds in a transition phase. The state variable is restored to the recorded one before any rule evaluation (line 12). From line 13 to 17, we can see that the port-in function is preferred over the local transition function during the rule evaluation. The *event-flag* is set (line 20 to 22) if the new value is derived from external events (line 14). Once this flag is set, no further modification to the cell's value is allowed in the following rounds until the simulation is advanced to the next virtual time (line 19).

The preemptive semantics of the transition logic is realized in line 23 to 43. There are three possible cases that can happen in each round: a new value change occurs (line 23 to 26), the value is changed back to the *previous-value* (line 27 to 36), or the value is changed further from the result of the previous round (line 37 to 43). For all these cases, the *transient-value* always follows the newly generated *new-value* (line 24, 28, and 38). Thereby, it records the tentative value change in the present round and will be used in the conditional expression of the immediately subsequent round to choose different logic for different cases. Once a new value change is detected, the *new-value* is inserted into the queue and an output is scheduled (line 25 to 26). If the cell's value is changed back to the *previous-value* (line 27), i.e. there is actually no value change if we consider the computation up to the current round as a whole, the cell preempts the result of the previous round by removing the previously inserted element from the queue (line 30), and reschedules output based on the current queue (line 31 to 35). On the other hand, if the cell's value is changed further (line 37), the cell preempts the previous result by

replacing the element with a new one, and reschedules output accordingly (line 40 to 41). That is, the cell implements preemptive logic in the multi-round transition phase when no time change is found (line 29 and 39). The preemptive semantics is indirectly applied to the state of the cell since the cell's state, decided by the *holdIn* function, is always updated according to the current queue whose elements are maintained by the preemptive logic.

5.9.2. Cell-DEVS models with inertial delay

We now present the new algorithms for Cell-DEVS atomic models with inertial delay. The same simplifications have been done in the following algorithms as in the previous subsection. Unlike in the previous case, Cell-DEVS atomic models with inertial delay need to explicitly apply the preemptive semantics to their states in the transition logic.

The algorithm for the initialization function is given in Figure 35. The cell's future value *f* is initialized to the initial value (line 4), which is also copied in *f-record* (line 5). Notice that the cell needs to explicitly make a copy of its state, *state-record*, and the duration of the state, *delay-record* (line 6 to 7). The other operations are the same as in the initialization function for cells with transport delay.

```
1. when the initialization function is invoked
2.          get the cell's initial value v
3.          event-flag = false
4.          f = v
5.          f-record = f
6.          state-record = passive
7.          delay-record = infinity
8.          time-record = 0
9.          state-variable-record = state-variable
10.         holdIn(active, 0)
11. end when
```

Figure 35. Initialization algorithm in Cell-DEVS models with inertial delay

The λ and δ_{int} functions are also implemented as in the previous conservative version [Tro03]. Shown in Figure 36 and 37, an imminent cell simply sends its current future value to the output port in the λ function, and then it changes to the passive state in the following δ_{int} function.

```
1. when the λ function is invoked
2.          send f to the output port
3. end when
```

Figure 36. Algorithm for the λ function in Cell-DEVS models with inertial delay

```
1. when the δint function is invoked
2.          passivate()
3. end when
```

Figure 37. Algorithm for the δ_{int} function in Cell-DEVS models with inertial delay

The new state transition logic is implemented in the δ_{ext} function, as shown in Figure 38.

```
1. when the δext function is invoked
2.          set the values of neighboring cells based on the current message bag
3.          time-change = false
4.          if time-record != current-time then
5.                      time-record = current-time
6.                      time-change = true
7.                      event-flag = false
8.                      f-record = f
9.                      state-variable-record = state-variable
10.                     state-record = current-state
11.                     if current-state = active then
12.                                 delay-record = tN – current-time
13.                     else
14.                                 delay-record = infinity
15.                     end if
16.         end if
17.         state-variable = state-variable-record
18.         if there are external events in the current message bag then
19.                     new-value = port-in-function()
20.         else
21.                     new-value = local-transition-function()
22.         end if
23.         if event-flag = false then
24.                     if new-value is derived from external events then
25.                                 event-flag = true
26.                     end if
27.                     if new-value != f then
28.                                 if current-state = passive then
29.                                             holdIn(active, delay)
30.                                 else
31.                                             if new-value != f-record then
32.                                                         if tN – current-time > 0 then
33.                                                                     holdIn(active, delay)
34.                                                         end if
35.                                             else if time-change = false then
36.                                                         if state-record = passive then
37.                                                                     passivate()
38.                                                         else
39.                                                                     holdIn(active, delay-record)
40.                                                         end if
41.                                             end if
42.                                 end if
43.                                 f = new-value
44.                     end if
45.         end if
46. end when
```

Figure 38. Algorithm for the δ_{ext} function in Cell-DEVS models with inertial delay

The cell detects time changes and does the housekeeping operations at the beginning of R_0 for each simulation time just like an atomic model with transport delay does. When the δ_{ext} function is invoked, the current state of the cell, *current-state*, may be either passive or active. If it is passive, the cell's current future value *f* has been committed and the duration of the state is infinity. Otherwise, the current future value is still a tentative one and the cell will remain in active state until the time of the next scheduled state transition (t_N) comes. Hence, the duration of the active state is given by (t_N - current-time). The cell records the current *f* at the beginning of R_0 for reference in the following rounds of state transitions at this time (line 8). Also, it copies the current state (line 10) and the duration of that state (line 11 to 15) in *state-record* and *delay-record* respectively. The user-defined sate variable and external events are handled in the same way as in transport-delay cells.

The preemption is done once a change of the future value is detected (line 27), and the operations are carried out in two steps: one is to preempt the cell's current state along with its duration (line 28 to 42), and the other is to preempt the cell's current future value (line 43). While the second step can be done with ease by simply assigning the *new-value* to the future value, preemption of the state needs to be handled more carefully. Here, we have two different cases: if the current state is passive, it can be directly preempted with the *holdIn* function (line 29). Notice that this operation not only preempts the state itself (from passive to active), but also preempts the duration of the state (from infinity to a certain value of *delay*). Also, this operation is common for both preemption of events happened at different time (i.e. later events preempt earlier ones, referred to as *situation-type-A*) and preemption of events occurred at the same time but in different rounds of a transition phase (i.e. events in a later round preempt those in previous rounds, referred to as *situation-type-B*).

If the current state is active, it should be preempted differently depending on whether the preemption is done in *situation-type-A* or in *situation-type-B*. The operations for *situation-type-A* are already defined by the semantics of the inertial delay [Wai02b]. The state itself remains active and the duration of the state is changed from the current value of (t_N - *current-time*) to the new *delay* (line 32 to 34). Notice that a conditional expression (line 31) is added to the operation. However, this condition only takes effect in *situation-type-B*. The reason for this is that whenever the simulation time changes, the *f-record* is updated to have the same value of *f* (line 8). Hence, the condition expressed in line 31 is always true whenever the condition in line 27 is

satisfied in *situation-type-A*. On the other hand, the cell's *f* may or may not be changed to the *f-record* during the following rounds in *situation-type-B*. If it is not changed to *f-record* (line 31), the preemption logic is the same as in *situation-type-A* (line 32 to 34). Otherwise, we can conclude that there is actually no value change when the multiple rounds at this time are considered as a whole. Hence, the current state is recovered to the *state-record* (line 36 to 40). Notice that the duration of the state is recovered to the *delay-record* as well (line 39). This recovery can only occur in the multiple rounds of a transition phase, as secured by the condition in line 35.

CHAPTER 6 ENHANCEMENTS TO PCD++ AND THE WARPED KERNEL

The new algorithms for the simulation and modeling frameworks of the PCD++ toolkit have been presented in Chapter 5. However, before the toolkit can be used to execute DEVS and Cell-DEVS models optimistically in distributed environments, it must be enhanced to address a variety of issues. This chapter is concerned with the essential enhancements to the PCD++ and the WARPED kernel to ensure correct and efficient execution of simulations. The notion of *wall clock time slice* (WCTS) is presented in Section 6.1 as an abstraction for the simulation process on each LP. A new state of the NC, called as *dormant*, is introduced in Section 6.2 to handle asynchronous execution of the LPs. Section 6.3 is devoted to dealing with rollbacks happened at virtual time 0, a problem left unsolved in the WARPED kernel. A two-level *user-controlled state saving* (UCSS) strategy is proposed in Section 6.4 to achieve efficient and flexible state saving at runtime. The issue of messaging anomalies receives great attention in Section 6.5. Both the algorithm of the NC and the WARPED kernel are enhanced significantly to address this issue. Finally, the *one log file per node* strategy is presented in Section 6.6 to break the bottleneck in the bootstrap operations.

6.1. AN ABSTRACTION FOR THE SIMULATION PROCESS

Based on the characteristics of the PCD++ message-passing paradigm, this section gives a new abstraction that allows a higher-level understanding of the simulation process on each LP.

In an event-driven simulation, simulation time advances from the timestamp of one group of simultaneous events to that of the next group. Therefore, from a computational standpoint, the sequential simulation on a LP can be viewed as a sequence of computation units, one for each group of simultaneous events, transforming the system mapped on that processor according to the P-DEVS formalism. Each computation unit is performed by the hosting processor during a span of time as measured by a physical wall clock. Such computation unit is referred to as *wall clock time slice* (WCTS). A WCTS comprising simultaneous events occurred at virtual time t is denoted as *WCTS-t*, and t is called as *the virtual time of the WCTS*.

The WCTS is an entirely different concept from the "time slice" used in discrete time modeling approach, in which the simulation time is divided into a sequence of equal-sized time steps, each step is called as a time slice, and the simulation advances from one time step to the next. On the contrary, the physical execution time of the simulation on a LP is subdivided into a series of wall clock time slices with not necessarily equal lengths, one for a group of simultaneous events executed at a specific virtual time, and the simulation time jumps from the virtual time of a WCTS to the next.

The sequential simulation on a LP can be represented in terms of WCTS, as shown in Figure 39.

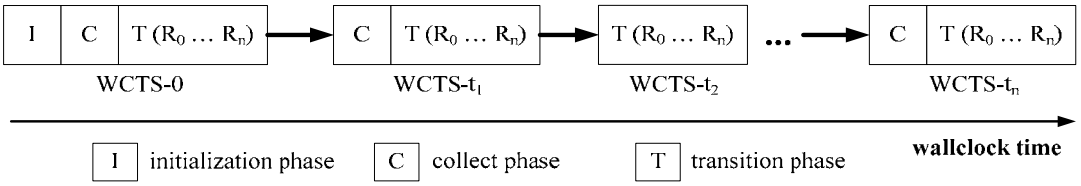


Figure 39. WCTS representation for the simulation on a LP

In the diagram, the details of PCD++ processors and message exchanges between them disappear. Instead, the simulation on a LP is viewed as a sequence of wall clock time slices linked together along the time axis, each stands for the execution of simultaneous events at a specific simulation time on all the PCD++ processors associated with this LP according to the P-DEVS formalism. Also, each WCTS-t may contain one mandatory transition phase and one optional collect phase. Using the WCTS abstraction makes the otherwise daunting task of analyzing potentially huge number of PCD++ processors involved in the simulation and the complex message exchanges between them manageable.

Several properties of the WCTS are summarized as follows:

(1) The simulation on a LP starts with WCTS-0, the only WCTS with all three phases.

(2) Wall clock time slices are linked together by messages sending from the NC to the FC (shown as black arrows in the diagram). When the NC determines the next simulation time at the end of a WCTS, it sends out messages that will be executed by the FC at the new simulation time, initiating the next WCTS on the LP. Hence, the messages linking two adjacent wall clock time slices have send time equal to the virtual time of the previous WCTS and receive time equal to that of the next. For example, the linking messages between WCTS-t_1 and WCTS-t_2 have send time of t_1

and receive time of t_2. All other messages executed in a WCTS have the same send and receive time that is equal to the virtual time of the WCTS.

(3) The completion of the simulation on a LP is marked by a WCTS sending out no linking messages, e.g. WCTS-t_n in the diagram. The NC on that LP enters into a special state called *dormant*. The whole simulation finishes only when all participating LPs have completed their corresponding parts of the simulation. A dormant NC may be reactivated later by messages from other remote NCs and subsequently initiates more wall clock time slices on the receiving LP. More details on the dormant state of the NC will be covered in the next section.

(4) Wall clock time slices are *atomic* computation units during rollback operations. Since the NC is the only PCD++ processor that receives messages from other LPs during the simulation, rollbacks are typically triggered by a remote straggler or anti-message at the NC. In such case, the NC is the local rollback originator, and the rollbacks are propagated from the NC to the other local processors. During the process, anti-messages may be sent to other LPs, triggering further rollbacks on those LPs. Let's focus our analysis on a single LP. A typical rollback scenario is shown in Figure 40.

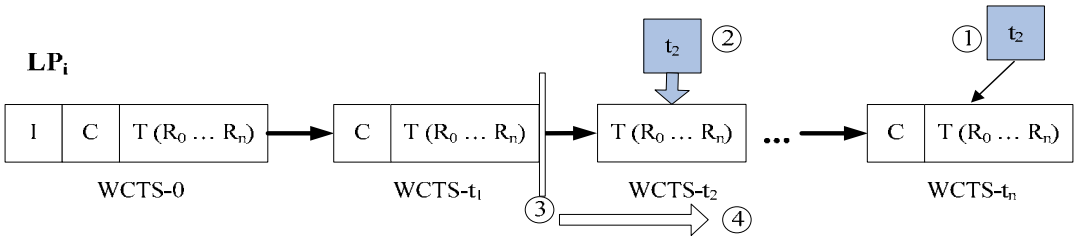


Figure 40. Typical rollback scenario shown in terms of wall clock time slices

In the diagram, the simulation on LP_i is executing in WCTS-t_n when a straggler or anti-message with timestamp t_2 arrives at the NC (action 1). Based on the kernel rollback mechanisms, the received straggler or anti-message is inserted into WCTS-t_2 (a message implosion happens in WCTS-t_2 if it is an anti-message) (action 2). Then, the rollbacks are propagated among the PCD++ processors, restoring their states to those saved at the end of WCTS-t_1 (action 3), and all messages in WCTS-t_2 up to WCTS-t_n are undone. After the rollbacks, the simulation on LP_i resumes forward execution from the unprocessed linking messages between WCTS-t_1 and WCTS-t_2 (action 4). Simply put, the arrival of a straggler or

anti-message modifies the WCTS to which it belongs, and the simulation resumes execution from the modified WCTS after the rollbacks, taking the straggler or anti-message into account.

However, rollbacks may also be initiated at the FC instead of the NC due to messaging anomalies occurred in the simulation. In this case, the NC needs to perform a series of cleanup operations after the kernel rollbacks, which will be discussed in Section 6.5.

6.2. DORMANT STATE OF NODE COORDINATORS

In optimistic simulations, LPs are allowed to execute as fast as they can. Therefore, some LPs may have processed all their local events while waiting for other lagging-behind LPs to finish their work in order to complete the whole simulation. Meanwhile, the lagging-behind LPs may send messages to the waiting LPs and thereby reactivate them. These messages may or may not trigger rollbacks on the waiting LP. If rollbacks happen, the waiting LP will be reactivated automatically by the WARPED kernel, simply because some events on that LP are unprocessed during the rollbacks and will be executed by the LTSF scheduler afterwards. Hence, we only need to consider how to reactivate the waiting LP if no rollback happens.

To this end, we define a special state called *dormant* for the NC. The NC enters into the dormant state once all local events have been processed on its associated LP. Later, if the NC receives messages from other running LPs, it exits the dormant state and reactivates the simulation on its LP again. The whole simulation ends when the NCs on all the LPs have entered into the dormant state, and the GVT is advanced to infinity.

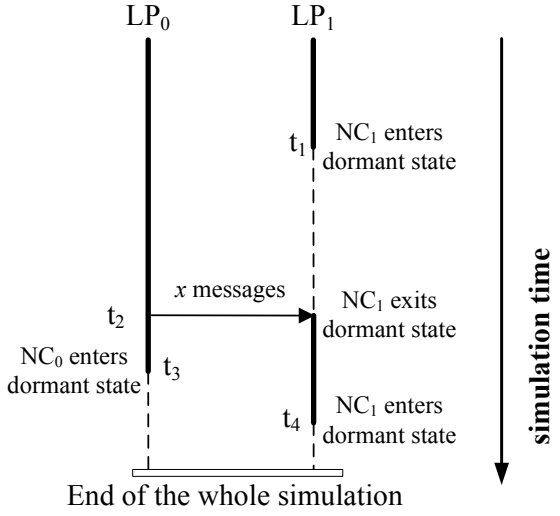


Figure 41. Example scenario for state changes of the NC during the simulation

Figure 41 shows an example scenario involving two LPs, where the running state of a LP is represented by solid lines and the waiting state is illustrated with dashed lines. In this example, all local events on LP_1 are finished at virtual time t_1. Since there is no unprocessed event left in the input queue, the simulation time on LP_1 jumps to infinity. As a result, NC_1 enters the dormant state at time t_1. Meanwhile, LP_0 keeps executing its events and NC_0 sends external messages to NC_1 at time t_2, reactivating the simulation on LP_1. Hence, NC_1 exits the dormant state at t_2, and the simulation time on LP_1 jumps back from infinity to t_2. Both LPs run from time t_2 to t_3. Then, LP_0 finishes all its events at t_3 and NC_0 enters the dormant state. The simulation time on LP_0 jumps from t_3 to infinity. LP_1 continues execution until time t_4 when its local events are also finished and NC_1 enters the dormant state as well. Thus, the simulation time on LP_1 is advanced from t_4 to infinity. Eventually, the GVT advances to infinity and the whole simulation ends.

We now give the algorithms for the NC to enter into and exit from the dormant state. The NC enters into the dormant state once the computed next simulation time is greater than the stop time (line 10 in Figure 26), indicating that all the events on the LP have been processed. Figure 42 shows the code snippet for entering the dormant state in the NC algorithm for (*D, t*). The complete NC algorithm for (*D, t*) will be presented in Section 6.5 after more enhancements are added to it in the following sections.

```
10.        if min-time > stop-time then
11.            dormant = true
```

Figure 42. Code snippet for entering dormant state in the NC algorithm for (*D, t*)

The NC exits the dormant state and reactivates the simulation on its LP upon the arrival of external messages from other remote NCs. In this case, the NC spontaneously flushes the received external messages in its *NC Message Bag* (without the presence of a (*D, t*) from the child FC), followed by a (**, t*), to the FC to reactivate the simulation on the LP.

The enhanced NC algorithm for (*x, t*) is shown in Figure 43. As usual, the (*x, t*) is inserted into the *NC Message Bag* (line 2). The reactivation operation is performed based on three conditions: (1) the NC needs to be in the dormant state to ensure that the spontaneous reactivation will not interfere with the normal execution of the simulation (line 4); (2) the timestamp of the external messages must be less than or equal to the stop time in order to maintain a clear cut of the simulation results at the user-specified stop time (line 4); (3) all external messages with the same minimum timestamp need to be inserted into the *NC Message*

Bag so that they can be processed in a lump to reduce the potential number of rounds in the transition phase at this virtual time (line 5).

```
1.  when a (x, t) is received from a remote NC
2.          insert message x to the NC Message Bag
3.          bag-time = the time of the NC Message Bag
4.          if (dormant = true) & (bag-time <= stop-time) &
5.             (all events in the input queue with timestamp = bag-time have been processed) then
6.                      dormant = false
7.                      t_L = bag-time; ta = 0
8.                      for each x in the NC Message Bag with bag-time do
9.                              send (x, t) to the child FC
10.                     end for each
11.                     remove all x in the NC Message Bag with bag-time
12.                     send (*, t) to the child FC
13.                     next-message-type = @
14.         end if
15. end when
```

Figure 43. Enhanced NC algorithm for (*x, t*)

To reactivate the simulation on the LP, the NC first resets the *dormant* flag to exit from the dormant state (line 6). The simulation time is set to the current time of the *NC Message Bag* (line 7). Then, all external messages with the minimum timestamp are flushed to the FC and removed from the *NC Message Bag* (line 8 to 11). The NC also sends a (**, t*) to the FC to trigger the appropriate state transitions at the receiving Simulators (line 12). The *next-message-type* is set to @ accordingly (line 13).

6.3. HANDLING ROLLBACKS AT TIME ZERO

Kernel rollback operations rely on correctly restoring the states of the processes to those saved ones with virtual time *strictly less than* the rollback time. However, the problem of handling rollbacks at virtual time 0 is left unsolved in the kernel. If a process receives a straggler with timestamp 0, the state restoration will fail since no state with negative virtual time can be found in the state queue. This problem is illustrated in Figure 44.

Two LPs are involved in the simulation as shown in Figure 44(a). While LP_0 is running at virtual time t_n, LP_1 is still executing events at time 0. Then, LP_1 sends a straggler with timestamp 0 to LP_0, triggering rollbacks on LP_0. Figure 44(b) shows the details of the rollbacks on LP_0. Upon receiving the straggler (action 1), the NC on LP_0 tries to restore to a state that was previously saved *before* virtual time 0 (the current rollback time). Of course, no such state can be

found in its state queue (action 2). Therefore, the rollback operation fails (action 3), resulting in a runtime crash.

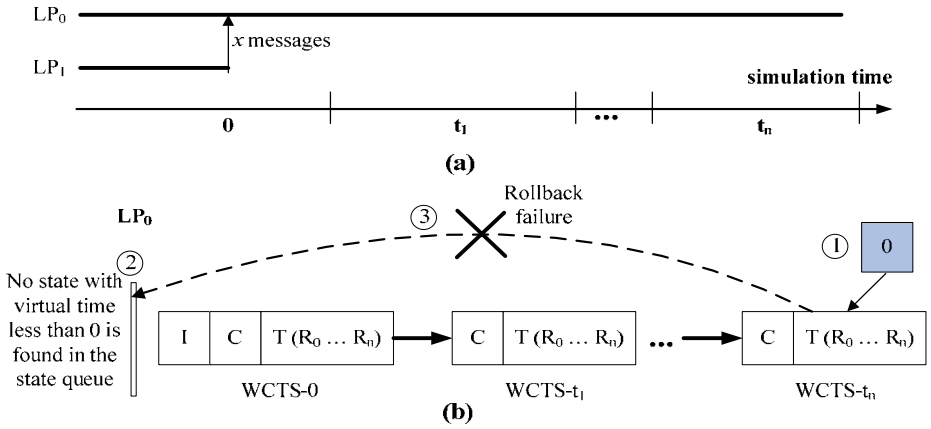


Figure 44. Rollback at virtual time 0

There are basically two different approaches to solving this problem: one is to save a special state that has an artificial negative virtual time at the head of each state queue, and let the process bounce back from it using the standard kernel rollback mechanisms; the other is to synchronize the LPs at an appropriate stage with MPI Barriers so that no straggler message with timestamp 0 will ever be received by any process in the simulation.

Both approaches have their relative advantages and associated overheads. The former is a pure optimistic approach in the sense that no explicit synchronization is used. The direct cost of this approach is small. Only a special state is added to each state queue, containing all necessary information based on which the execution of a process can restarts. However, there is a performance hazard in this approach. The probability of *rollback echoes* [Fuj00] increases significantly at virtual time 0. Take the previous example, LP_0 can successfully perform the rollbacks and resume forward execution from WCTS-0 based on this approach. In the meantime, LP_1 finishes its execution at time 0, and the simulation time on LP_1 is advanced to t_1. Then, LP_0 may send a message with timestamp 0 back to LP_1, forcing it to restart execution from time 0 as well. This scenario can happen repeatedly on the LPs, resulting in an unstable situation where there is no progress in simulation time as the simulation proceeds.

On the other hand, the second approach tries to avoid the problem altogether by using explicit synchronizations. The best place to implement the MPI Barrier is after the collect phase in WCTS-0, as illustrated in Figure 45.

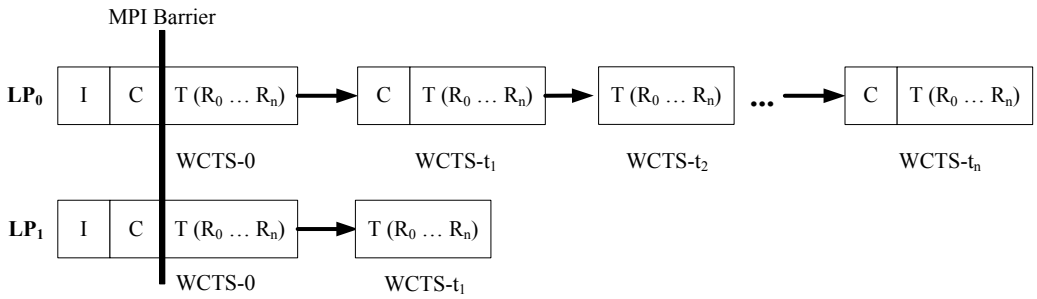


Figure 45. Using MPI Barrier to avoid rollbacks at virtual time 0

As mentioned in Section 5.6, outgoing inter-LP communication happens only in the collect phases. Hence, messages with timestamp 0 are sent to remote LPs only in the collect phase of WCTS-0. The LPs are synchronized by a MPI Barrier at the end of this collect phase so that these messages can be received by their destinations before the simulation time advances beyond time 0. Therefore, no straggler with timestamp 0 will be received by any LP afterwards. Once the LPs exit from the barrier, they can safely continue optimistic execution based on the standard kernel rollback mechanisms. The states saved for the events executed at virtual time 0 provide the necessary cushion for later rollbacks on the processes. The key is to keep the synchronized execution as short as possible. From the diagram, we can see that the execution of the LPs is synchronized during the initialization and collect phase in WCTS-0. That is, we sacrifice some of the potential benefit of optimistic execution during this period. However, this cost is small since the length of the synchronized execution is trivial compared with the whole simulation. Further, this approach is relatively easy to be implemented in PCD++: First, the MPI Barrier (which is provided at the MPI layer) needs to be wrapped in a public service function, called as *synchronizeLPs*, in the WARPED kernel so that the PCD++ processors can invoke it when necessary. Secondly, this service function is invoked in the NC algorithm for (*D, t*) once the end of the collect phase in WCTS-0 is detected.

Based on the above analysis, we chose to implement the latter approach in PCD++. The pseudo-code snippet is shown in Figure 46, which is inserted between line 2 and line 3 in the previous NC algorithm for (*D, t*) (Figure 26 in Section 5.5.3).

```
2.1. if (t = 0) & (D.ta = 0) & (next-message-type = *) then
2.2.        call kernel service function synchronizeLPs()
2.3. end if
```

Figure 46. Code snippet for handling rollbacks at time 0 in the NC algorithm for (*D, t*)

The end of the collect phase in WCTS-0 is detected by the NC using three conditions: (1) the current simulation time is zero; (2) the value of *sigma* (ta) in the received (*D, t*) is also zero; and (3) the current *next-message-type* is internal. Once found, the NC simply invokes the *synchronizeLPs* service function that has been implemented in the kernel.

6.4. USER CONTROLLED STATE SAVING MECHANISM

Two kinds of state-saving strategies are provided in the WARPED kernel, namely the *copy state-saving* (CSS) strategy and the *periodic state-saving* (PSS) strategy. They are realized using different types of state managers. The former is enforced by state managers of type *StateManager*, which saves the state of a simulation object after executing each event. The latter is implemented by state managers of type *InfreqStateManager* that only saves a simulation object's state infrequently every a number of events. Simulator developers can choose either type of state managers at compile time. Once selected, all the simulation objects will use the same type of state managers throughout the simulation. This rigid mechanism has two major disadvantages: First, it ignores the fact that simulator developers may have the knowledge on how to save states more efficiently to reduce the state-saving overhead. Secondly, it eliminates the possibility that different simulation objects may use different types of state managers to fulfill their specific needs at runtime.

To overcome these shortcomings, we implemented a two-level *user-controlled state-saving* (UCSS) mechanism in the kernel so that simulator developers can utilize more flexible and efficient state-saving strategies at runtime. This section focuses on the interplay between the UCSS and the CSS strategy, while the integration of the UCSS and the PSS strategy will be presented in Section 7.2.

In order to directly control the state-saving operation at runtime, we defined a flag called *skip-state-saving* with an initial value of false in each simulation object. If it is set to true by a simulation object, the state-saving operation will be skipped. Hence, a simulation object can make state-saving decisions based on application-specific criteria. The resulting UCSS mechanism has a two-level structure, as shown in Figure 47. At the bottom level, the CSS strategy is implemented by the *StateManager* as usual, and on top of that, the application-specific state-saving policy is governed by the *skip-state-saving* flag, which has a higher priority

than the *StateManager*.

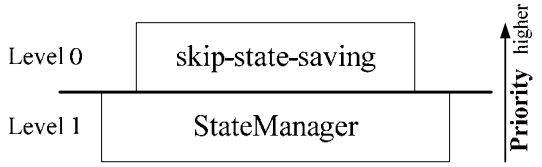


Figure 47. UCSS structure with copy state-saving strategy

The kernel algorithm, *executeSimulation*, for executing events and saving states with the *StateManager* is modified to realize the UCSS mechanism. The enhanced algorithm is shown in Figure 48, where the modifications are highlighted in bold font. The logic is straightforward. The *executeProcess* function is invoked to trigger the appropriate message-processing algorithm defined in the simulation object. The CSS policy, implemented in the *saveState* function of the *StateManager* (line 8), only takes effect when the *skip-state-saving* is false. Otherwise, no state is saved after executing the current event. Instead, the flag is reset to false so that a new state-saving decision can be made during the execution of the next event (line 6). That is, the UCSS operates on an event-by-event basis for each simulation object.

```
1. when the executeSimulation function is invoked
2.          set the inPos in the current state = the current event to be executed in the input queue
3.          executeProcess()
4.          set the outPos in the current state = the last event in the current output queue
5.          if skip-state-saving = true then
6.                    skip-state-saving = false
7.          else
8.                    call StateManager's saveState()
9.          end if
10. end when
```

Figure 48. Enhanced kernel algorithm for executing events and saving states (UCSS)

During rollbacks, the state of a PCD++ processor is always restored to the *last* state saved at the end of a WCTS with virtual time strictly less than the present rollback time. Hence, it is sufficient for a processor to save its state only after processing the last event in each WCTS for rollback purposes. The state-saving operation can be safely skipped after executing all the other events. From the PCD++ message-passing paradigm, we can see that the last event in a WCTS is processed at the end of R_n in the transition phase. Although the actual number of rounds in a transition phase cannot be determined for sure, we can at least identify the *type* of the messages executed at the end of the transition phases by a given processor. For the NC and FC, it must be a (*D, t*), and for the Simulators, it should be a (**, t*). Therefore, PCD++ processors need to save

states only after processing these particular types of messages. Since the Root only processes output messages, it still saves state for each event. The resultant state-saving strategy is called as *message type-based state-saving* (MTSS), a specific type of UCSS for the PCD++ toolkit.

Considering that there are a large number of messages executed in each WCTS and they are dominated by external and output messages, MTSS can significantly reduce the number of states saved during the simulation when compared with the original CSS strategy. In some cases, reductions of up to 30% of memory consumption have been observed in our experiments. Further, the overhead of rollbacks is reduced as well because fewer states need to be removed from the state queues during rollback operations. Unlike the PSS strategy, MTSS is risk-free in the sense that there is no penalty for saving fewer states.

The MTSS strategy can be easily implemented at the PCD++ layer using the UCSS mechanism. A processor simply sets the *skip-state-saving* flag to true in all but the algorithm for the required type of messages. For example, a Simulator will set the flag to true in its algorithms for (*I, t*), (*@, t*), and (*x, t*). This flag is left untouched with value false in its algorithm for (**, t*) since the Simulator should save its state after processing such type of messages.

6.5. MESSAGING ANOMALIES

As discussed in Section 5.5.3, the NC calculates the next simulation time based on the time of its *NC Message Bag*. However, more lagging external messages with timestamp less than the resulting simulation time may arrive after the calculation, invalidating the previous computation result. In this case, the NC's speculative calculation of the next simulation time leads to *messaging anomalies* that cannot be recovered by the kernel rollback mechanisms alone. Messaging anomalies will be detected when the control returns to the NC in the transition phase of the next (wrong) simulation time. Once found, the NC needs to perform cleanup operations to restore the simulation to the status before the previous wrong computation.

6.5.1. Speculative computation of the Node Coordinator

Figure 49 shows an example scenario, where the simulation on the LP involves three PCD++ processors (the Simulator is labeled as S1). The execution sequence of the messages is denoted by the numbers in the diagram. Only the final portion of WCTS-t_a is illustrated.

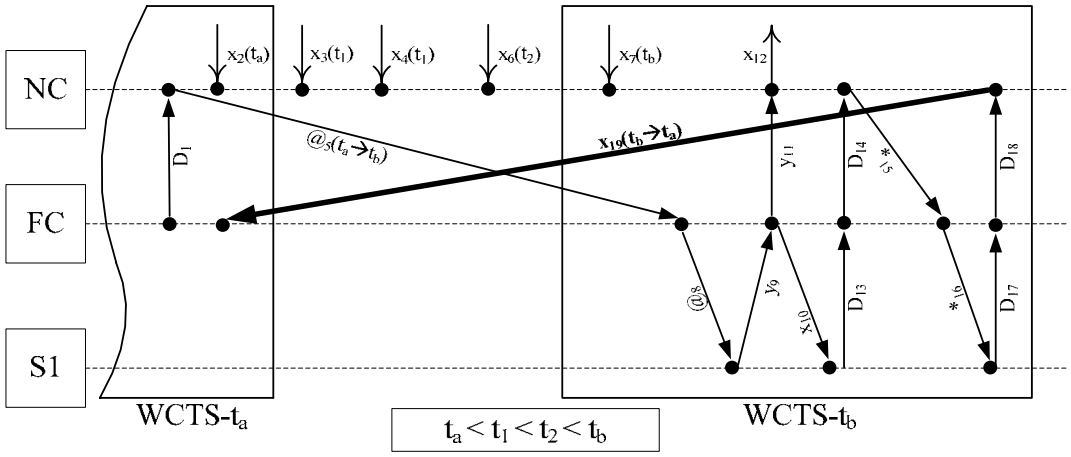


Figure 49. Example scenario of messaging anomalies

Suppose that, when the last done message (D_1) from the FC is executed by the NC at the end of WCTS-t_a, there is neither external messages in the *NC Message Bag* nor external events in the *Event List* and the closest state transition time carried in D_1 is t_b. Hence, the NC calculates the next simulation time as t_b. Consequently, it sends a collect message ($@_5$) with send time t_a and receive time t_b to the FC. However, before $@_5$ is executed by the FC, three external messages (x_2 with receive time t_a, x_3 and x_4 with receive time t_1) arrive at the NC. Since these messages have smaller timestamp than $@_5$, they are immediately inserted into the *NC Message Bag*. The LVT in the NC is thus advanced to t_1. The arrival of these external messages invalidates the previously computed next simulation time t_b, but this wrong calculation has not yet been detected.

The collect message $@_5$ is then executed by the FC, starting the collect phase of WCTS-t_b. Meanwhile, two more external messages (x_6 with receive time t_2, and x_7 with receive time t_b) arrive at the NC, and get inserted into the *NC Message Bag*. Notice that no rollback happens since the timestamps of these two messages are greater than the NC's current LVT when they are executed (i.e. $t_2 > t_1$ and $t_b > t_2$ for x_6 and x_7 respectively). The collect phase of WCTS-t_b continues, executing messages in the range of [$@_8$, D_{14}]. At this moment, the LVT in all processors has been advanced to t_b. At the end of the collect phase of WCTS-t_b, the NC sends an internal message ($*_{15}$) to the FC. Thus, the simulation enters into R_0 of the transition phase in WCTS-t_b. At the end of R_0, a done message (D_{18}) is sent to the NC from the FC. During processing D_{18}, the NC computes the next simulation time again based on the current *NC Message Bag*, which now contains five external messages (x_2, x_3, x_4, x_6 and x_7). The NC finds that the minimum timestamp is t_a, the timestamp of x_2. Hence, it sends an external message (x_{19})

with send time t_b and receive time t_a ($t_b > t_a$) to the FC, as shown by the bold arrow in the diagram. However, x_{19} is a straggler message for the FC since its timestamp is less than the FC's current LVT. According to the kernel rollback mechanism, x_{19} is inserted into both the NC's output queue and the FC's input queue, and rollbacks propagate from the FC to the other processors immediately.

Nonetheless, the rollbacks happened here is different from those discussed previously. Two kernel assumptions as described in Section 4.1 are violated: First, the rollback at the FC is triggered by an ***abnormal straggler message*** (x_{19} in the example) with a send time *greater* than the receive time, which violates Kernel Assumption 3. Since the events are ordered by their send time in the output queues, this abnormal straggler message is misplaced in the NC's output queue, resulting in causality errors and runtime crash later on during the simulation. Secondly, the rollbacks occur right *in the middle of* processing the done message (e. g. D_{18}) by the NC. This violates Kernel Assumption 7, which demands that all rollbacks should be carried out between event executions. Therefore, the rollbacks are not transparent to the NC any more. For example, when the NC regains control after the kernel rollbacks, it needs to handle its current state (which has been restored during the rollbacks) properly after processing the current done message D_{18}. The NC is also responsible for removing the abnormal straggler message from its output queue and the FC's input queue to avoid runtime crash later on. The ***false messages*** derived from the wrong calculation of the next simulation time (e.g. $@_5$) need to be handled properly by the NC as well. Further, the state saved after the wrong calculation (i.e. state saved after processing D_1) contains incorrect data that must be recovered (e.g. its *outPos* points to the false message $@_5$). Without removing the incorrect data in this state, restoration to it in later rollbacks will cause failure of the simulation. In short, the NC has to perform a series of cleanup operations when it regains control after the kernel rollbacks in the midst of executing the done message (e.g. D_{18}).

6.5.2. Two types of messaging anomalies

Let's denote the simulation time at which the NC makes the speculative computation as t_a and the calculated next simulation time as t_b. After the calculation, more external messages with timestamp less than t_b arrive, and messaging anomalies will occur at the end of R_0 of the transition phase of WCTS-t_b when the done message from the FC is processed by the NC. We only need to consider the cases where external messages with timestamp in the range of [t_a, t_b]

get inserted into the *NC Message Bag* after the speculative calculation. If external messages with timestamp less than t_a arrive, all the processors on this LP will be rolled back to a previous virtual time. These are the normal rollbacks initiated at the NC that can be handled by the kernel without problem. On the other hand, external messages with timestamp greater than t_b will be simply inserted into the input queue, and they will not be executed by the NC until their virtual time comes. In the range of $[t_a, t_b]$, whether there are external messages with time t_a inserted into the *NC Message Bag* is crucial in determining the NC's cleanup operations for recovering from the messaging anomalies.

Figure 50 shows the case that external messages with timestamp t_a are inserted into the *NC Message Bag*. The wrong computation of simulation time is found at the end of R_0 in WCTS-t_b when the NC sends an abnormal straggler message with send time t_b and receive time $\mathbf{t_a}$ to the FC, triggering rollbacks at the FC.

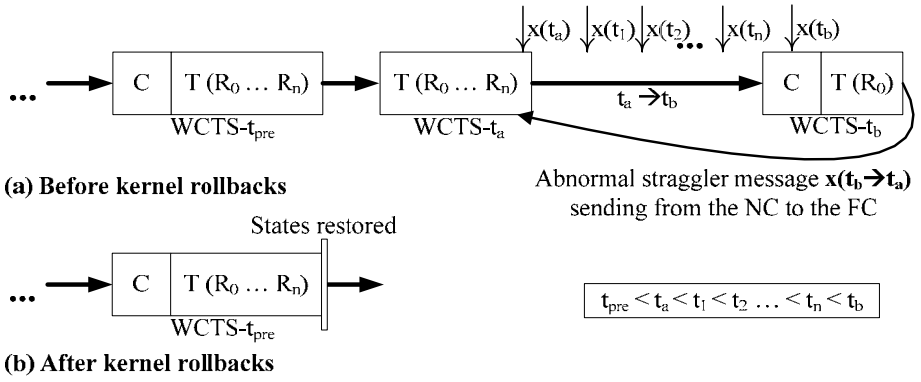


Figure 50. Messaging anomaly with empty NC Message Bag

Since the rollback time is $\mathbf{t_a}$, the FC sends anti-messages with minimum timestamp $\mathbf{t_a}$ to all the other processors, including the NC. After the kernel rollbacks, all processors on this LP are rolled back to *before* WCTS-t_a. Shown in Figure 50(b), the states of the processors are restored to those previously saved at the end of WCTS-t_{pre}, the WCTS before WCTS-t_a. All the lagging external messages (i.e. $x(t_a)$, $x(t_1)$, $x(t_2)$, … $x(t_n)$, and $x(t_b)$) are removed from the *NC Message Bag*, and their corresponding kernel events are unprocessed during the rollbacks. Then, the control is returned to the NC, which is still in the middle of processing the done message received at the end of R_0 in WCTS-t_b. At this point, the NC needs to perform the necessary cleanup operations. When the NC returns from its algorithm for (*D, t*), the simulation resumes forward execution starting from the unprocessed messages previously output from WCTS-t_{pre}. As all the lagging external messages are removed from the *NC Message Bag* and the states of the

processors are restored to the end of the previous WCTS during the kernel rollbacks, no erroneous data is left in the state queues. Therefore, the cleanup operations are relatively simple. This type of messaging anomalies is called as *anomaly with empty NC Message*.

On the other hand, if no external message with timestamp t_a arrives, the situation becomes more complex as shown in Figure 51.

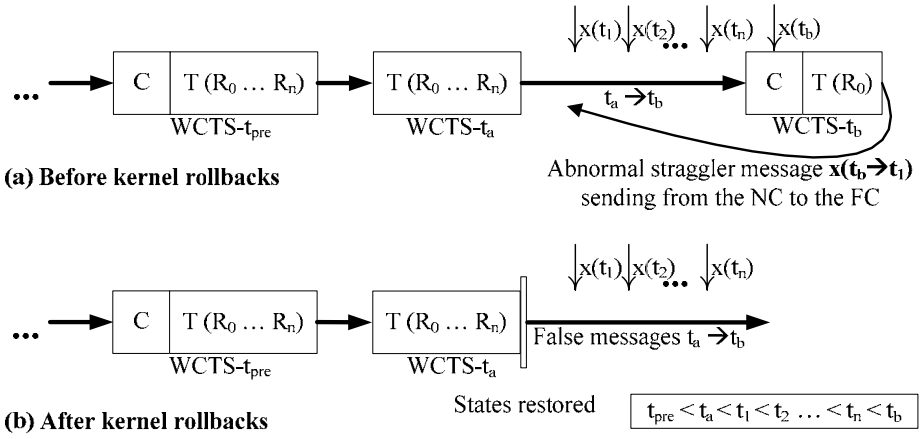


Figure 51. Messaging anomaly with non-empty NC Message Bag

Shown in Figure 51(a), the NC sends an abnormal straggler message with send time t_b and receive time $\mathbf{t_1}$ to the FC at the end of R_0 in WCTS-t_b. The rollback time for the FC is $\mathbf{t_1}$. Hence, the FC only sends anti-messages with minimum timestamp $\mathbf{t_b}$ to the other processors, including the NC. When the kernel rollbacks finish, the states of the processors are restored to those previously saved at the end of WCTS-t_a. Thanks to the MTSS strategy, the NC did not save states after processing the lagging external messages. Therefore, its state is also restored to the last state that was saved at the end of WCTS-t_a. As explained earlier, the restored state contains incorrect data that must be recovered by the NC itself after the kernel rollbacks.

During the kernel rollbacks, all messages executed by the FC and the Simulators in WCTS-t_b are cancelled. Nonetheless, only the external messages with time t_b are removed from the *NC Message Bag*. The other lagging external messages (i.e. $x(t_1)$, $x(t_2)$, … and $x(t_n)$) remain in the *NC Message Bag* after the kernel rollbacks. Also, the false messages originally sent from WCTS-t_a are unprocessed and kept in the FC's input queue as well as the NC's output queue, as shown in Figure 51(b). After the rollbacks, the LVT in the NC is reset to $\mathbf{t_n}$, while the LVT in the FC and the Simulators is restored to $\mathbf{t_a}$. In this case, more complex cleanup operations need to be done by the NC to erase all the wrong data left in the kernel. Since the *NC Message Bag* still

contains external messages after the kernel rollbacks, this type of messaging anomalies is called as *anomaly with non-empty NC Message Bag*.

6.5.3. Anomaly with empty NC Message Bag

In this section, we present the algorithm for handling anomaly with empty NC Message Bag in the NC algorithm for (*D, t*), as shown by the code snippet in Figure 52.

```
1. when cleanup operations for anomaly with empty NC Message Bag is invoked
2.         abnormal-straggler = NC's output queue→remove(tail)
3.         FC's input queue→removeStragglerEvent(abnormal-straggler)
4.         skip-state-saving = true
5. end when
```

Figure 52. NC algorithm for handling anomaly with empty NC Message Bag

Once an anomaly with empty NC Message Bag is detected after the kernel rollbacks, the NC needs to perform the following cleanup operations in its algorithm for (*D, t*):

(1) Remove the abnormal straggler message from the NC's output queue (line 2). Since this straggler message is saved at the end of the NC's output queue, we can directly remove it using the *remove* function provided by the kernel. This function also returns a reference to the message that has been removed, which can then be used to remove the same straggler message from the FC's input queue.

(2) Remove the abnormal straggler message from the FC's input queue (line 3). This is more difficult than the previous operation. The kernel does not provide function for removing a positive event from the input queue except during event implosion, where the positive event is annihilated by an incoming anti-message. Hence, we defined a new function called *removeStragglerEvent* for this purpose. The logic of this function is similar to that for event implosion. However, the positive message is annihilated by a reference to itself rather than the counterpart anti-message.

(3) Skip the state-saving operation after processing the current done message (line 4). During the kernel rollbacks, the state of the NC has been restored to the last state saved at the end of WCTS-t_{pre} (see Figure 50). Therefore, the NC should not save its current state after processing the present done message. Using the UCSS mechanism, the NC only needs to set *skip-state-saving* to true to do this.

After these cleanup operations, the NC returns from its algorithm for (*D, t*) and the simulation resumes forward execution.

6.5.4. Anomaly with non-empty NC Message Bag

The cleanup operations for anomalies with non-empty NC Message Bag are much more complex. Let's take a closer look at the status of the NC after the kernel rollbacks, as shown in Figure 53. The wrong computation of the next simulation time is made during executing D_a in WCTS-t_a. After processing D_a, a state that contains incorrect data resulting from the wrong computation is saved in the NC's state queue (referred to as S_a in the diagram). This wrong computation is detected by the NC at the end of R_0 in WCTS-t_b, in the middle of executing D_b. An abnormal straggler message is sent to the FC, triggering kernel rollbacks. During the rollbacks, the NC's state is restored to S_a.

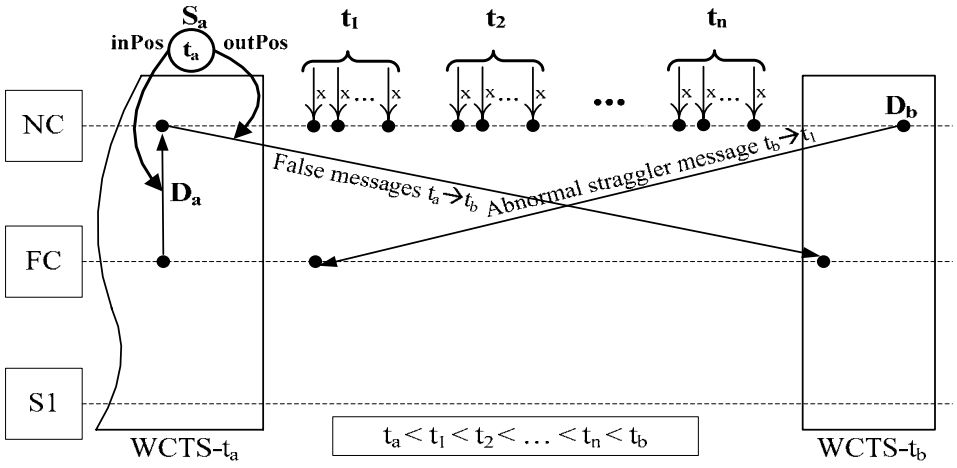


Figure 53. NC status during anomalies with non-empty NC Message Bag

In Figure 51, only one lagging external message for each distinct timestamp (from t_1 to t_n) is shown for simplicity. Actually, as illustrated in Figure 53, multiple external messages may coexist for each timestamp after the kernel rollbacks, and those messages with timestamp t_1 are sent to the FC in batches. For example, if there are three external messages with timestamp t_1 in the *NC Message Bag*, the first one sent to the FC will be the abnormal straggler message (with send time t_b and receive time t_1), which will trigger the rollbacks. The following two external messages will be sent to the FC right after the kernel rollbacks when the control is returned to the NC's logic for sending external messages. However, the send time of these two messages is changed to t_a because the current state of the NC has already been restored to S_a during the rollbacks. Of course, their receive time is still t_1. During the cleanup operations, therefore, not only the first abnormal straggler message needs to be removed from the NC's output queue and

the FC's input queue just like the operations discussed in the previous section, but also the other external messages sent along with the abnormal straggler need to be handled in the same way. After the cleanup operations, the NC will resend these external messages to the FC again, with the correct send and receive time.

- **Undue external events**

The false messages previously sent from the NC to the FC at the end of WCTS-t_a (i.e. the linking messages between WCTS-t_a and WCTS-t_b) are unprocessed and left in the NC's output queue and the FC's input queue during the kernel rollbacks. The false messages contain at least one control message, either a (*@, t_b*) or a (**, t_b*), and may optionally include multiple external messages representing the *undue external events* scheduled at time t_b. Hence, the NC needs to perform the following cleanup operations to handle the false messages:

(1) Remove the false messages from the NC's output queue and the FC's input queue. The NC needs to find all these false messages in the queues, and then remove them using the *removeStragglerEvent* function.

(2) Identify the potential undue external events sent in the false messages. Once found, the NC needs to restore the *event-pointer* in its current state to recover these undue events. Figure 54 shows an example scenario for recovering the *event-pointer*, where an external event with scheduled time *t* is depicted as E(*t*).

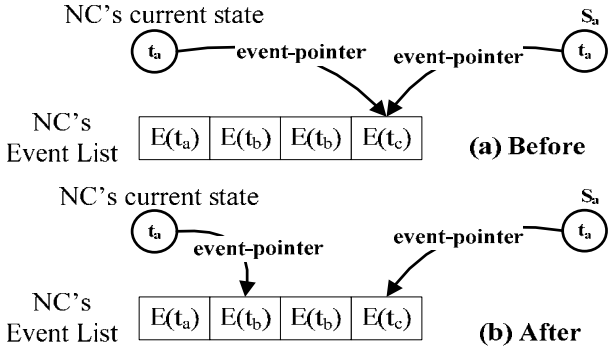


Figure 54. Restoring the event-pointer for undue external events

In this example, there are four external events in the NC's *Event List*; two of them are scheduled at time t_b. At the end of WCTS-t_a, the wrong simulation time t_b is calculated, and both $E(t_b)$ are sent in the false messages to the FC. The *event-pointer* in the NC's current state moves to $E(t_c)$ accordingly and gets saved as S_a in the state queue after processing D_a. During the following rollbacks, the NC's current state is restored to S_a. Hence, the *event-pointer* in both the

NC's current state and S_a point to $E(t_c)$, as shown in Figure 54(a). To undo the external events after the rollbacks, the NC needs to reset the *event-pointer* in its current state to the first $E(t_b)$, the original position before sending the false messages. In Figure 54(b), the *event-pointer* in S_a is left untouched during the operation. Handling the incorrect data in S_a is presented next.

- **Break point state**

An important cleanup operation is to deal with the incorrect data in S_a. While it is hard to directly modify the internal data contained in a previously saved state in the state queue, it is easier to correct the data in the NC's current state. As the NC's current state is restored to S_a by the kernel, they have identical data after the rollbacks. Mechanisms for fixing part of the data in the NC's current state have been revealed in our previous discussion, as summarized below:

(1) Removing and resending the lagging external messages with time t_1 restore the correct ordering of messages in the NC's output queue.

(2) Removing the false messages eliminates the effect of the previous wrong computation. New control message will be sent at the end of the cleanup operations, which, together with point (1), makes sure that the *outPos* in the NC's current state will points to the correct message in NC's output queue.

(3) The *inPos* in both S_a and the NC's current state remains pointing to the last done message of WCTS-t_a (e.g. D_a in Figure 53).

(4) Recovering the undue external events restores the *event-pointer* in the NC's current state to the correct position.

Another operation is that the NC needs to set the LVT value in its current state to t_1 so that all the messages resent during the cleanup operations have send time t_1 instead of t_a. This is important since the external messages at time t_1 may be cancelled later by their remote senders. In such case, the rollback time is t_1 for the NC, and the simulation on this LP will be rolled back to the end of WCTS-t_a. However, if the external messages were resent with send time t_a ($t_a < t_1$), they would not be eliminated properly since only messages with send time greater than t_a in the NC's output queue are cancelled.

The last issue is dealing with the incorrect data contained in state S_a. The idea is to replace S_a with a new copy of the NC's current state, which will be saved in the state queue when the NC algorithm for (*D, t*) returns. Removing S_a can be easily done since we know that it is now the last state in the NC's state queue. Saving a copy of the NC's current state in the state

queue is performed by the kernel automatically. However, simply replacing S_a in this way is not enough. Let's consider what happens if the lagging external messages at time t_1 are cancelled later by their remote senders. An example scenario is illustrated in Figure 55, showing the queues of the NC after the cleanup operations.

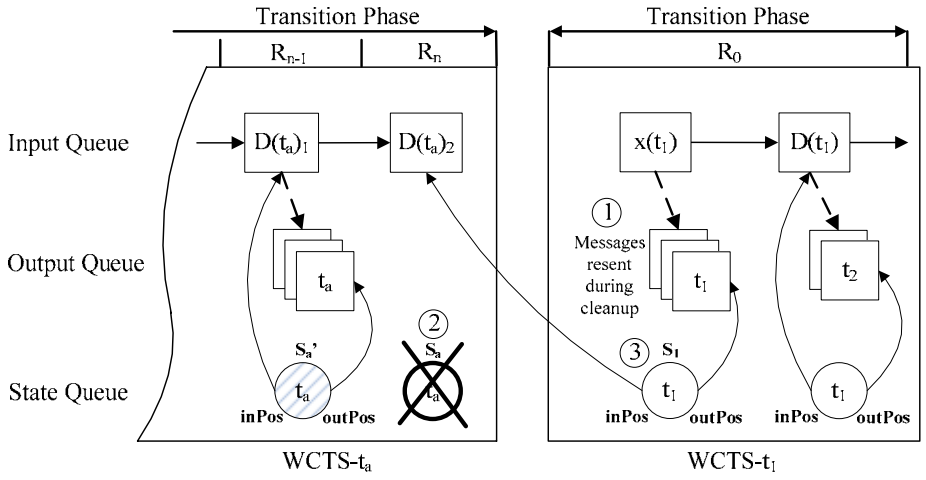


Figure 55. Example scenario for anomalies with non-empty NC Message Bag

The NC made a computation of the next simulation time t_b at the end of WCTS-t_a, during executing the last done message $D(t_a)_2$. After that, one lagging external message with timestamp t_1, $x(t_1)$, arrived, invalidating the previously computed t_b and resulting in messaging anomalies with non-empty NC Message Bag. Rollbacks were performed by the kernel. In the ensuing cleanup operations, the abnormal straggler message and the false messages are removed from the queues (not shown in the diagram). The LVT in the NC's current state is set to t_1, and correct messages are resent (action 1). Then the previous state saved after processing $D(t_a)_2$, i.e. S_a, is removed from the NC's state queue (action2). A new copy of the NC's current state, shown as S_1, is saved after the cleanup operations with the corrected data (action 3). As a result, its *outPos* refers to the last message resent during the cleanup operations and its *inPos* points to $D(t_a)_2$. The simulation continues after the cleanup operations.

Now, suppose an anti-message is received by the NC to cancel the previous lagging external message $x(t_1)$. Since the rollback time is t_1, the simulation is restored to the end of WCTS-t_a. There are two problems here:

(1) Since S_a has been removed, the NC's current state will be restored to the state saved before S_a, shown in the diagram as S_a'. However, the data contained in S_a' is stale and does not reflect the modifications that have been done during executing $D(t_a)_2$.

Also notice that, as part of the state restoration, the states following S_a' will be removed from the NC's state queue and S_1 is the first one to be removed.

(2) The linking messages between WCTS-t_a and WCTS-t_b have been removed from the queues during the previous cleanup operations. While these messages were proven false due to the arrival of the lagging external message $x(t_1)$, they are not now since it finally turns out that $x(t_1)$ itself is a false one. As a result, the simulation should resume forward execution starting from exactly these (removed) linking messages after the present rollbacks. However, the nonexistence of these linking messages results in failure of the simulation.

Both of these problems can be solved if the done message, $D(t_a)_2$, is executed by the NC again immediately after the present rollbacks. By executing this done message, the data in the NC's current state (which has been restored to S_a') is updated and new linking messages are regenerated. A copy of the NC's current state will be saved in the state queue right after S_a', filling the gap produced by the removal of S_a.

To this end, a special mechanism needs to be implemented in the kernel. The first question we need to answer is how to detect that the NC is now rolled back to the end of a WCTS where anomalies with non-empty NC Message Bag have previously occurred. To do so, the NC must leave a tag in its queues after handling the messaging anomalies. Then, the special mechanism will be triggered once this tag is detected during later rollbacks.

A flag, called as *breakpoint*, is added to the abstract state definition in the kernel for this purpose. This flag is false by default. When the NC corrects the data in its current state during the cleanup operations, it sets this flag to true in its current state. As we know, a copy of the NC's current state is saved after the cleanup operations. Hence, the *breakpoint* flag saved in S_1 is true, and S_1 is called as a *breakpoint state*. Notice that S_1 will be the first to-be-removed state during the later rollbacks that bring the NC back to the end of WCTS-t_a. Also, its *inPos* points exactly to $D(t_a)_2$, the done message that needs to be reprocessed immediately after the later rollbacks.

The resulting kernel algorithm for state restoration is shown in Figure 56. The original algorithm consists two parts: (1) restoring the current state to the last state with LVT less than the rollback time (line 2 to 8); and (2) removing all the following states after the last state found in the state queue (line 14 to 17).

```
1. when state restoration is invoked on simulation object P with rollback-time
2.         last-state = the last state with LVT less than rollback-time in P's state queue
3.         if last-state != NULL then
4.                 P's current state = last-state
5.         else
6.                 raise error
7.         end if
8.         state-handle = last-state→next
9.         if (state-handle != NULL) & (P is the NC) & (state-handle→breakpoint = true) then
10.                to-execute = state-handle→inPos
11.                undo to-execute in P's input queue
12.                cache to-execute in the LTSF scheduler to execute it after the current rollbacks
13.        end if
14.        while state-handle != NULL do
15.                remove state-handle from P's state queue
16.                state-handle = state-handle→next
17.        end while
18. end when
```

Figure 56. Enhanced kernel algorithm for state restoration during rollbacks

The enhancement for handling breakpoint states is added to the original algorithm, shown in bold font in the diagram (line 9 to 13). Before removing the states from the state queue, a test is done on the first to-be-removed state, *state-handle*, obtained in line 8. If it is a breakpoint state (line 9), the kernel undoes the message referenced by its *inPos* (line 10 and 11) and adjusts the LTSF scheduler so that this message will be executed immediately after the current rollbacks.

- **Pending wall clock time slices**

From the previous discussion, we can see that the lagging external messages left in the *NC Message Bag* after kernel rollbacks actually represent a series of pending wall clock time slices (each for the group of external messages with identical timestamp) that need to be inserted between the previous and the false WCTS. Shown in Figure 53, for example, there are n pending wall clock time slices for virtual time from t_1 to t_n to be inserted between WCTS-t_a and WCTS-t_b. These pending wall clock time slices can only be accommodated into the simulation process one by one. During the cleanup operations, the NC will resend the external messages with the least timestamp, t_1, along with the appropriate control message to the FC, initiating WCTS-t_1. When the simulation is executed in the normal mode, external messages with timestamp from t_2 to t_n cannot be processed by the NC before the completion of WCTS-t_1 simply because these messages, having timestamps greater than t_1, will not be selected by the LTSF scheduler as long as there are still messages with timestamp t_1 to be processed in the input queue. It is the false

advance of the simulation time to t_b that makes all these lagging external messages coexisting in the *NC Message Bag*, violating the local causality constraint.

Therefore, after resending external messages with time t_1, the NC needs to perform the following operations to handle the extra external messages with timestamp from t_2 to t_n in the NC Message Bag:

(1) Reset the NC's LVT to t_1. As explained earlier, the LVT in the NC is restored to t_n during the kernel rollbacks. Since we are going to execute the first pending wall clock time slice, WCTS-t_1, the NC's LVT should be reset to t_1 accordingly.

(2) Rollback the NC's file queue with a rollback time of t_2 to remove the log information, if any, for these extra external messages.

(3) Undo the corresponding kernel events for the extra external messages in the input queue. These kernel events will be processed again in batches after the completion of the current (inserted) WCTS-t_1 just like in the normal execution.

(4) Remove the extra external messages from the *NC Message Bag* to eliminate the causality errors.

We now give the algorithm for handling anomalies with non-empty NC Message Bag, as shown in Figure 57. When the anomaly is detected, the NC first removes all the external messages with time t_1, including the abnormal straggler, from the NC's output queue and the FC's input queue (line 2 to 5). Then, the NC searches the FC's input queue to find the false messages previously sent to WCTS-t_b as a result of the wrong computation of the next simulation time (line 6). The undue external events sent along with the false messages, if any, are retrieved (line 8). Once found, the *event-pointer* in the NC's current state is recovered properly (line 9 to 12). The NC also locates the false messages saved in its output queue (line 13). These false messages are removed from the NC's output queue and the FC's input queue (line 14 to 19). Then, the erroneous state saved at the end of WCTS-t_a is retrieved from the tail of the NC's state queue (line 20), and gets removed thereafter (line 21).

```
1. when cleanup operations for anomaly with non-empty NC Message Bag is invoked
2.      for each external message with time t_l in the NC Message Bag do
3.              abnormal-straggler = NC's output queue→remove(tail)
4.              FC's input queue→removeStragglerEvent(abnormal-straggler)
5.      end for each
6.      false-input-messages = FC's input queue→findFalseInputMessages(t_b)
7.      if the size of false-input-messages > 0 then
8.              undue-external-events = findUndueExternalEvents(false-input-messages)
9.              if the size of undue-external-events > 0 then
10.                     future-external-event-time = the scheduled time of the undue-external-events
11.                     rollbackExternalEvents(undue-external-event-time)
12.             end if
13.             false-output-messages = NC's output queue→findFalseOutputMessages(false-input-messages)
14.             for each message m in false-output-messages do
15.                     NC's output queue→remove(m)
16.             end for each
17.             for each message m in false-input-messages do
18.                      FC's input queue→removeStragglerEvent(m)
19.             end for each
20.             false-state = NC's state queue→tail
21.             NC's state queue→remove(false-state)
22.     else
23.             raise error
24.     end if
25.     NC's current state→LVT = t_l
26.     NC's current state→breakpoint = true
27.     for each external message x with timestamp t_l in the NC Message Bag do
28.             resend (x, t_l) to the child FC
29.     end for each
30.     remove all x in the NC Message Bag with timestamp t_l
31.     if the size of the NC Message Bag > 0 then
32.             reset the NC's LVT = t_l
33.             rollbackFileQueues(the time of the NC Message Bag)
34.             for each external message x remained in the NC Message Bag do
35.                     NC's input queue→unprocessEvent(x)
36.             end for each
37.             remove all x in the NC Message Bag
38.     end if
39.     send (*, t_l) to the child FC
40.     next-message-type = @
41. end when
```

Figure 57. NC algorithm for anomalies with non-empty NC Message Bag

The LVT in the NC's current state is modified from t_a to t_l, and the *breakpoint* flag is set to true (line 25 and 26). A copy of the NC's current state will be saved in the state queue to replace the erroneous state we have just removed. Then, the external messages with time t_l are resent to the FC and subsequently removed from the *NC Message Bag* (line 27 to 30). After this, the NC tests whether there are extra external messages with timestamp greater than t_l left in the *NC Message Bag* (line 31). If so, the NC resets its LVT back to t_l (line 32). The NC's file queues

are rolled back using the current time of the *NC Message Bag* (i.e. t_2) to erase the information that may have been logged for these extra external messages (line 33). Further, the corresponding kernel events for the extra external messages are undone in the input queue (line 34 to 36). All the extra external messages with time t_2 up to t_n are removed from the *NC Message Bag* to maintain the local causality constraint (line 37). Finally, the NC sends a (*, t_1) to the FC and sets the *next-message-type* accordingly (line 39 and 40).

6.5.5. Enhanced NC algorithm for done message

There are two issues that we have not yet discussed so far. One is detecting messaging anomalies in the NC algorithm for (*D, t*); the other is identifying the type of the anomaly after kernel rollbacks. These issues are addressed as follows:

(1) Detecting anomalies in the NC algorithm for (*D, t*). Kernel rollbacks are performed in the middle of the NC algorithm for (*D, t*) only when anomalies occur. The NC's LVT will be decreased during the rollbacks. Therefore, the NC can compare its LVT recorded before and after sending out external messages to the FC. The cleanup operations will be performed once a change in the NC's LVT is found.

(2) Identifying the type of the anomaly. After the kernel rollbacks, the NC can easily identify the type of the anomaly based on the status of its *NC Message Bag*. If the bag is empty, anomaly with empty NC Message is confirmed; otherwise, anomaly with non-empty NC Message is identified.

The enhanced NC algorithm for (*D, t*) is given in Figure 58, which combines the logic for normal execution (as shown by the simplified version in Figure 26) and that for handling both types of messaging anomalies (as shown in Figure 52 and 57 respectively).

At the beginning of this algorithm, the NC records its current LVT in *initial-LVT* (line 2). The execution continues as usual until the external messages in the *NC Message Bag* are sent to the FC (line 23 to 27), during which messaging anomalies and kernel rollbacks may happen. Then, the NC queries its current LVT again, which may have been decreased as a result of kernel rollbacks, and records the new value in *new-LVT* (line 28). The *new-LVT* is compared with the *initial-LVT*. If no change is detected, the processing continues in the normal way (line 29 to 37). Otherwise, a messaging anomaly is found, and the NC invokes the corresponding cleanup operations based on the current status of its *NC Message Bag* (line 38 to 43).

```
1. when a (D, t) is received from the child FC
2.          initial-LVT = the NC's current LVT
3.          t_L = t; t_N = t_L + D.ta
4.          if (t = 0) & (D.ta = 0) & (next-message-type = *) then
5.                      call kernel service function synchronizeLPs()
6.          end if
7.          if next-message-type = * then
8.                      send (*, t) to the child FC
9.                      next-message-type = @
10.         else
11.                     min-time = MIN( timestamp of the event pointed by event-pointer,
12.                                     time of the NC Message Bag,
13.                                     t_N )
14.                     if min-time > stop-time then
15.                                 dormant = true
16.                     else
17.                                 if min-time = the timestamp of the event pointed by event-pointer then
18.                                             for each x in the Event List with min-time do
19.                                                         send (x, t) to the child FC
20.                                                         move event-pointer to the next event
21.                                             end for each
22.                                 end if
23.                                 if min-time = the time of the NC Message Bag then
24.                                             for each x in the NC Message Bag with min-time do
25.                                                         send (x, t) to the child FC
26.                                             end for each
27.                                 end if
28.                                 new-LVT = the NC's current LVT
29.                                 if new-LVT = initial-LVT then
30.                                             remove all x in the NC Message Bag with min-time
31.                                             if t_N = min-time then
32.                                                         send (@, t) to the child FC
33.                                                         next-message-type = *
34.                                             else
35.                                                         send (*, t) to the child FC
36.                                                         next-message-type = @
37.                                             end if
38.                                 else
39.                                             if the size of the current NC Message Bag = 0 then
40.                                                         invoke cleanup operations for anomaly with empty NC Message Bag
41.                                             else
42.                                                         invoke cleanup operations for anomaly with non-empty NC Message Bag
43.                                             end if
44.                                 end if
45.                     end if
46.         end if
47. end when
```

Figure 58. Enhanced NC algorithm for (*D, t*)

6.6. ONE LOG FILE PER NODE STRATEGY

Logging facility is provided to log the messages received by the PCD++ processors during the simulation in a human readable format. In the previous versions, one log file is created for each

PCD++ processor. Depending on the size of the model, this can consume a lot of file descriptors. For complex models, the required number of file descriptors often exceeds the upper limit imposed by the underlying operating system. Also, creating these files and transferring data to them constitute a large operational overhead, especially when the files are accessed via a Network File System (NFS) during the simulation. When considering the overhead in Time Warp optimistic simulations, the cost is prohibitive since one file queue is maintained in the kernel for each of these files and all the file queues participate in rollback operations.

To reduce the overhead of file I/O operations, we implemented a new strategy, called as *one log file per node*, in the PCD++ toolkit. Based on this strategy, only one log file is created on each node for all the processors mapped onto it. The strategy is described as follows:

(1) If the user chooses to log only output messages, only a single log file is created for the FC on each node. In this case, the output messages from all the Simulators running on a node are logged in the FC's log file. Usually, this is sufficient for visualization purposes. Since the FC is the immediate destination of the output messages from the Simulators, these messages can be logged directly in the file queue that is locally maintained by the FC.

(2) Otherwise, only a single log file is created for the NC on each node for logging whatever types of messages, or all of them, as specified by the user. The NC's file queue is shared among all the processors on that node. Messages received by the NC itself are logged directly into the NC's file queue, while the other processors on that node must first get a reference to the local NC (which can be done in constant time) and then log their received messages into the NC's file queue.

The one log file per node strategy has the following advantages:

(1) The required number of file descriptors for logging purposes is upper-bounded by the number of machines used in the simulation, rather than increasing linearly with the size of the model. For example, a 30 by 30 Cell-DEVS model executed on three machines consumes totally three file descriptors (one on each machine) rather than nearly one thousand in the previous case.

(2) The bootstrap time is reduced considerably. During the bootstrap operations, the log files are created in the NFS over the network. Thanks to the dramatic decrease in the number of files opened in this process, the bootstrap time is reduced from tens

of seconds to around one second for a middle sized model.

(3) The kernel rollback operations are accelerated since only one operation is performed to restore the single file queue maintained in the kernel.

(4) The communication overhead is reduced as well. The data concentrated in a single file queue is flushed to the physical log file in bigger chunks, and less frequently, over the network.

(5) Higher scalability is allowed under this strategy. Much larger models can be executed without running out of file descriptors.

CHAPTER 7 OPTIMIZATION ALGORITHMS IN THE WARPED KERNEL

Since Jefferson's original presentation of the Time Warp mechanism, many refinements have appeared in the literature, which can be considered as falling into two distinct categories [Low99]: reducing the operational overhead of the Time Warp mechanism, and exploiting more parallelism than is available in the basic protocol. This chapter covers the integration of several optimization algorithms into the PCD++ toolkit to improve the performance. The algorithms discussed here are provided in the WARPED kernel to address both types of optimizations. The *one anti-message per rollback* strategy aiming at reducing the overhead of sending anti-messages during rollbacks is introduced in Section 7.1. The UCSS strategy originally presented in Chapter 6 is further extended to work with the *periodic state saving* (PSS) strategy to reduce the state-saving overhead, as discussed in Section 7.2. Finally in Section 7.3 the *lazy cancellation* strategy is integrated into the toolkit to exploit the parallelism available within a LP. Unfortunately, these algorithms cannot be used directly in the PCD++ toolkit as they are. Enhancements and special considerations are given more emphasis in the following discussions.

7.1. ONE ANTI-MESSAGE PER ROLLBACK

When rollback happens on a process, all messages saved in its output queue that have send time equal to or greater than the rollback time are sent to their original receivers as anti-messages. Since multiple messages may have been previously sent to a receiver, the same number of anti-messages will be sent to that receiver during the rollback, resulting in a flood of anti-messages exchanged between the processes with the concomitant high communication overhead. However, if a process has several anti-messages to send to another process, it clearly suffices to send the one with the earliest time [Lub91], reducing the number of anti-messages that need to be transmitted to a certain extent.

The one anti-message per rollback strategy consists of two parts as follows:

(1) On the sender side, a temporary queue is used to hold the anti-messages to be sent to different receivers. The anti-messages in the sender's output queue that have send time equal to or greater than the rollback time are tested in sequence. Only a single

anti-message that has the earliest timestamp for each distinct receiver is extracted and inserted into the temporary queue, others are suppressed. Then, the sender sends the anti-messages in the temporary queue to their corresponding receivers.

(2) On the receiver side, upon receiving an anti-message, all positive messages from the same sender and with ***send time*** (originally, *receive time* is used in the kernel algorithm) equal to or greater than that of the anti-message are annihilated, others are unprocessed as usual. Thus, one anti-message is now capable of cancelling multiple positive messages in the receiver's input queue.

In order to integrate this strategy into the PCD++ toolkit, we made some minor modifications to the original algorithm in the WARPED kernel. One of them is that the *send time* is used as the criterion for annihilating messages in the receiver's input queue. As hinted in (2), the original algorithm uses the *receive time*, which can cause wrong message annihilations and runtime failure. For example, suppose that an anti-message with send and receive time of 200 triggers the rollback of the FC to the end of WCTS-100. The linking messages between WCTS-100 and WCTS-200 should be unprocessed rather than imploded during the rollback. If the receive time is used, these linking messages would have been imploded since they have the same receive time as the anti-message. As a result, the simulation cannot resume forward execution after the rollback. However, if the send time is used instead, these messages will be unprocessed normally since their send time 100 is less than the send time of the anti-message. In fact, according to Kernel Assumption 3, the send time must be less than or equal to the receive time for any message in the system. Hence, our version is stricter than the previous one, confining message implosions to the appropriate scope.

There are two points that deserve attention. First, this strategy is intended to reduce the number of anti-messages transmitted during rollbacks. It does not modify the part of the rollback algorithm for handling positive straggler messages, nor does it reduce the number of rollbacks. Secondly, this strategy still belongs to the aggressive cancellation category. There is no delay in the delivering of anti-messages, only with reduced number.

7.2. PERIODIC STATE SAVING

In Time Warp optimistic simulations, the state of each process must be saved regularly, regardless of whether or not rollbacks actually occur [Lin93]. To achieve better performance, one approach for reducing the operational overhead is to decrease the number of state-saving operations. Under the PSS strategy, the state of a simulation object is saved infrequently every a number of events. As introduced in Section 6.4, the WARPED kernel implements the PSS strategy using state managers of type *InfreqStateManager*, which is a subtype of the general *StateManager* that enforces the CSS strategy. An introduction of the original algorithm of the *InfreqStateManager* is given in Section 7.2.1. Section 7.2.2 covers the extension of our UCSS strategy to incorporate with the PSS strategy. The PSS strategy is integrated into the PCD++ toolkit using the additional flexibility made available by the enhanced UCSS mechanism, as discussed in Section 7.2.3. Finally, Section 7.2.4 describes the enhancements to the kernel fossil collection algorithm to address the specific requirements imposed by the PSS strategy.

7.2.1. Strategy description

Figure 59 shows the original implementation of the PSS strategy in the WARPED kernel using a period of 2. The *InfreqStateManager* uses an integer, called as *state-period*, to control the state-saving interval in terms of WCTS. The value of *state-period* can be set by users at compile time. Once set, this value will not change during the simulation. That is, the state manager implements the PSS strategy using a static state-saving interval.

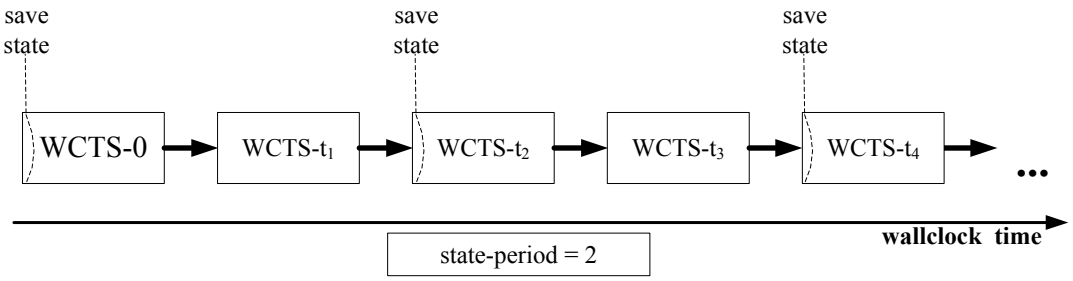


Figure 59. Periodic state-saving strategy with a static interval of 2

In the example, the *state-period* is set to 2. Therefore, the states of all the processes on this LP will be saved every two wall clock time slices (e.g. for virtual time 0, t_2, t_4, t_6, and so on). Further, the state of a process is saved only after executing the *first* event in the WCTS, as illustrated by the dashed arc in the diagram.

As not every event is check-pointed, a process needs to redo intermediate events between the last saved state before the rollback time and the straggler that caused the rollback to reinstate the content of its current state, an execution phase called *coasting forward* [Fuj90]. Consider the example in Figure 60 where rollbacks are triggered by a straggler message with timestamp t_4 when the simulation executes in WCTS-t_5.

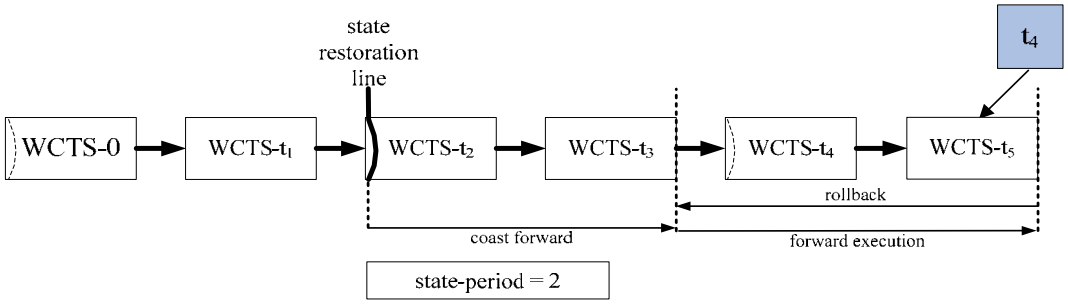


Figure 60. Rollbacks with the periodic state-saving strategy

If the CSS strategy is employed, the states of the processes would be restored to those saved at the end of WCTS-t_3, and from there the forward execution resumes right after the rollbacks. Now, the processes did not save their states in WCTS-t_3 under the PSS strategy. The most recent states before the current rollback time t_4 were saved at the beginning of WCTS-t_2. Therefore, the coasting forward operations are performed, reprocessing the events from the second one in WCTS-t_2 (the event immediate after the check-pointed event) up to the last one in WCTS-t_3. Fortunately, the overhead of coasting forward is less than normal event execution since any scheduling of new events is suppressed. The only purpose for the coasting forward phase is to reinstate the content of the current states of the processes at the end of WCTS-t_3, based on which the forward execution can be carried out.

The coasting forward operation is also controlled by the *state-period* variable in the kernel. If it is set to a nonnegative value in a state manager, the coasting forward operation will be performed for the process associated with that state manager during rollbacks. On the other hand, if it is -1, as in the case of the CSS strategy, the coasting forward phase is skipped.

A prominent challenge of integrating the PSS strategy into the PCD++ toolkit is handling the messaging anomalies as discussed in Chapter 6. If the NC saves its state infrequently, the strategic breakpoint states that should be saved after recovering from the anomalies may be lost. Moreover, it is hard, if not impossible, to regenerate the breakpoint state during the coasting forward phase in later rollbacks. This problem can be solved if the NC can still use the CSS strategy with the *StateManager* and avoid the coasting forward phase during rollbacks, while all

the other processors utilizing the PSS strategy with the *InfreqStateManager*. In this case, we have a hybrid strategy where both CSS and PSS strategies are employed simultaneously in the simulation system. Since there is only one NC on each machine (and potentially tens of thousands of Simulators), using this strategy does not wipe out much of the advantage associated with PSS. However, the kernel only allows one type of state managers, either the *StateManager* or the *InfreqStateManager*, to be used for all the processors in the system. Hence, a more flexible mechanism is needed to realize the hybrid strategy, as we will explain in the following sections.

7.2.2. UCSS mechanism revisited

In Section 6.4, we introduced the two-level UCSS mechanism that allows the application to decide when to save the state of a process on an event-by-event basis. This mechanism has been integrated with the CSS strategy, resulting in the risk-free MTSS strategy that can greatly reduce the number of states saved during the simulation. We now extend the UCSS mechanism so that processes can choose either type of state-saving strategy individually at runtime.

The *skip-state-saving* flag is still used with the highest priority in the extended UCSS mechanism. Additionally, we defined another flag with a lower priority, called as *do-state-saving*, in the *InfreqStateManager* associated with each process. By default, this flag has a value of false. The state-saving algorithm in the *InfreqStateManager* is modified so that, if the *do-state-saving* is set to true by a process, the *InfreqStateManager* saves states after every event, just like the *StateManager* does under CSS strategy. Therefore, the enhanced UCSS mechanism has a structure as illustrated in Figure 61.

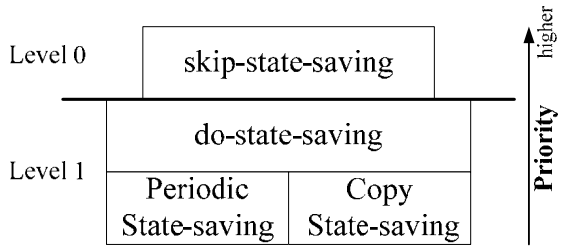


Figure 61. UCSS structure for hybrid state-saving strategy

Using the UCSS mechanism, a process can dynamically switch between the PSS and the CSS strategies by setting the *do-state-saving* flag at level 1. As the *skip-state-saving* flag has a higher priority at level 0, the MTSS strategy, previously works with the CSS strategy, takes effect under the PSS strategy as well. Thus, the enhanced UCSS mechanism virtually gives

simulator developers the full power to dynamically choose the best possible combination of state-saving strategies at runtime.

7.2.3. Integrating PSS strategy in PCD++

Thanks to the UCSS mechanism, integrating the PSS strategy into the PCD++ toolkit can be done with ease, as described below:

(1) The NC sets the *do-state-saving* flag to true in its associated *InfreqStateManager* when it is created. Hence, the NC can still utilize the CSS strategy during the simulation. On the other hand, all the other PCD++ processors (i.e. the FC, the Root, and the Simulators) use their *InfreqStateManager* to realize the PSS strategy.

(2) The NC also sets the *state-period* variable in its state manager to -1 to suppress the coasting forward operation during rollbacks. The value of *state-period* for the other PCD++ processors is specified by the user at compile time.

(3) The *skip-state-saving* flag is set when necessary by all the processors, including the NC, in their message-processing algorithms as described in Section 6.4. Hence, the MTSS strategy is integrated into the hybrid state-saving mechanism as well. In this case, a processor's current state is no longer saved after executing the first event in a WCTS. Instead, it is saved after processing the first *required* message (as demanded by the MTSS strategy), which is close to the end of the WCTS. This reduces the number of events that need to be reprocessed in the coasting forward phase, allowing better performance.

A rollback scenario under the hybrid state-saving strategy is depicted in Figure 62.

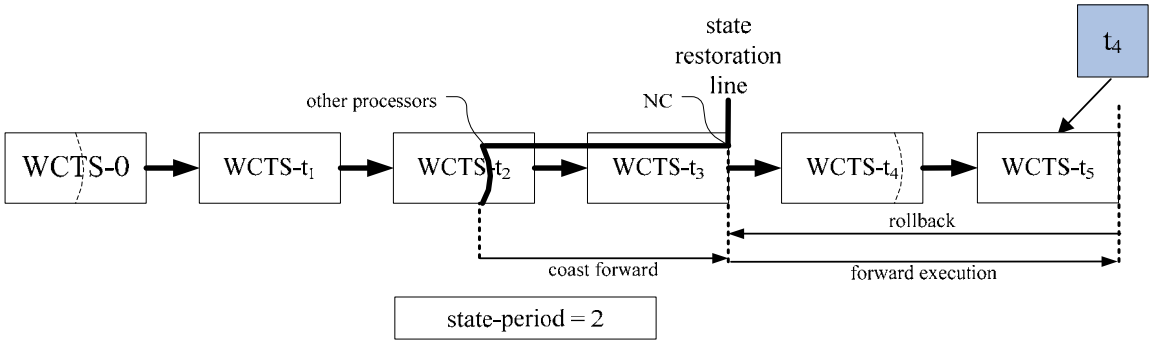


Figure 62. Rollbacks under the hybrid state-saving strategy

There are two major differences compared with the previous case as in Figure 60. First, the states of the processors are saved much closer to the end of the WCTS, shown by the dashed

arc in the corresponding wall clock time slices. This leads to shorter coasting forward phases. Secondly, since the NC still operates under the CSS strategy, its state is restored directly to the end of WCTS-t_3 during the rollback and no coasting forward is performed for the NC.

7.2.4. Modifications to the fossil collection algorithm

The fossil collection algorithm was discussed in Section 4.2.2. When the GVT moves forward, the GVT manager reclaims all but the last saved state older than the GVT along with the messages with timestamps less than the GVT in the input and output queues. This algorithm works well with the CSS strategy. Actually, the GVT value indicates the least timestamp of any potential straggler or anti-message that could be received by a process. In other words, it is the minimum rollback time for any process in the system. During rollbacks, the state of a process will be restored to the last one saved at virtual time earlier than the GVT. As we know, this last state is left in the state queue during fossil collections. Hence, the state restoration is successfully performed for the process, and the rollback completes as expected even in this extreme case where the rollback time is equal to the GVT.

However, when this fossil collection algorithm is used with the PSS strategy, runtime crash can happen during the coasting forward operations. Since the state of a process is saved infrequently, the restored state, i.e. the last one available in the state queue older than the GVT, could be saved at virtual time much earlier than the GVT. Although the state restoration can be done without problem in this case, the following coasting forward operations will fail since the events with timestamp between the time of the restored state and the GVT have already been garbage collected.

This scenario is illustrated in Figure 63, where the fossil collections have been done with a GVT value of 28. At this time, a straggler message with timestamp 32 arrives, forcing the process to be rolled back to virtual time 30. However, there is no state saved at time 30, and the last state available in the state queue was saved at time 15, a time well before the GVT. Hence, the state of the process is restored to this last state and the events with timestamp between 21 and 30 need to be reprocessed during the coasting forward. However, some of these events (e.g. events with timestamp 21 and 25) have already been reclaimed during the previous fossil collections, resulting in the failure of the coasting forward operation.

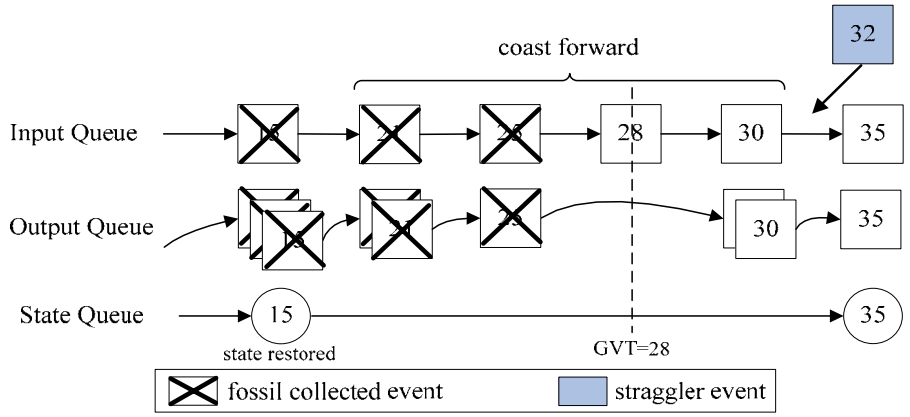


Figure 63. Example scenario for the failure of coasting forward operation

Hence, the fossil collection algorithm needs to be enhanced to deal with this problem. In the new algorithm, fossil collection is no longer performed using the computed GVT. Rather, a minimum value among the virtual time of the last states saved older than the GVT is calculated for all the processes mapped on a LP. Then, this minimum value is used to do the fossil collection. Figure 64 shows a typical scenario for fossil collections on two LPs under the new scheme.

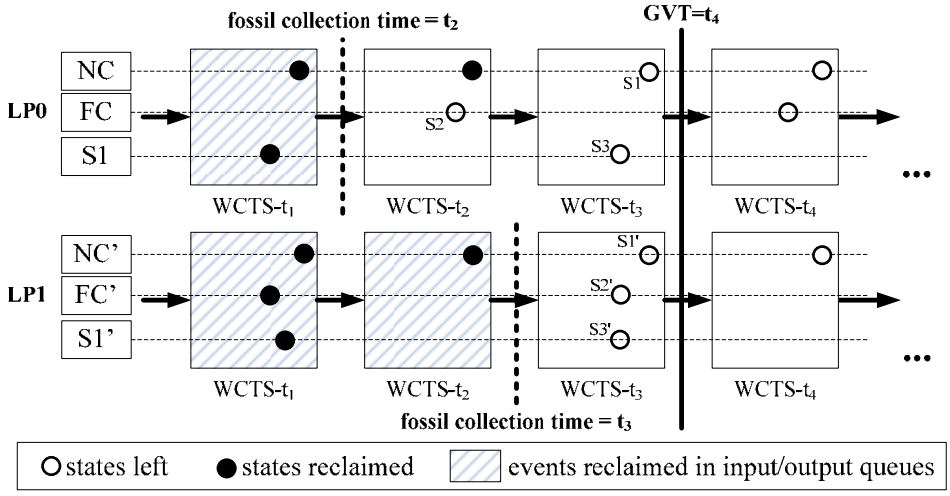


Figure 64. Example scenario for fossil collections under the new scheme

As shown in the diagram, after computing the new GVT value of t_4, the GVT manager on each LP performs fossil collection for the processes under its control. To do so, the GVT manager calculates the minimum virtual time among the last states saved before the current GVT. The resulting minimum virtual time among state S_1, S_2, and S_3 on LP0 is t_2, while that among state S_1', S_2', and S_3' on LP1 is t_3. Hence, virtual time t_2 and t_3 (instead of the GVT t_4) are

used respectively by the GVT managers on LP0 and LP1 to do fossil collections. The events with timestamp less than the corresponding minimum virtual time are removed from the input and output queues. As we can see, potentially different virtual time may be used by the LPs during fossil collections under the new scheme.

Eventually, the GVT is advanced to infinity when the simulation ends. Only at that time, does the GVT manager use the GVT value directly to do the last fossil collection. Thus, all states and events in the queues are reclaimed and data in the file queues are flushed, just like in the original scheme.

7.2.5. Miscellaneous modifications

Before the PSS strategy can be successfully integrated into the PCD++ toolkit, two more problems need to be fixed in the kernel, as summarized below:

(1) Determining the start point of the coasting forward operation. The original algorithm for the PSS strategy supposed that the state of a process is saved only after executing the *first* event in a WCTS. Hence, it rigidly chose the second event in that WCTS as the start point of the coasting forward operation during rollbacks. Under the MTSS strategy, however, a process now saves its state only after executing the first required message, which is certainly not the first and should be close to the end of the WCTS. Therefore, we need a more flexible way to determine the start point of the coasting forward. The resulting algorithm retrieves the *inPos* of the state restored during the rollback, and starts the coasting forward from the event immediately after the one pointed by the *inPos*, regardless of its actual position in the WCTS.

(2) Suppressing output to files. During the coasting forward, any scheduling of new events is suppressed by the kernel. However, the same was not done to prevent potential output to files. Duplicate data was inserted into the file queues as events were reprocessed. The original algorithm has been enhanced to withhold operations on the file queues in the coasting forward phase.

7.3. LAZY CANCELLATION

With the aggressive cancellation strategy, all the messages that have been optimistically processed ahead of the rollback time must be cancelled. However, it is possible that a true message may be sent prematurely for the wrong reason [Lin91]. Better performance could be achieved if the cancellation of the true message is suppressed during the rollback. The lazy cancellation strategy is a refinement that can be thought of as repairing incorrect computation, rather than discard it altogether as the aggressive cancellation strategy does [Low99]. To do so, the sending of anti-messages is deferred during rollbacks and the process resumes forward execution immediately after the state restoration and message un-processing. New output messages are compared with those speculatively generated before the rollback. If they are deemed as identical, called as *lazy hit*, no action is taken; otherwise, referred to as *lazy miss*, the deferred anti-message is sent out, followed by the new output message, to replace the original one at the destination.

Nevertheless, this strategy can also degrade performance since incorrect computation is not cancelled as prompt as is the case with aggressive cancellation. A key factor, so-called the *sensitivity of output message* [Lin91], in determining the performance of lazy cancellation is the probability that a given straggler will actually affect the results of the messages that were rolled back to accommodate the straggler, i.e. the probability of lazy miss. If this probability is low, then the lazy cancellation strategy can be expected to outperform the aggressive cancellation strategy.

The following provides a summary of our attempts to integrate the lazy cancellation strategy into the PCD++ toolkit.

(1) Choosing the appropriate implementation level. Previously, the strategy was realized at the local level in the WARPED kernel, where simulation objects perform lazy cancellation operations independent of each other. However, this approach will cause rollback failure when used in PCD++. As the PCD++ processors on a LP work cooperatively to implement the P-DEVS formalism and control messages are passed back and forth between them during the simulation, a processor cannot resume forward execution regardless of the others. As discussed in Section 6.1, all the processors on a LP must be rolled back collectively to the end of a previous WCTS to ensure that the forward execution can be successfully resumed after the

rollbacks. Therefore, we modified the kernel algorithm to implement the lazy cancellation strategy at the partition level instead. In this case, message cancellations between the LPs are performed using the lazy cancellation strategy, while those between the local processors (or simulation objects) within a LP still adhere to the aggressive cancellation strategy.

(2) Implementing the function for comparing external messages. The kernel invokes a function called *lazyCmp* to determine whether a lazy hit or miss occurs during lazy cancellation. Since the lazy cancellation strategy is now realized at the partition level and only external messages are exchanged between the LPs, we only need to define the *lazyCmp* function for comparing external messages at the PCD++ layer. Two external messages are considered as identical if they have the same send time, receive time, sender, receiver, sign, destination port, and value.

CHAPTER 8 EXPERIMENTS AND PERFORMANCE ANALYSIS

In this chapter, we study the performance of the PCD++ toolkit for Cell-DEVS models quantitatively. Our experiments were carried out on a HP PROLIANT DL Server, a cluster of 32 compute nodes (dual 3.2GHz Intel Xeon processors, 1GB PC2100 266MHz DDR RAM) running Linux WS 2.4.21 interconnected through Gigabit Ethernet and communicating over MPICH 1.2.6. A brief introduction to the Cell-DEVS models tested in our experiments is provided in Section 8.1. The performance metrics are presented in Section 8.2. The improvements achieved by using the one log file per node strategy and the message type-based state-saving (MTSS) strategy are covered in Section 8.3 and 8.4 respectively. The execution results of the Cell-DEVS models using the standard Time Warp algorithms are presented in Section 8.5, while the effects of different Time Warp optimizations are discussed in Section 8.6.

8.1. INTRODUCTION TO THE CELL-DEVS MODELS

The performance of the PCD++ simulator was tested with two Cell-DEVS models, including a model for fire propagation in forest based on Rothermel's mathematical definition [Rot72] and a 3-D watershed model representing a hydrology system originally presented in [Moo96] and enhanced in [Ame01]. Since these models have already been validated in the previous researches, we focus on the model verification in our experiments to ensure that the PCD++ simulator executes the models correctly.

The correctness of the simulation is verified using the methods as summarized below:

(1) Verification with debugging files. During the simulation, a debugging file is created on each node to log the detailed information of the messages executed on that node. These debugging files are carefully analyzed after the simulation to ensure that the variables defined in the models and the PCD++ processors are manipulated correctly according to the message-processing algorithms as defined in Section 5.5.

(2) Verification of the simulation results. A distributed simulation is correct when it produces simulation results that are legal results from a traditional, single process simulator [Fre02]. In our experiments, the models are first executed on a single

node using the standalone version of the CD++ toolkit, and the generated simulation results are used as reference outputs for verification purposes. These models are then executed with PCD++ on multiple nodes. After each run, the simulation results are compared with the reference outputs to ensure that the same results are generated with PCD++ as those produced with the standalone version.

Also notice that the performance data presented in the following sections only reflects the evaluation results for the specific models tested in our experiments. A thorough analysis of the performance of the PCD++ simulator for models with different characteristics needs to use some benchmarks such as the DEVStone [Gli04], which will be addressed in the future work.

The CD++ definitions of the fire propagation and the watershed models are given next, while more details on the models themselves can be found in [Ame01].

The fire propagation model computes the ratio of spread and intensity of fire in forest based on specific environmental and vegetation conditions. Figure 65 shows the definition of the model in CD++ using environmental values obtained for a fuel model group number 9, a SE wind of 24.135 km/h and a cell size of 15.24×15.24 m.

```
[top]
components : forestfire

[forestfire]
type : cell              dim : (30,30)          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 (1,-1)!=? and 0<(1,-1)}
rule : {(1,0)+(15.24/5.106976)} {(15.24/5.106976)*60000} {(0,0)=0 and (1,0)!=? and 0<(1,0)}
rule : {(0,-1)+(15.24/5.106976)} {(15.24/5.106976)*60000} {(0,0)=0 and (0,-1)!=? and 0<(0,-1)}
rule : {(-1,-1)+(21.552615/1.872060)} {(21.552615/1.872060)*60000} {(0,0)=0 and (-1,-1)!=? and 0<(-1,-1)}
rule : {(1,1)+(21.552615/1.872060)} {(21.552615/1.872060)*60000} {(0,0)=0 and (1,1)!=? and 0<(1,1)}
rule : {(-1,0)+(15.24/1.146091)} {(15.24/1.146091)*60000} {(0,0)=0 and (-1,0)!=? and 0<(-1,0)}
rule : {(0,1)+(15.24/1.146091)} {(15.24/1.146091)*60000} {(0,0)=0 and (0,1)!=? and 0<(0,1)}
rule : {(-1,1)+(21.552615/0.987474)} {(21.552615/0.987474)*60000} {(0,0)=0 and (-1,1)!=? and 0<(-1,1)}
rule : {(0,0)} 0 { t }
```

Figure 65. Definition of the fire propagation model in CD++

The watershed model represents the water flow and accumulations depending on the characteristics of different vertical layers: air, vegetation, surface waters, soil, ground water, and bedrock [Moo96]. Based on the mathematical equations, it was coded as a 3-D Cell-DEVS model in CD++ to simulate the accumulation of water under the presence of constant rain (7.62 mm/hr) [Ame01]. Figure 66 shows the model definition with a 20×20×2 cell space in CD++.

```
[top]
components : watershed

[watershed]
type : cell                dim : (20,20,2)          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)..(19,6,0)}                    stones { (0,13,0)..(19,19,0) }
localtransition : hydrology
[grass]
rule : {0.07 + (0,0,0) - if(((-1,0,0) != ?) and (((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(((1,0,0) != ?) and (((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,-
1,0) != ?) and (((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(((0,1,0) != ?) and
(((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(((-1,0,0) != ?) and (((-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(((1,0,0) != ?) and (((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,-1,0) != ?) and (((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(((0,1,0) != ?) and (((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 }

[stones]
rule : {0.09 + (0,0,0) - if(((-1,0,0) != ?) and (((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(((1,0,0) != ?) and (((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,-
1,0) != ?) and (((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(((0,1,0) != ?) and
(((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(((-1,0,0) != ?) and (((-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(((1,0,0) != ?) and (((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,-1,0) != ?) and (((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(((0,1,0) != ?) and (((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 }

[hydrology]
rule : {0.12 + (0,0,0) - if(((-1,0,0) != ?) and (((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(((1,0,0) != ?) and (((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,-
1,0) != ?) and (((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(((0,1,0) != ?) and
(((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(((-1,0,0) != ?) and (((-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(((1,0,0) != ?) and (((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,-1,0) != ?) and (((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(((0,1,0) != ?) and (((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 }
```

Figure 66. Definition of the watershed model in CD++

8.2. PERFORMANCE METRICS

A set of 21 key values was collected during the experiments to gauge the performance and profile the simulation system. These values fall into two categories based on their intended purposes, namely *performance measurement* and *system profiling*.

The first group consists of 3 values collected from the execution environment to measure the performance of the simulator in terms of execution time, memory consumption, and CPU utilization, as shown in Table 1. The *overall speedup* for N nodes is defined as follows.

$$\text{Overall Speedup} = \frac{T(1)}{T(N)} \qquad (1)$$

Where T(N) represents the total execution time taken by the simulation running on N nodes, and T(1) stands for the *best* possible serial execution time measured on one node.

Table 1. Metrics for performance measurement

Category	Metric Name	Description
Performance Measurement	Execution Time (T)	Total execution time of the simulation
	Memory Usage (MEM)	Average and maximum amount of memory consumed by the simulation (Kb)
	CPU Usage (%CPU)	Share of the elapsed CPU time expressed as a percentage

The second group has 18 values acquired by the PCD++ toolkit itself at runtime to profile the simulation system, as shown in Table 2. They are generated at the partition level by the LP on each compute node. Combining the data collected on all the nodes, we can have a general picture about the execution of the whole simulation system.

Table 2. Metrics for system profiling

Category	Metric Name	Description
System Profiling	Events Received (ER)	Number of events inserted into the input queue
	Events Imploded (EI)	Number of events annihilated in the input queue[1]
	Events Executed (EE)	Number of events executed
	Event-executing Time (ET)	Time spent on executing events
	States Saved (SS)	Number of states saved during the simulation
	States Skipped (SK)	Number of states skipped by the MTSS strategy
	States Reduced (SR)	Number of states reduced by the PSS strategy
	State-saving Time (ST)	Time spent on saving states
	Primary Rollbacks (PR)	Number of primary rollbacks
	Secondary Rollbacks (SR)	Number of secondary rollbacks
	Rollbacks (RB)	Total number of rollbacks
	Rollback Length (RBL)	Length of rollbacks expressed as the number of events unprocessed
	Rollback Time (RBT)	Time spent on rollbacks
	Coast Forward Length (CFL)	Length of coasting forward phases expressed as the number events reprocessed
	Coast Forward Time (CFT)	Time spent on coasting forward operations
	Lazy Hit (LH)	Lazy hits under lazy cancellation strategy
	Lazy Miss (LM)	Lazy misses under lazy cancellation strategy
	Bootstrap Time (BT)	Time spent on bootstrap operations
	Running Time (RT)	Time spent on simulation after bootstrap (T - BT)

[1] When the one anti-message per rollback strategy is used, EI is the number of anti-messages received on each node. The actual number of message implosions should be greater than the EI value since one anti-message is capable of annihilating multiple positive messages in this case.

In the standalone and conservative versions, we can tally the number of messages executed in the simulation by analyzing the generated log files. However, the numbers of events received (ER), imploded (EI) and executed (EE) in optimistic simulations need to be collected at runtime within the kernel. The log files only contain those true messages that have survived throughout the simulation. The average time required for processing an event is the ratio of the total time spent on event execution (ET) to the number of events executed in the simulation (EE). Metrics related to state-saving operations include the number of states saved in the state queues (SS), the number of states skipped by the MTSS (SK) and PSS (SR) strategies respectively, and the time spent on saving states (ST), by which we can have a better insight into the state-saving process. Also, the average time required for saving a state can be calculated by ST/SS. The total number of rollbacks (RB) is the sum of the primary (PR) and secondary (SR) rollbacks happened in the system (i.e. RB = PR + SR). The total length of the rollbacks (RBL) and the time for performing the rollback operations (RBT) are collected during the simulation, based on which we can obtain both the average length (RBL/RB) and the average processing time (RBT/RB) of a single rollback. The overhead of coasting forward is profiled by the length (CFL) and dedicated time (CFT) of the operations, while the effect of the lazy cancellation strategy is measured by the number of lazy hit (LH) and miss (LM). The bootstrap time (BT) serves two purposes: first, we can conveniently use it to measure the performance improvement resulting from the one log file per node strategy, as we will discuss in the next section; secondly, more accurate results about the real gain derived from the parallel algorithms can be obtained if we subtract the BT from the total execution time (T). Hence, we use two different speedups in our analysis. An *overall speedup* based on the total execution time (including the BT) reflects how much faster the simulation runs on multiple machines than it does on a single one as felt by the users. The definition of the overall speedup is given by Equation (1). Moreover, an *algorithm speedup* (without considering the BT) is used to assess the performance gain attributed to the parallel algorithms alone, as defined below.

$$\text{Algorithm Speedup} = \frac{T(1) - BT(1)}{T(N) - \frac{1}{N}\sum_{i=1}^{N} BT(i)} = \frac{RT(1)}{RT(N)} \qquad (2)$$

Where BT(1) and RT(1) represent the bootstrap and running time collected on a single node respectively, BT(i) stands for the bootstrap time recorded on node i ($1 \le i \le N$) when the simulation is executed on N nodes, and RT(N) is the average running time on N nodes. As in the

overall speedup definition, T(N) represents the total execution time taken by the simulation running on N nodes, and T(1) stands for the *best* possible serial execution time measured on one node.

8.3. EFFECT OF ONE LOG FILE PER NODE

In this section, we discuss the performance improvement derived from the one log file per node strategy using a fire propagation model of 900 cells arranged in a 30×30 mesh. The standard Time Warp algorithms, i.e. LTSF scheduling, copy state-saving, pGVT algorithm, and aggressive cancellation, were used in the experiments. Also, the MTSS strategy was turned on in all runs. The simulator was configured to log all the messages exchanged during the simulation.

The model was executed on a single node and 4 nodes (respectively) with and without using the one log file per node strategy (respectively) to simulate the behavior of forest fire during a period of 5 hours. The resulting execution time (T) and bootstrap time (BT) in these four cases are illustrated in Figure 67, where the BT for 4 nodes is the arithmetic average of the BT values collected on these nodes.

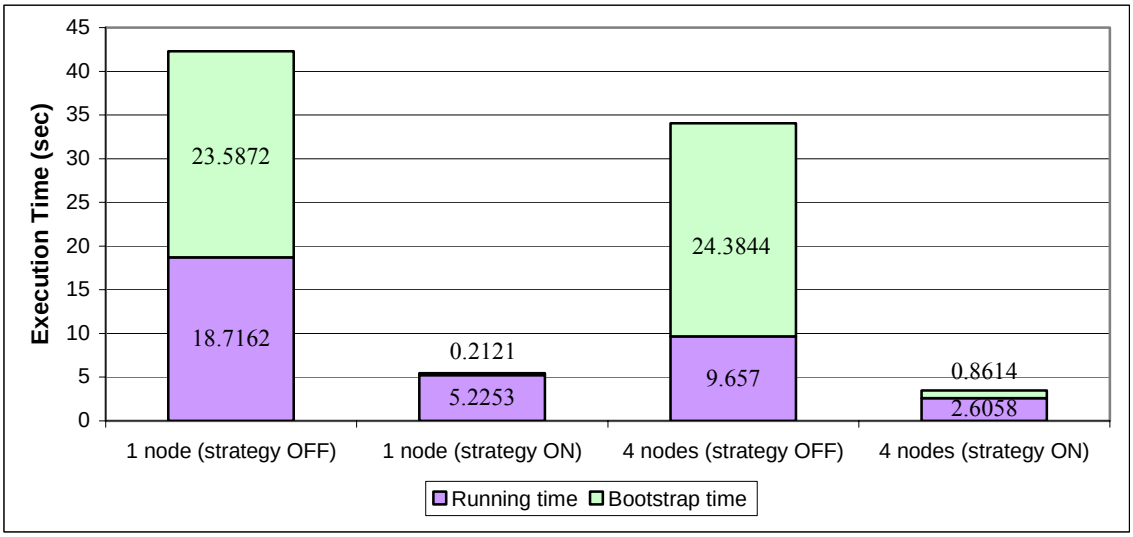


Figure 67. Execution and bootstrap time before and after one log file per node strategy on 1 and 4 nodes

Notice that the bootstrap time is even greater than the actual running time when the strategy is turned off. This clearly indicates that the bootstrap operation is really a bottleneck during the simulation. When the strategy is turned on, the bootstrap time is reduced significantly. As we can see, it is reduced by 99.1% on a single node and by 96.47% on 4 nodes. Furthermore,

the running time is decreased considerably as well due to more efficient communication, I/O, and rollback operations associated with the one log file per node strategy, as discussed in Section 6.6. It is reduced by 72.08% on 1 node and by 73.02% on 4 nodes.

The CPU usage (%CPU) monitored in our experiments also suggests that the file I/O operation is a major barrier in the bootstrap phase. The CPU usage collected before and after applying the one log file per node strategy on a single node is shown in Figure 68.

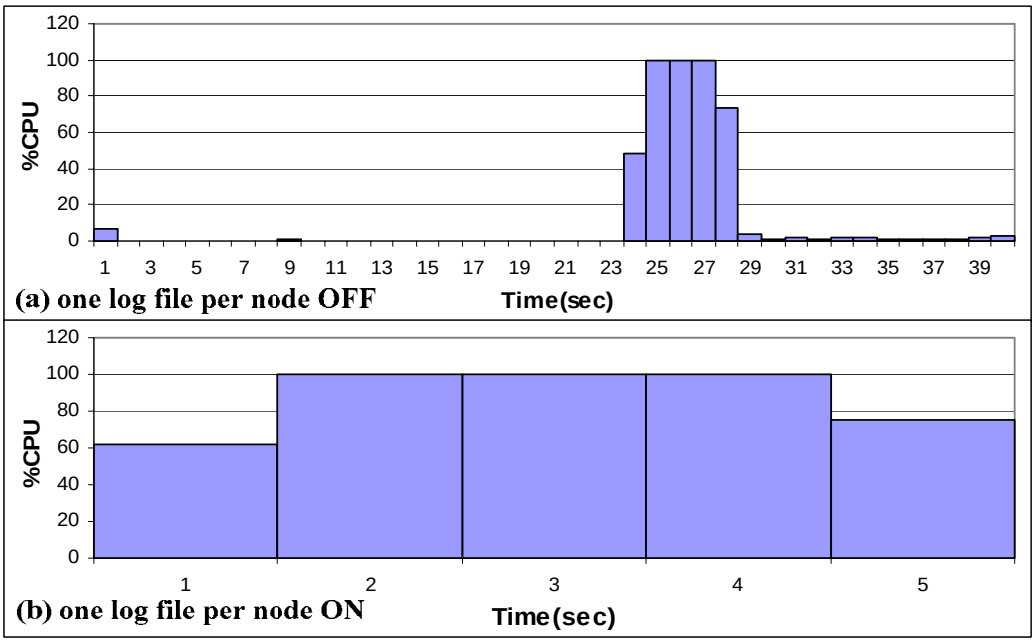


Figure 68. CPU usage before and after one log file per node strategy on 1 node

Shown in Figure 68(a), when the strategy is turned off, the CPU essentially remains idle in the first 23 seconds (corresponding to the observed BT), during which a majority of time has been dedicated to I/O operations for creating the log files at the NFS server over the network. At the end of the simulation, the logged data is flushed to the physical files, resulting in intensive I/O operations again. As expected, the CPU rests idle in the last 12 or so seconds. On the other hand, the computation is condensed when the strategy is applied to the simulator, as shown in Figure 68(b). Hence, the CPU is utilized much more efficiently with the one log file per node strategy. The similar pattern was observed in simulations running on multiple nodes.

In addition, several other observations can be obtained in the experiments as follows:

(1) The bootstrap time tends to increase when more nodes are used to do the simulation. For example, the BT increased from 0.2121 seconds on 1 node to 0.8614 seconds on 4 nodes in our experiments. The reason is that the number of log

files increases with the number of nodes, causing higher delays in communication and file I/O operations at the NFS server.

(2) The bootstrap time also tends to increase somewhat along with the size of the model because of the additional operations for memory allocation and object initialization in the main memory. However, this is a relatively moderate increase when compared with the previous case.

(3) Even though the bootstrap time is reduced significantly with the one log file per node strategy, it still constitutes an overhead that cannot be ignored when we measure the real effect of the parallel algorithms. In the experiments, it accounts for 3.9% and even 24.84% of the total execution time on 1 and 4 nodes respectively. This is why we need to use both the overall and algorithm speedups in our analysis of the performance.

8.4. EFFECT OF MESSAGE TYPE-BASED STATE SAVING

The MTSS strategy has been introduced in Section 6.4. We now demonstrate its associated performance improvement when used with the CSS strategy. The same fire propagation model was used for this purpose. Besides the standard Time Warp algorithms, the one log file per node strategy is also applied to the simulator in the following experiments.

The model was executed on 1 and 4 nodes (respectively) with and without the MTSS strategy (respectively). The number of states saved in the simulation (SS) and the time spent on state-saving operations (ST) are shown in Figure 69. Here, the data for 4 nodes is the average of the corresponding values collected on the nodes.

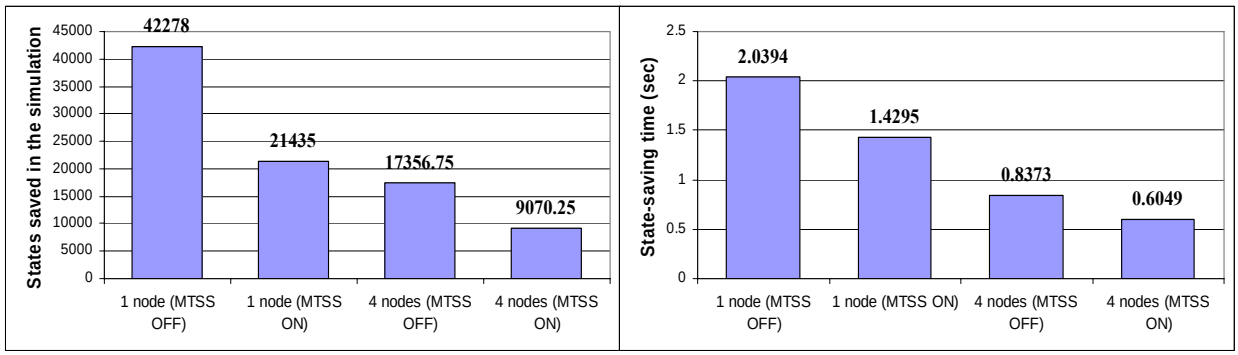


Figure 69. States saved and state-saving time before and after MTSS strategy on 1 and 4 nodes

Owing to the MTSS strategy, the number of states saved during the simulation is reduced by 49.29% and 47.74% on 1 and 4 nodes respectively. Accordingly, the time spent on state-saving operations is decreased by 29.9% and 27.76%.

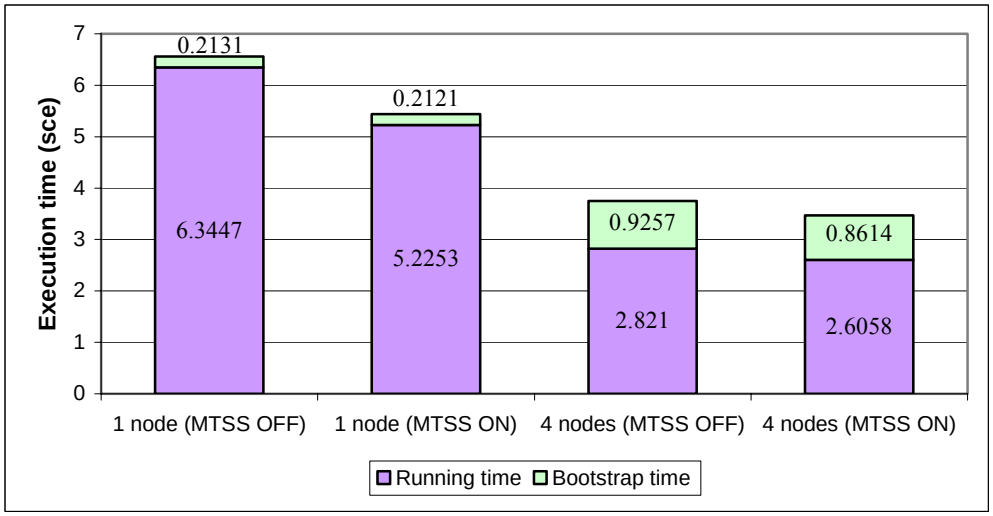


Figure 70. Running and bootstrap time before and after MTSS strategy on 1 and 4 nodes

The resultant running and bootstrap time are shown in Figure 70, where the BT for 4 nodes is the average of the corresponding values collected on the nodes. While the bootstrap time remains nearly unchanged in both cases, the actual running time is reduced by 17.64% and 7.63% on 1 and 4 nodes respectively because fewer states are saved in the state queues and, potentially, removed from the queues during rollbacks.

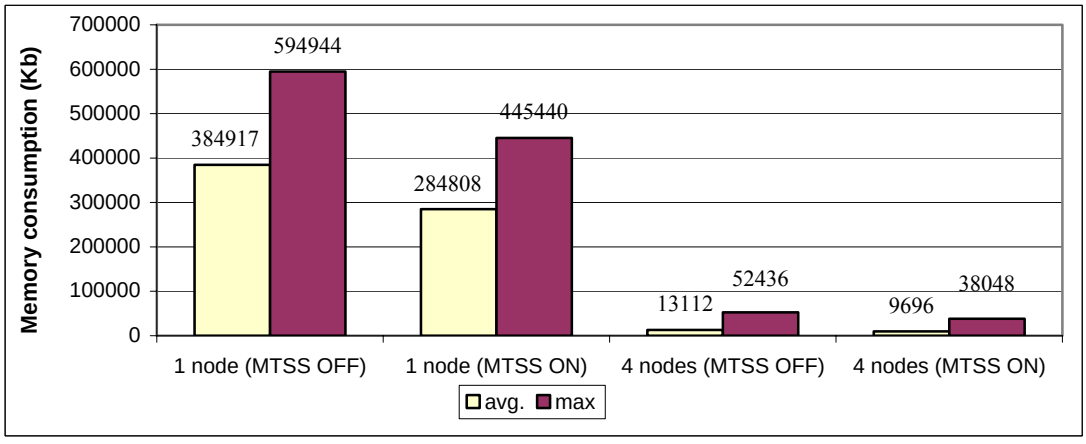


Figure 71. Average and maximum memory consumption before and after MTSS strategy

Probably the most noticeable effect of the MTSS strategy is the decrease in memory consumption. Figure 71 shows the time-weighted average and maximum memory consumption with and without the strategy for the fire propagation model on 1 and 4 nodes. The time-

weighted average was calculated using an interval of 1 second. For 4 nodes, the data was also averaged over the nodes. The average memory consumption declines by 26% in both cases, while the peak memory consumption decreases by 25.13% and 27.44% on 1 and 4 nodes respectively.

8.5. EXPERIMENTS WITH STANDARD TIME WARP PROTOCOL

Performance is of paramount important in parallel and distributed simulations. The key metrics for evaluating the performance of the PCD++ simulator are the execution time and speedup. In this section, we analyze the execution results of the Cell-DEVS models with the standard Time Warp algorithms. The one log file per node and MTSS strategies were applied to the simulator in the experiments as well.

A simple partition strategy was adopted for all the models in the following tests. It evenly divides the cell space into horizontal rectangles, as illustrated in Figure 72 for a 30×30 model partitioned over 3 nodes. Using different partition strategies could have a big impact on the performance of the simulation. Since the workload on the nodes is unpredictable and keeps changing during the simulation, it is hard, if not impossible, to predict the best partition strategy for a given model before the simulation. This problem can be alleviated by using some dynamic load-balancing techniques in the simulation algorithms, which is out of the scope of this work.

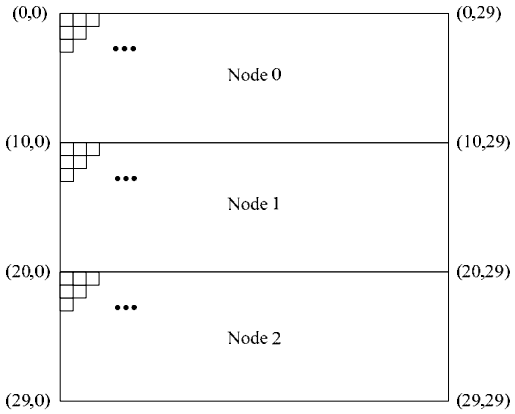


Figure 72. A simple partition strategy for Cell-DEVS models

- **Fire Propagation**

The fire propagation model is used again to assess the performance of the PCD++ simulator. We tested this model using different sizes of cell spaces: 20×20 (400 cells), 25×25

(625 cells), 30×30 (900 cells) and 35×35 (1225 cells). The model was executed to simulate the fire behavior over a period of 5 hours.

Figure 73 shows a comparison between our optimistic simulator and the previous conservative simulator [Tro01, Tro03] for different model sizes on a set of compute nodes. In all cases, the optimistic simulator markedly outperforms the conservative one.

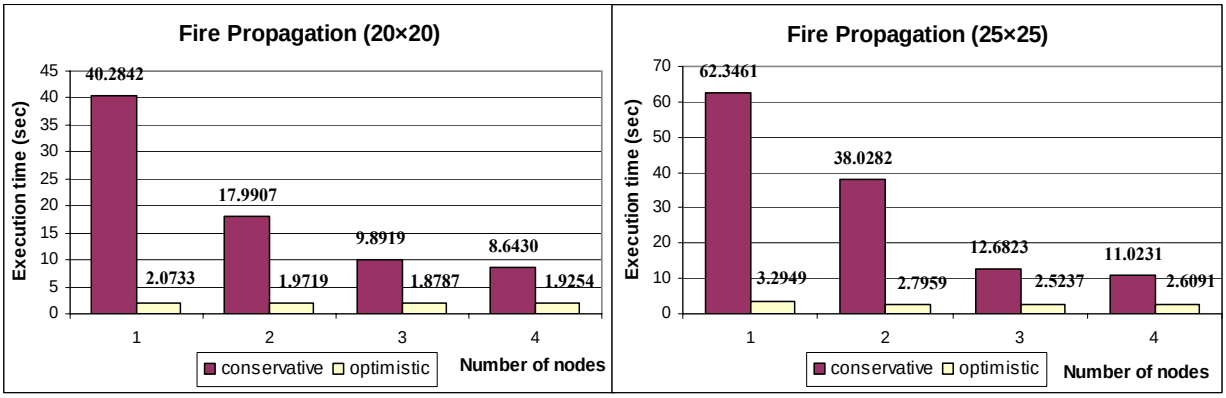


Figure 73. Comparison between optimistic and conservative simulators using the fire model

There are three major contributing factors:

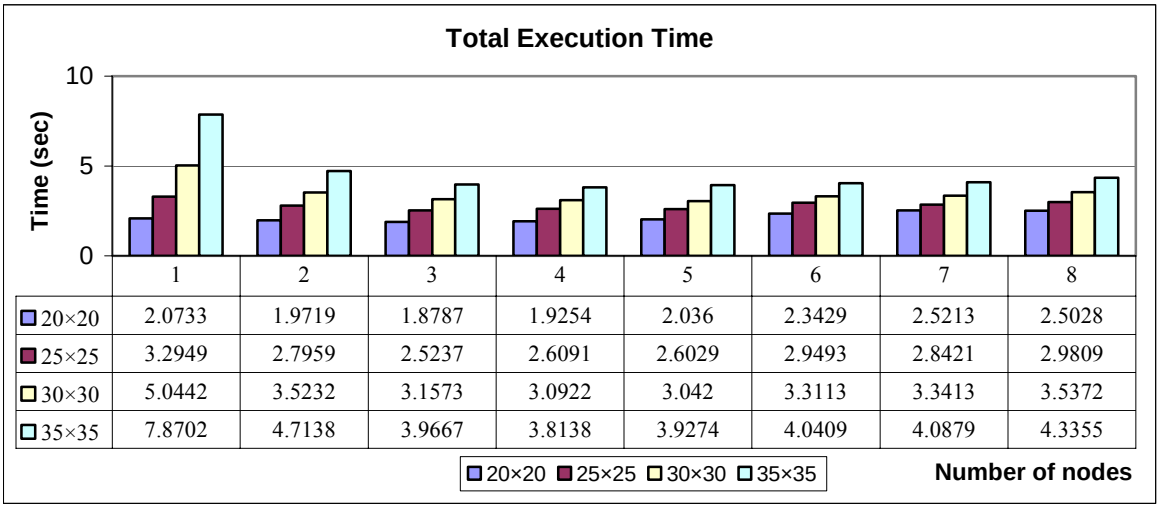
(1) The PCD++ toolkit has been optimized with the one log file per node strategy. Hence, its bootstrap time is substantially less than that of the conservative one. Although the data logged during the simulations is the same for both simulators, the number of log files generated by PCD++ is only a small fraction of that created by the conservative simulator. This factor accounts for much of the difference in the execution time on a single node.

(2) The Time Warp optimistic algorithms avoid, for the most part, the serialization of execution that is inherent in the conservative algorithms, and hence exploit higher degree of concurrency in the application.

(3) The non-hierarchical approach adopted in the PCD++ toolkit outperforms the hierarchical one of the conservative simulator. The flattened structure reduces the communication overhead and allows more efficient message exchanges between the PCD++ processors.

Figure 74 shows the total execution time of the fire model with different sizes executed on 1 up to 8 nodes. For any given number of nodes, the execution time always increases as the size of the model goes up. Moreover, the execution time rises less steeply when more nodes are used to do the simulation. For example, as the model size increases from 400 to 1225 cells, the

execution time ascends sharply by nearly 280% (from 2.0733 to 7.8702 seconds) on 1 node, whereas it merely rises by 93% (from 2.036 to 3.9274 seconds) on 5 nodes.



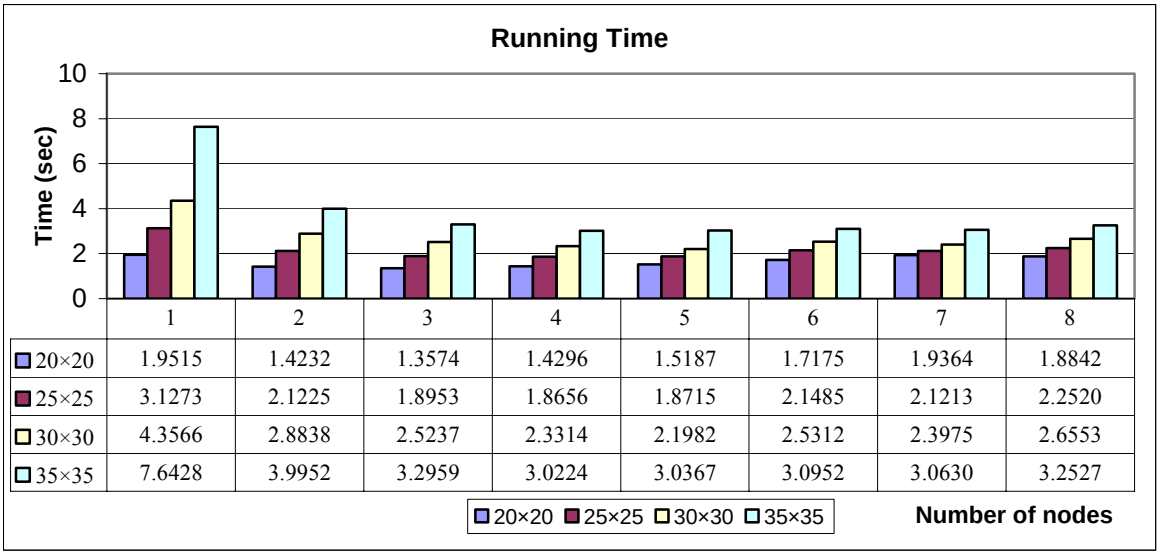
	1	2	3	4	5	6	7	8
20×20	2.0733	1.9719	1.8787	1.9254	2.036	2.3429	2.5213	2.5028
25×25	3.2949	2.7959	2.5237	2.6091	2.6029	2.9493	2.8421	2.9809
30×30	5.0442	3.5232	3.1573	3.0922	3.042	3.3113	3.3413	3.5372
35×35	7.8702	4.7138	3.9667	3.8138	3.9274	4.0409	4.0879	4.3355

Figure 74. Total execution time for fire model of various sizes on a set of nodes

On the other hand, for a fixed model size, the execution time tends to, but not always, decrease when more nodes are utilized. For instance, the execution time for the 20×20 model decreases from 2.0733 to 1.8787 seconds when the number of nodes climbs from 1 to 3. However, when the number of nodes increases further, the downward trend of execution time is reversed. It increases from 1.9254 to 2.5028 seconds as the number of nodes rises from 4 to 8. Actually, when the number of nodes goes beyond 5, the execution time is even larger than that recorded on a single node. The similar pattern can be discerned in the diagram for all the different model sizes tested in the experiment. When the model is partitioned onto more and more nodes, the increasing overhead involved in inter-LP communication and potential rollbacks eventually degrades the performance of the simulation system. Therefore, choosing the appropriate number of nodes to execute a given model is actually an art of balance. A trade-off between the benefits of higher degree of parallelism and the concomitant overhead costs needs to be reached when we consider different partition strategies, which could be one of the most difficult decisions for the modelers.

From Figure 74, we can also find that the best performance can be achieved on a larger number of nodes as the model size increases. The shortest execution time is achieved on 3 nodes for the 20×20 and 25×25 models, while it is obtained on 4 or 5 nodes for the other two larger models. It is clear that we should use more nodes to simulate larger and more complex models where intensive computation is the dominant factor in determining the system performance.

The bootstrap time collected during the simulations is subtracted from the total execution time to measure the performance gain attributed to the Time Warp optimistic algorithms. Due to the one log file per node strategy, the recorded bootstrap time varies from 0.1218 to 1.0249 seconds depending on the model size and the number of nodes involved in the simulation. The resulting running time is shown in Figure 75.



	1	2	3	4	5	6	7	8
20×20	1.9515	1.4232	1.3574	1.4296	1.5187	1.7175	1.9364	1.8842
25×25	3.1273	2.1225	1.8953	1.8656	1.8715	2.1485	2.1213	2.2520
30×30	4.3566	2.8838	2.5237	2.3314	2.1982	2.5312	2.3975	2.6553
35×35	7.6428	3.9952	3.2959	3.0224	3.0367	3.0952	3.0630	3.2527

Figure 75. Running time for fire model of various sizes on a set of nodes

Based on the above execution and running time, we can calculate the overall and algorithm speedups using Equation (1) and (2) respectively, as shown in Figure 76.

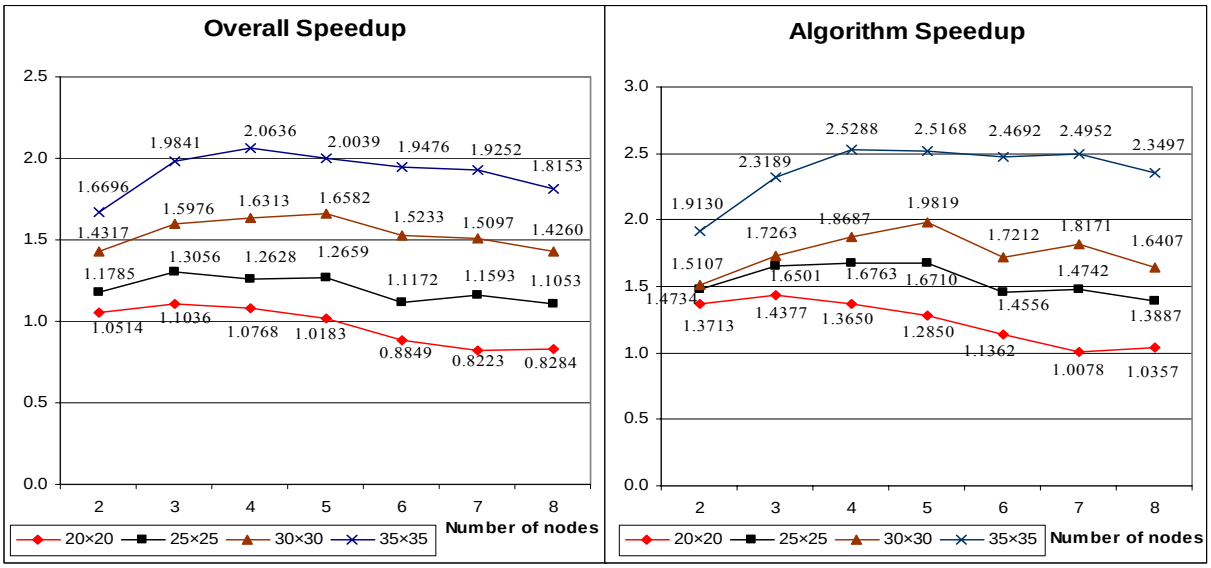


Figure 76. Overall and algorithm speedups for fire model of various sizes on a set of nodes

As we can see, higher speedup can be obtained with larger models. For any given model size, a peak value exists in the speedup curve, indicating the best performance achieved for that model. For example, the highest overall and algorithm speedups for the 35×35 model are obtained on 4 nodes with values of 2.0636 and 2.5288 respectively. The algorithm speedup is always higher than its counterpart overall speedup, an evidence showing that the Time Warp optimistic algorithms are major contributors to the performance improvement.

- **A Watershed Model**

The watershed model was tested in our experiments to evaluate the performance of PCD++ for simulating models of complex physical system. Due to its complex rule definitions, this model requires high computing power to carry out the simulation. Also, it uses a neighborhood consisting of 10 cells at both layers of the cell space, which allows us to investigate how well our simulator performs when the interaction between neighboring cells is frequent. We executed the model using two different cell spaces: 15×15×2 (450 cells) and 20×20×2 (800 cells).

Figure 77 shows the total execution time and running time collected on a set of nodes for the 15×15×2 model. The execution time adheres to the same pattern as discussed previously, where the best performance is achieved on 5 nodes with execution and running time of 6.1538 and 5.6743 seconds respectively. In all cases, the bootstrap time is well below 1 second.

	1	2	3	4	5	6	7	8
Total execution time	16.8036	11.793	8.3285	7.3205	6.1538	7.5166	7.4827	6.4963
Running time	16.6668	11.1522	7.7191	6.8140	5.6743	6.8870	6.8219	5.8806

Figure 77. Total execution and running time for the 15×15×2 watershed model

The resulting speedups are illustrated in Figure 78. The best overall and algorithm speedups achieved for the 15×15×2 model are 2.7306 and 2.9373 respectively, higher than those obtained with the fire models.

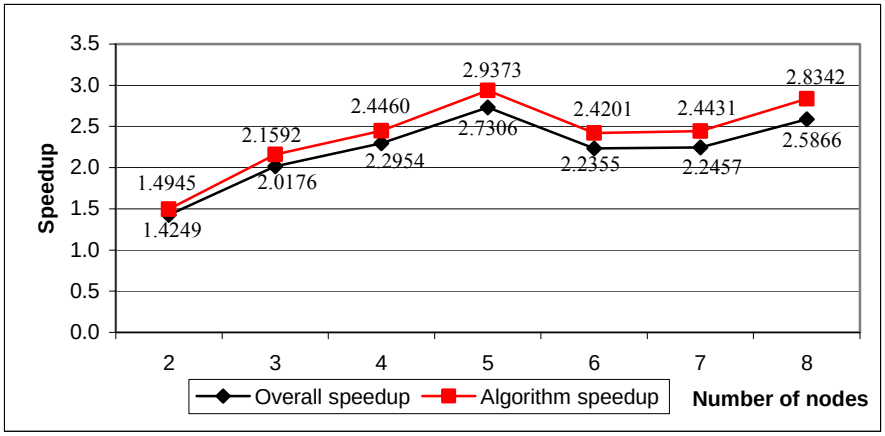


Figure 78. Overall and algorithm speedups for the 15×15×2 watershed model

The experimental results for the 20×20×2 model are shown in Figure 79. When the model was executed on a single node, the memory usage climbed almost to the limit, resulting in high memory swapping and a very long execution time of 586.505 seconds (the conservative simulator generated an even longer execution time in our experiments – 1030.153 seconds on 1 node and 166.1047 on 2 nodes, which are not shown in the diagram). As we can see, the execution time decreases sharply to only 18.3177 seconds on 2 nodes. Besides the performance gain from the parallel algorithms, one major reason is that the memory consumption falls significantly on each node when the model is partitioned into multiple parts. The shortest execution time is obtained on 6 nodes.

Figure 79. Total execution and running time for the 20×20×2 watershed model

Since the serial execution time measured on one node is large, the calculated speedups are exceptionally high, as shown in Figure 80. However, we know that these speedups are exaggerated by the high memory swapping happened on a single node. This shows that many

other factors need to be considered in the experiments to evaluate the performance accurately. Memory consumption, communication network, and NFS server are some prominent examples.

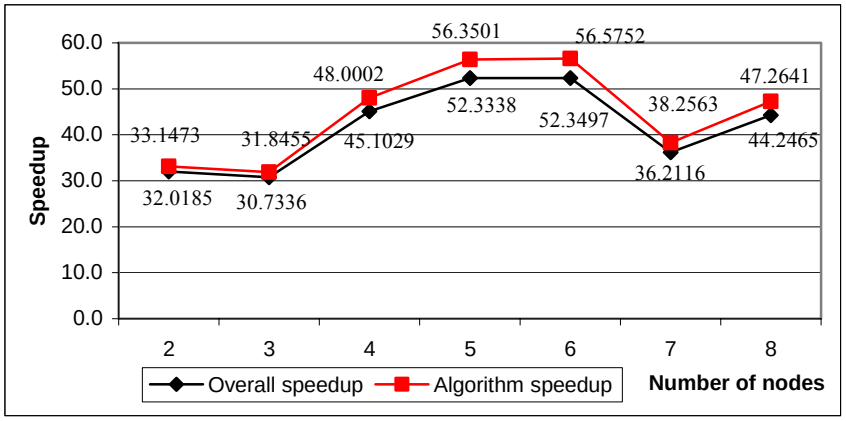


Figure 80. Overall and algorithm speedups for the 20×20×2 watershed model (false)

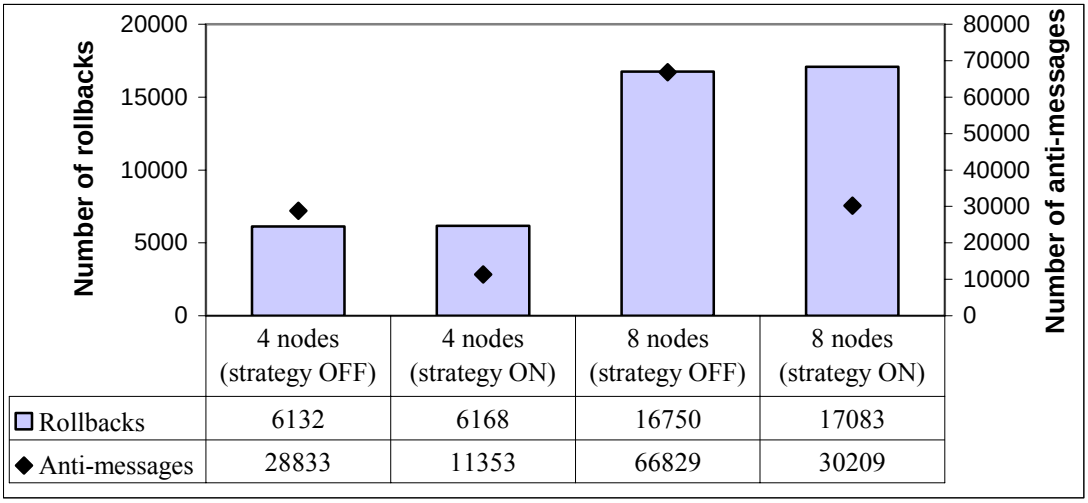
8.6. TIME WARP OPTIMIZATIONS

In this section, we study the potential performance gain of different Time Warp optimization algorithms that have been integrated into the PCD++ toolkit, including the one anti-message per rollback strategy, the periodic state saving strategy and the lazy cancellation strategy. The fire propagation model was used in our experiments for testing purposes. Each scenario was tested for 10 runs and the average values of these runs were calculated.

- **One anti-message per rollback**

We executed the 35×35 model on 1, 4, and 8 nodes with and without the one anti-message per rollback strategy respectively. Figure 81 shows the average number of rollbacks and anti-messages on 4 and 8 nodes.

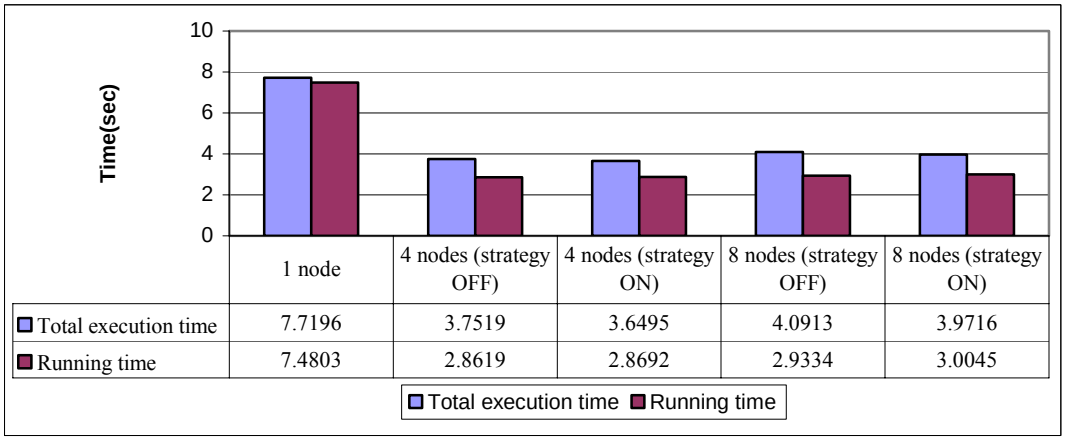
On 4 nodes, the number of rollbacks remains almost unchanged before and after applying the strategy. However, the number of anti-messages generated during the rollbacks declines sharply by 60.62% from 28,833 to 11,353. Similar results can be found on 8 nodes as well. In this case, the number of rollbacks actually increases by 2% after applying the strategy, whereas the number of anti-messages reduces by 54.8% from 66,829 to 30,209. As expected, the one anti-message per rollback strategy does greatly reduce the number of anti-messages during rollbacks.



	4 nodes (strategy OFF)	4 nodes (strategy ON)	8 nodes (strategy OFF)	8 nodes (strategy ON)
Rollbacks	6132	6168	16750	17083
Anti-messages	28833	11353	66829	30209

Figure 81. Number of rollbacks and anti-messages for the 35×35 fire model

The total execution and running time for these cases is shown in Figure 82.



	1 node	4 nodes (strategy OFF)	4 nodes (strategy ON)	8 nodes (strategy OFF)	8 nodes (strategy ON)
Total execution time	7.7196	3.7519	3.6495	4.0913	3.9716
Running time	7.4803	2.8619	2.8692	2.9334	3.0045

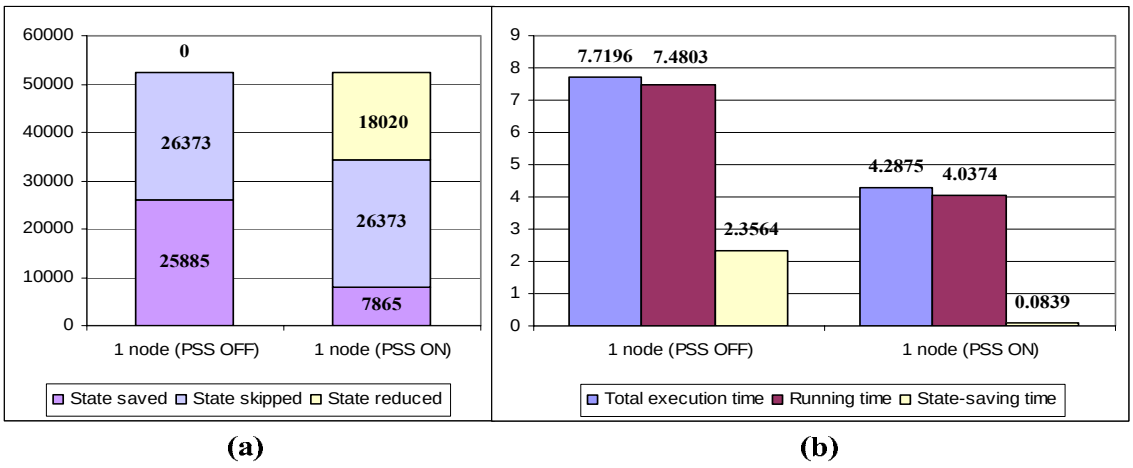
Figure 82. Total execution and running time for the 35×35 fire model

There is no obvious performance gain due to the one anti-message per rollback strategy found in the actual running time. Rather, the running time increases by 0.26% on 4 nodes and by 2.42% on 8 nodes when this strategy is turned on, mainly because the number of rollbacks is increased proportionally in these scenarios. In our simulations, inter-LP anti-messages, which have higher communication overhead, constitute just a small fraction of the total anti-messages generated during rollbacks. Most of the anti-messages are exchanged locally between PCD++ processors within a LP via the main memory. Hence, the sharp reduction in the number of anti-messages does not translate into significant decrease in communication overhead. Further, the models are executed over a fast Gigabit Ethernet network. The performance gain of the strategy would be more visible on slower networks.

- **Periodic state saving**

The 35×35 fire model was executed on 1 and 4 nodes to measure the performance of the PSS strategy. The *state-period* variable was set to 2 in our experiments. Hence, the states of PCD++ processors were saved every two wall clock time slices during the simulation.

Figure 83 shows the execution results on a single node. In this case, 51,028 events are executed in the simulation. If the CSS strategy is used, there would be this number of sates saved in the state queues. As shown in Figure 83(a), 26,373 states (51.68%) are skipped by the MTSS strategy at the upper level of the UCSS mechanism, while an additional 18,020 states (35.31%) are reduced by the PSS strategy at the lower level. As a result, the actual number of states saved during the simulation is only 7,865 or 13.01% of the total number of executed events.



(a) **(b)**

Figure 83. Execution results for the 35×35 fire model before and after PSS strategy on 1 node

From Figure 83(b), we can see that the time spent on state-saving operations reduces by 96.44% from 2.3546 to 0.0839 seconds under the PSS strategy. The reduced state-saving overhead is reflected in the total execution and running time, which declines by 44.46% and 46.03% respectively.

Since no rollback can happen when the simulation is executed on a single node, any reduction in state-saving overhead will have a positive effect on the overall performance. However, this is not the case when the simulation is run in parallel on multiple nodes. The related execution results on 4 nodes are listed in Table 3.

The number of states saved during the simulation (SS) decreases by 60.6% from 42,558 to 16,767 and the time spent on saving states (ST) drops by 86.87% under the PSS strategy. However, there is a price to pay for this. The direct cost of the PSS strategy is the coasting

forward phase added to each rollback. As we can see, 31,197 events (CFL) are reprocessed during the 7,561 rollbacks (RB), consuming a total of 0.9565 seconds (CFT) processing time. On average, 4.13 events are reprocessed in each coasting forward phase. Further, the presence of the coasting forward phase may lead to more and longer rollbacks, an indirect cost of the PSS strategy. In the experiment, the number of rollbacks happened during the simulation (RB) increases by 23.3%. Also, the average length of each rollback (RBL/RB) rises from 3.59 to 4.18 events (or a 16.43% increase). The time spent on rollback operations increases from 0.2569 to 1.1826 seconds accordingly. Therefore, the overhead of the PSS strategy outweighs its potential benefits in our experiment. As shown in the table, the total execution and running time increases by 4.83% and 8.21% respectively.

Table 3. Execution results for the 35×35 fire model before and after PSS strategy on 4 nodes

	4 nodes (PSS OFF)	**4 nodes (PSS ON)**
T(4)(sec)	3.7519	3.9331
RT(4)(sec)	2.8619	3.0969
SS	42558	16767
SK	40598	45112
SR	–	32237
ST(sec)	0.8741	0.1148
RB	6132	7561
RBL	22008	31578
RBT(sec)	0.2569	1.1826
CFL	–	31197
CFT(sec)	–	0.9565

Currently, the PCD++ simulator only employs a PSS strategy with fixed-sized checkpoint interval, which is set arbitrarily by the user at compile time. To achieve better performance, other adaptive state-saving algorithms as presented in [Lin93, Pal93, Fle95] need to be adopted in the toolkit as well.

- **Lazy cancellation**

The 35×35 fire model was executed on 4 and 8 nodes with and without applying the lazy cancellation strategy respectively. The relevant execution results on 4 nodes are listed in Table 4.

Thanks to the lazy cancellation strategy, the number of rollbacks declines by 22.8% and the time spent on rollback operations decreases by 12.57%. As a result, the total execution and running time reduces moderately by 4.5% and 1.7% respectively. A high ratio of lazy hit (LH) to lazy miss (LM) was observed in our experiments, indicating a good performance of the lazy cancellation strategy. Notice that the average rollback length (RBL/RB) increases from 3.59 to

4.28 events. Since rollbacks are delayed, the overhead associated with each rollback tends to increase under the lazy cancellation scheme. However, this increased overhead is compensated by the reduction in the total number of rollbacks, leading to a positive overall performance gain.

Table 4. Execution results for the 35×35 fire model before and after lazy cancellation on 4 nodes

	4 nodes (lazy cancellation OFF)	**4 nodes (lazy cancellation ON)**
T(4)(sec)	3.7519	3.5830
RT(4)(sec)	2.8619	2.8120
RB	6132	4734
RBL	22008	20274
RBT(sec)	0.2569	0.2246
LH	–	206
LM	–	2

The effect of the lazy cancellation strategy is demonstrated more clearly in the 8-node scenario where rollbacks happened frequently during the simulation, as shown in Table 5.

Table 5. Execution results for the 35×35 fire model before and after lazy cancellation on 8 nodes

	8 nodes (lazy cancellation OFF)	**8 nodes (lazy cancellation ON)**
T(8)(sec)	4.0913	3.7939
RT(8)(sec)	2.9334	2.8318
RB	16750	10392
RBL	47239	38953
RBT(sec)	0.4267	0.3223
LH	–	686
LM	–	5

When the model is executed on 8 nodes, the lazy cancellation strategy reduces the number of rollbacks and the time for rollback operations by 37.96% and 24.47% respectively with an even higher ratio of LH to LM. The resulting performance improvement is better than that observed in the 4-node case as well. The total execution and running time reduces by 7.27% and 3.46% respectively. Like in the previous case, the average rollback length (RBL/RB) increases from 2.82 to 3.75 events. As long as the probability of lazy miss is low, the lazy cancellation strategy can be expected to outperform the aggressive cancellation scheme.

CHAPTER 9 CONCLUSIONS AND FUTURE WORK

This work tackles the problem of executing DEVS and Cell-DEVS models in parallel and distributed environments based on the Time Warp optimistic synchronization protocol. A new extension to the CD++ toolkit, PCD++, was developed in our research to meet the need for faster and more efficient simulation of complex models.

A high-level overview of the WARPED kernel and PCD++ toolkit was provided and the kernel assumptions were clearly summarized. The original kernel algorithms have several flaws that lead to runtime failure. Solutions for these problems were discussed and the kernel algorithms were revised to correctly carry out secondary rollbacks. The Time Warp protocol requires a clear separation between processes and their states, which is too restrictive in some occasions. Therefore, we provided a more flexible mechanism that allows simulator developers to maneuver the data that is inappropriate to be managed by the Time Warp mechanism during rollbacks.

Based on previous studies, we adopted a flattened structure for the PCD++ toolkit to reduce the communication overhead. A special structure called NC Message Bag was defined for inter-LP communications. The algorithms for the four types of DEVS processors, i.e. Simulator, FC, NC, and Root, were redesigned to address the need of distributed optimistic simulation. The mechanisms for starting and terminating the simulation were enhanced in line with the optimistic and decentralized approach to distributed simulation. Three different methods for saving and restoring state variables were proposed and the criteria for choosing the appropriate method for different variables were given.

The message-passing paradigm in PCD++ was illustrated using the event precedence graph. Several key characteristics, especially the multi-round execution of the transition phase, were identified that have a significant impact on the computation of the models. Based on these characteristics, the algorithms for Cell-DEVS models with transport and inertial delays were adapted to the asynchronous state transition paradigm to ensure correct simulation.

The simulation process on each LP was abstracted using the notion of WCTS, which greatly simplifies the task of analyzing the complex message exchanges between the DEVS processors involved in the simulation. The WCTS properties were presented to capture the

essence of optimistic simulation in PCD++. A special dormant state was defined for the NC and algorithms were given for the NC to enter the dormant state and to reactivate the simulation afterwards. Two different solutions to the problem of dealing with rollbacks at virtual time 0 were discussed. Based on their relative merits, we solved this problem using explicit synchronization among the LPs.

A two-level UCSS mechanism was proposed so that simulator developers can utilize more flexible and efficient state-saving techniques during the simulation. This mechanism was then integrated with the copy state-saving strategy to implement the risk-free MTSS strategy, a specific optimization for the PCD++ toolkit that can significantly reduce the number of states saved during the simulation. It was also combined with the periodic state-saving strategy to realize a hybrid technique that allows dynamic integration of different state-saving strategies at runtime.

The speculative computation of the NC may lead to messaging anomalies that cannot be recovered by the kernel rollback operations alone. Two types of anomalies were discussed and the corresponding algorithms for handling these anomalies were presented. The concept of breakpoint state was introduced to the kernel state definition. In addition, the state restoration mechanism was enhanced accordingly to handle the breakpoint states during rollbacks.

To remove the bottleneck caused by file I/O operations, we implemented the one log file per node strategy in the PCD++ toolkit. The number of file descriptors consumed in the simulation is upper-bounded and the operational overhead is reduced significantly under this strategy. Furthermore, several other optimizations to the Time Warp protocol were integrated into the PCD++ toolkit, including the one anti-message per rollback strategy for reducing the overhead of sending anti-messages during rollbacks, the PSS strategy for reducing state-saving overhead, and the lazy cancellation strategy for exploiting parallelism available within a LP.

A series of experiments were conducted to measure the performance of the PCD++ toolkit. Several complex Cell-DEVS models were tested using different sizes of cell spaces and on different number of nodes. A collection of 21 metrics was used to gauge the performance and to profile the simulation system. The effects of different optimization strategies were studied quantitatively. We showed that our optimistic simulator markedly outperforms the conservative one in all testing scenarios. Considerable speedups were observed in our experiments, indicating the PCD++ toolkit is well-suited for simulating large and complex models.

9.1. FUTURE WORK

There are several issues with regard to PCD++ that should be further investigated:

(1) Optimism control. In the WARPED kernel, no restriction exists on the maximum lag in virtual time between the fastest and the slowest LPs. Over-optimism encourages rollbacks and can degrade system performance. Also, it results in poor memory utilization due to the wide gap between GVT and the most recent virtual time in the system. Many schemes have been proposed to introduce conservatism to Time Warp in order to throttle the most speculative computations. Some of these algorithms need to be incorporated into the PCD++ toolkit such as moving time windows (MTW) [Fuj00, Fuj03] and the Filter algorithm [Pra91].

(2) Dynamic load balancing. Load balancing is a vital factor in the performance of distributed simulation. Dynamic load balancing allows processes to migrate over the compute nodes during the execution of parallel simulations, which also helps control over-optimism. In PCD++, mechanisms need to be implemented to support migrating Simulators between LPs. The migration choice is a trade-off between optimizing communication load and computation load.

(3) Kernel tuning. As discussed in Chapter 7, three different Time Warp optimizations are implemented in the PCD++ toolkit. However, many other optimizations have not yet been integrated into the toolkit such as those introduced in Chapter 3. The impact of these optimizations needs to be tested in order to determine the best combination of strategies for simulating DEVS and Cell-DEVS models.

(4) Further experiments. More testing of the PCD++ toolkit using a benchmark such as DEVStone [Gli04] should be conducted to further analyze the performance of the simulator. Different partition strategies need to be tested in the experiments to investigate the appropriate strategies for a set of models with different characteristics. In addition, guidelines need to be provided to users as to how many nodes should be used to execute models with different sizes and characteristics.

REFERENCES

[Ame01] Ameghino J.; Troccoli, A.; Wainer, G. "Models of complex physical systems using Cell-DEVS". The 34th IEEE/SCS Annual Simulation Symposium. 2001.

[Bry77] Bryant, R. E. "Simulation of packet communication architecture computer systems". Massachusetts Institute of Technology. Cambridge, MA. USA. 1977.

[Cha78] Chandy, K. M.; Misra J. "Distributed simulation: A case study in design and verification of distributed programs". IEEE Transactions on Software Engineering. pp.440-452. 1978.

[Che98] Chetlur, M.; Abu-Ghazaleh, N.; Radhakrishnan, R.; Wilsey, P. A. "Optimizing Communication in Time-Warp Simulators". Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS'98). 1998.

[Che04] Cheon, S.; Seo, C.; Park, S.; Zeigler, B. "Design and implementation of distributed DEVS simulation in a peer to peer network system". Advanced Simulation Technologies Conference – Design, Analysis, and Simulation of Distributed Systems Symposium. Arlington, USA. 2004.

[Cho94] Chow, A. C.; Zeigler, B. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism". Proceedings of the Winter Computer Simulation Conference. Orlando, FL. USA. 1994.

[Dab03] D'Abreu, M.; Wainer, G. "Models for Continuous and Hybrid System Simulation". Proceedings of the 2003 Winter Simulation Conference. 2003.

[Dav00] Davila, J.; Uzcagegui, M. "GALATEA: A multi-agent, simulation platform". Proceedings of the International Conference on Modeling, Simulation and Neural Networks. Merida, Venezuela. 2000.

[Del02] de Lara, J.; Vangheluwe, H. "ATOM3: A tool for multi-formalism modeling and meta-modeling". European Joint Conference on Theory and Practice of Software. Grenoble, France. 2002.

[Dso94] D'Souza, L. M.; Fan, X.; Wilsey, P. A. "pGVT: An algorithm for accurate GVT estimation". Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS'94). pp. 102-109. 1994.

[Eln02] Elnozahy, E. N.; Alvisi, L.; Wang, Y. M.; Johnson, D. B. "A survey of rollback-recovery protocols in message-passing systems". ACM Computing Surveys (CSUR). Vol. 34(3), pp. 375-408. 2002.

[Fil02] Filippi, J. B.; Bernardi, F.; Delhom, M. “The JDEVS modeling and simulation environment”. Proceedings of the Integrated Assessment and Decision Support Conference (IEMSS’02). pp. 283-288. Lugano, Switzerland. 2002.

[Fle95] Fleischmann, J.; Wilsey, P.A. “Comparative analysis of periodic state saving techniques in Time Warp simulations”. Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS’95). 1995.

[Fre02] Frey P.; Radhakrishnan, R.; Carter, H.W.; Wilsey, P. A.; Alexander, P. “A formal specification and verification framework for Time Warp-based parallel simulation”. IEEE Transactions on Software Engineering. Vol. 28. No. 1. 2002.

[Fuj90] Fujimoto, R. M. “Optimistic approaches to parallel discrete event simulation”. Transactions of the Society for Computer Simulation. 7(2):153-191. June 1990.

[Fuj00] Fujimoto, R. M. “Parallel and Distributed Simulation Systems”. A Wiley-Interscience publication. ISBN 0-471-18383-0. 2000.

[Fuj03] Fujimoto, R. M. “Distributed simulation systems”. Proceedings of the 2003 Winter Simulation Conference. pp. 124-134. 2003.

[Gia76] Giambiasi, N.; Miara, A. “SILOG: A practical tool for digital logia circuit simulation”. Proceedings of the 16th D.A.C. San Diego. 1976.

[Gli04] Glinsky, E. “New Techniques for Parallel Simulation of DEVS and Cell-DEVS Models in CD++”. M. A. Sc. Thesis. Carleton University. Canada. 2004.

[Gro96] Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. “A high-performance, portable implementation of the MPI message-passing interface standard”. Parallel Computing. Vol. 22, pp. 789-828. 1996.

[Gru93] Grunwald, D.; Zorn, B. “CustoMalloc: Efficient Synthesized Memory Allocators”. Software – Practice and Experience. Vol. 23, pp. 851-869. 1993.

[Jef85] Jefferson, D. “Virtual Time”. ACM Transactions on Programming Languages and Systems. 7(3):405-425. 1985.

[Kim04] Kim, K.; Kang, W. “CORBA-based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One”. International Conference on Computational Science and Its Applications (ICCSA). Assisi, Italy. 2004.

[Knu73] Knuth, D. E. “Fundamental Algorithms”. Vol. 1. The Art of Computer Programming. Second Edition. Addison-Wesley. 1973.

[Kof03] Kofman, E.; Lapadula, M.; Pagliero, E. “PowerDEVS: a DEVS-based environment for hybrid system modeling and simulation”. Technical Report LSD0306. LSD, University Nacional de Rosario. 2003.

[Lin91] Lin, Y. B.; Lazowska, E. D. "A study of Time Warp Rollback Mechanisms". ACM Transactions on Modeling and Computer Simulations. Vol. 1, No. 1. January 1991.

[Lin93] Lin, Y. B.; Preiss, B. R.; Loucks, W. M.; Lazowska, E. D. "Selecting the Checkpoint Interval in Time Warp Simulation". Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS'93). 1993.

[Low99] Lowry, M. C.; Ashenden, P. J.; Hawick, K. A. "Distributed High-Performance Simulation using Time Warp and Java". Technical Report, Department of Computer Science. The University of Adelaide. South Australia. 1999.

[Lub91] Lubachevsky, B.; Weiss, A.; Shwartz, A. "An analysis of rollback-based simulation". ACM Transactions on Modeling and Computer Simulation. Vol. 1, No. 2, pp. 154-193. April 1991.

[Mat93] Mattern, F. "Efficient algorithms for distributed snapshots and global virtual time approximation". Journal of Parallel and Distributed Computing. Vol. 18, No. 4. 1993.

[Mar99] Martin, D. E.; McBrayer, T. J.; Radhakrishnan, R.; Wilsey, P. A. "WARPED – A Time Warp Parallel Discrete Event Simulator (Documentation for version 1.0)". Available at: http://www.ececs.uc.edu/~paw/warped/doc/index.html. 1999.

[Moo96] Moon, Y.; Zeigler, B.; Ball, G.; Guertin, D. P. "DEVS representation of spatially distributed systems: validity, complexity reduction". IEEE Transactions on Systems, Man and Cybernetics. pp. 288-296. 1996.

[Nut06] Nutaro, J. ADEVS website. Available at: http://www.ece.arizona.edu/~nutaro. [Accessed June, 2006]

[Pal93] Palaniswamy, A. C.; Wilsey, P. A. "An Analytical Comparison of Periodic Checkpointing and Incremental State Saving". Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS'93). 1993.

[Pra91] Prakash A.; Subramanian, R. "Filter: An algorithm for reducing cascaded rollbacks in optimistic distributed simulations". Proceedings of the 24th Annual Simulation Symposium. pp. 123-132. 1991.

[Pra99] Praehofer, H.; Sametinger, J.; Stritzinger, A. "Discrete event simulation using the JavaBeans component model". Proceedings of International Conference on Web-Based Modeling & Simulation. San Francisco, CA. USA. 1999.

[Rad97] Radhakrishnan, R.; Moore, L.; Wilsey, P. A. "External Adjustment of Runtime Parameters in Time Warp Synchronized Parallel Simulators". Proceedings of the 11th International Parallel Processing Symposium. 1997.

[Rad98] Radhakrishnan, R.; Martin, D. E.; Chetlur, M.; Rao, D. M.; Wilsey, P.A. "An Object-Oriented Time Warp Simulation Kernel". Proceedings of the International Symposium on

Computing in Object-Oriented Parallel Environments (ISCOPE'98). Vol. LNCS 1505, pp. 13-23. Springer-Verlag. 1998.

[Rod99] Rodriguez D.; Wainer, G. "New Extensions to the CD++ Tool". Proceedings of the 32nd SCS Summer Computer Simulation Conference. Vancouver, Canada. 1999.

[Ron94] Ronneren, R.; Ayani, R. "Adaptive checkpointing in Time Warp". Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS'94). 1994.

[Rot72] Rothermel, R. "A mathematical model for predicting fire spread in wild-land fuels". Research Paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station. 1972.

[Sar98] Sarjoughian, H. S.; Zeigler, B. "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment". Proceedings of the International Conference on Web-Based Modeling and Simulation. Vol. 5, pp. 29-36. San Diego, CA. USA. 1998.

[Seo04] Seo C.; Park, S.; Kim, B.; Cheon, S.; Zeigler, B. "Implementation of distributed high-performance DEVS simulation framework in the Grid computing environment". Advanced Simulation Technologies Conference (ASTC). Arlington, VA. USA. 2004.

[Sha99] Sharma, G. D.; Abu-Ghazaleh, N. B.; Rajasekaran, U. K. V.; Wilsey, P. A. "Optimizing Message Delivery in Asynchronous Distributed Applications". Proceedings of the 5th International Euro-Par Conference on Parallel Processing. Lecture Notes In Computer Science. Vol. 1685, pp. 1204-1208. 1999.

[Tro01] Troccoli, A.; Wainer, G. "CD++, a tool for simulating Parallel DEVS and Parallel Cell-DEVS models". Técnica Reporta. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina. 2001.

[Tro03] Troccoli, A.; Wainer, G. "Implementing Parallel Cell-DEVS". Proceedings of the 36th Annual Simulation Symposium (ANSS'03). IEEE. 2003.

[Uhr01a] Uhrmacher, A. M. "Dynamic structures in modeling and simulation: a reflective approach". ACM Transactions on Modeling and Computer Simulation. Vol. 11(2), pp. 206-232. 2001.

[Uhr01b] Uhrmacher, A. M.; Kullick, B. G. "Interacting multi-agent and simulation systems – an exploration into Mole and James". Proceedings of the 5th International Conference on Autonomous agents. pp. 122-123. 2001.

[Wai98] Wainer, G.; Giambiasi, N. "Specification, modeling and simulation of timed Cell-DEVS spaces". Technical Report n.: 98-007. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina. 1998.

[Wai99] Wainer, G.; Giambiasi, N. "Avoiding serialization in Timed Cell-DEVS". Proceedings of the 31st SCS Summer Computer Simulation Conference. Chicago. USA. 1999.

[Wai00a] Wainer, G.; Zeigler, B. “Experimental Results of Timed Cell-DEVS Quantization”. Proceedings of AIS’2000. Tucson. Arizona. 2000.

[Wai00b] Wainer, G. “Improved cellular models with Parallel Cell-DEVS”. Transactions of the Society for Computer Simulation International. Vol. 17, No. 2, pp. 73-88. 2000.

[Wai01a] Wainer, G.; Christen G.; Dobniewski, A. “Defining models with the CD++ toolkit”. Proceedings of the European Simulation Symposium. Marseille, France. SCS Publisher. 2001.

[Wai01b] Wainer, G.; Giambiasi, N. “Timed Cell-DEVS: modeling and simulation of cell spaces”. Invited paper for the book Discrete Event Modeling & Simulation: Enabling Future Technologies. Springer-Verlag. 2001.

[Wai02a] Wainer, G. “CD++: a toolkit to develop DEVS models”. Software – Practice and Experience. Vol. 32, pp. 1261-1306. 2002.

[Wai02b] Wainer, G.; Giambiasi, N. “N-dimensional Cell-DEVS models”. Discrete Event Dynamic Systems. Springer Netherlands. ISSN 0924-6703. Vol. 12. No. 2. 2002.

[Wol86] Wolfram, S. “Theory and applications of cellular automata”. Vol. 1. Advances Series on Complex Systems. World Scientific. Singapore. 1986.

[Zei76] Zeigler, B. “Theory of modeling and simulation”. First Edition. Wiley. 1976.

[Zei93] Zeigler, B.; Kim. J. “Extending the DEVS-Scheme knowledge-based simulation environment for real-time event-based control”. IEEE Transactions on Robotics and Automation. Vol. 9(3), pp. 351-356. 1993.

[Zei96] Zeigler, B.; Moon, Y.; Kim, D.; Kim, J. G. “DEVS-C++: A high performance modeling and simulation environment”. The 29th Hawaii International Conference on System Sciences. 1996.

[Zei98a] Zeigler, B. “DEVS theory of quantization”. DARPA Contract N6133997K-0007. ECE Department. University of Arizona. Tucson. 1998.

[Zei98b] Zeigler, B.; Cho, H.; Lee, J.; Sarjoughian, H. “The DEVS/HLA distributed simulation environment and its support for predictive filtering”. DARPA Contract N6133997K-0007. ECE Department. University of Arizona. Tucson. 1998.

[Zei99a] Zeigler, B.; Kim, D.; Buckley, S. “Distributed supply chain simulation in a DEVS/CORBA execution environment”. Proceedings of the 1999 Winter Simulation Conference. 1999.

[Zei99b] Zeigler, B.; Sarjoughian H. S. “Support for hierarchical modular component-based model construction in DEVS/HLA”. Simulation Interoperability Workshop. 1999.

[Zei00] Zeigler, B.; Kim, T.; Praehofer, H. “Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems”. Academic Press. 2000.

[Zha06] Zhang, M.; Zeigler, B.; Hammonds, P. “DEVS/RMI – An auto-adaptive and reconfigurable distributed simulation environment for engineering studies”. DEVS Integrative M&S Symposium (DEVS’06). Huntsville, Alabama, USA. 2006.