NPSI Adaptive Synchronization Algorithms for Parallel Discrete Event Simulation

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Sudhir Srinivasan

August 1995

August 1995

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy (Computer Science)

Sudhir Srinivasan

This dissertation has been read and approved by the Examining Committee:

Paul F. Reynolds, Jr. (Dissertation Advisor)

James M. Ortega (Committee Chairperson)

Andrew S. Grimshaw

Jörg Liebeherr

Ronald D. Williams

Accepted for the School of Engineering and Applied Science:

Dean Richard W. Miksad
School of Engineering and Applied Science

August 1995

To My Parents

Abstract

Synchronization of parallel discrete-event simulations (PDES) is a difficult problem. Despite a decade and a half of research with several successes, a synchronization algorithm that consistently performs well over a wide range of applications eludes the community. This is due primarily to the fact that the synchronization requirements of simulations are generally irregular, dynamic and unpredictable. The thesis developed here is that an adaptive approach based on the use of *near-perfect state information* (NPSI) offers significant potential to be consistently efficient.

We present two contributions of our investigations into optimality of adaptive protocols: a *correct* sufficient condition for a protocol to be super-critical, and proof that independence among events is not necessary for super-critical speed, an observation that has significance in lower bound theory of PDES.

We establish the foundation for a new class of synchronization algorithms called NPSI adaptive synchronization algorithms (protocols). These are characterized by the use of near-perfect state information to adaptively control the optimism of processes in a PDES. We propose a design framework for NPSI protocols and describe the *Elastic Time Algorithm* (ETA) based on this framework. An extensive performance study shows that ETA consistently outperforms Time Warp — the protocol favored by many researchers — over a wide range of workloads. The success of ETA substantiates our thesis and is a significant contribution of our work to PDES research.

In the first analytical result concerning adaptive protocols, we identify a general class of adaptive protocols (the AAWP's) and show that Time Warp and AAWP's can arbitrarily outperform each other. This result is significant as it demonstrates a theoretical limitation of AAWP's and suggests that adaptive protocols must be designed with care.

Adviser:

Paul F. Reynolds, Jr.
Associate Professor
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Acknowledgments

I am deeply grateful to my parents for instilling in me the value of a good education. This dissertation would not have been possible without all the encouragement they have given me and the sacrifices they have made. Their confidence and pride in my achievements have helped carry me through several grueling times. I thank them for providing a good family environment and for supporting me in various decisions in my life.

To Paul, my advisor and good friend, I express my heartfelt appreciation for his invaluable encouragement throughout my years at UVa. His generosity, imagination and ability to work on several problems at once have been inspirational. I am grateful to him for educating me on the various aspects of being a good researcher. I thank him for the countless hours spent discussing non-academic matters of life. I treasure this opportunity I have had to work with him and look forward to many more.

Words cannot express how much the love and unfailing confidence of my dear wife, Padmini, have meant to me these five years. I thank her for always being there for me (despite a commute marriage) and for being the stabilizing force in my life.

I am grateful to my brothers, Sriram and Sanjay and my sisters-in-law for their constant support. The encouragement I have always received from my aunts and uncles and from Padmini's family is deeply appreciated. I cherish the enduring friendship of the "PU gang" — Raghu, Kiran, Bala, Rashmi, Dinesh and Rajesh.

The many research meetings with the Smart Interconnection Networks group have almost always been productive for me. I thank Craig Williams and Bronis de Supinski for numerous exciting discussions — it has been a pleasure working with them.

The expert assistance of Bob Ross in grappling with the PRN is greatly appreciated. Some of the most entertaining moments of my student life were spent listening to Bob while slaving over the Beast.

I thank my committee members Professors James Ortega, Andrew Grimshaw, Jorg Liebeherr and Ronald Williams for their insights and suggestions about my research. I also thank the faculty of the Computer Science Department for their support.

I am especially thankful to Ray Lubinsky and Ann Bailey for helping me out with the computer systems (particularly the cluster machines and SES licensing) many times

over the years. Ann’s concern for my simulations during crunch time is much appreciated. I am grateful to Ginny, Brenda, Tammy and Barbara for their invaluable assistance with the many bureaucratic aspects of graduate life.

Many factors helped me get through rough times during the two years of weekend commutes. I thank my room buddies Dallas, Laura, Emily, Charlie, Matt, Alan and Mary-Ann for keeping the hours lively. Many a bug has been removed while soaking up the sun on the wax-stained front steps of Olsson Hall. I thank Isaac Asimov for Daneel and Giskard, the Robots of Dawn that kept me company during the last few crucial months.

I am grateful to Robin Arietta of Scientific and Engineering Software for providing us with the ultimate parallelism in simulation: multiple licenses!

This work was supported in part by Mystech Associates, Inc. (Academic Affiliates Program).

Contents

List of Figures

List of Symbols

EP_i. . . . . *error potential* for LP_i

GVT . . . . *global virtual time* in an aggressive simulation, defined as the smallest timestamp among all of the following: (i) unexecuted events, (ii) events being executed, and (iii) messages in transit

GVT_a (*t*) . *actual* value of GVT (as defined above) at time *t*

GVT_c (*t*) . *computed* value of GVT at time *t* — an approximation of GVT_a (*t*) provided by any GVT computation algorithm

M_1 . . . . . mapping to compute EP_i

M_2 . . . . . mapping to translate EP_i into control over optimism

s. . . . . *slope*, the scaling factor used in the M_2 for the Elastic Time Algorithm

$TSVT_i$ . . . *target-specific virtual time* in an aggressive simulation, defined as the value of GVT computed over the predecessor set of LP_i

α_i . . . . . *minimum future time* of LP_i defined as MIN (η_i, υ_i)

α_i' . . . . . $MIN_{LP_k}(\alpha_i)$ $\forall LP_k \in$ the predecessor set of LP_i

δ_i . . . . . real-time delay computed by the M_2 for the Elastic Time Algorithm

η_i . . . . . *next event time* of LP_i defined as σ_i while LP_i executes an event or the timestamp of the next scheduled event otherwise

μ_i . . . . . *minimum time of* LP_i defined as MIN (σ_i, υ_i)

ρ_i . . . . . primary acknowledgment submitted by LP_i

ρ' . . . . . global minimum primary acknowledgment

σ_i . . . . . *logical clock* of LP_i

τ_i . . . . . secondary acknowledgment submitted by LP_i

τ' . . . . . global minimum secondary acknowledgment

υ_i . . . . . *minimum unreceived message time* defined as the smallest of the timestamps of all messages sent by LP_i that have not been received or processed

Chapter 1

Introduction

We address the parallelization of discrete-event simulations. Discrete-event simulation has proven to be a difficult application to parallelize despite significant amounts of potential parallelism. Traditional approaches employing less dynamic analysis than that developed here have failed to provide consistently good performance across a wide range of applications. The thesis developed here is that our approach — NPSI adaptive synchronization algorithms — offers significant promise for providing consistently good performance.

In this chapter, we describe the synchronization problem that is at the core of parallel discrete-event simulation (PDES). We introduce and explain important terminology in PDES. Traditionally, two opposite approaches have been pursued, namely conservative and optimistic — we describe each briefly. We close with a brief overview of NPSI adaptive synchronization algorithms.

1.1 Discrete-Event Simulation

Discrete-event simulation (DES) is a widely used technique for system modeling — combat training, telecommunication networks, computer systems, traffic networks and control systems are but a few of the many areas where simulation is used. It is used for a variety of reasons including training of personnel, system design verification and performance evaluation. A DES can be viewed as a sequence of state changes or *events*, determined by the nature of the system being simulated (commonly referred to as the *physical system*). The notion of time in the physical system is translated into simulated or *logical time* in a simulation. Each event is assumed to occur at a discrete point in logical time and is therefore associated with a *logical timestamp* (or simply, timestamp). Clearly, the simulation will be correct (or accurate) if events are executed in non-decreasing order by timestamp, since otherwise it is possible to have an event affect its past (which is impossible in the physical system).

A sequential simulator typically has three data structures: (i) the *events list* — a list of events scheduled to be executed, sorted in ascending order of their timestamps, (ii) the *logical clock* — a value that represents the time up to which the physical system has been simulated and usually equals the timestamp of the event being executed currently, and (iii) the *state variables* — application-specific variables that describe the state of the physical system. Figure 1 shows the sequential DES algorithm. The simulator proceeds by removing the first event from the events list and executing it. Execution of an event may cause other events to be placed on the events list. Thus there is a causal relation between (some) events. A simulator correctly simulates the physical system if it maintains the causal relations among the events of the physical system. Executing events in strict timestamp order guarantees correctness.

1.2 Parallel Discrete-Event Simulation

As the name suggests, *parallel discrete-event simulation* (PDES) is the process of executing discrete-event simulations on multiple processors. Although PDES arose as a case study in the design and verification of distributed programs [ChMi79], it has developed into a significant research problem in its own right. PDES has become very important because sequential discrete event simulations consume unacceptably large amounts of time. A typical engineering simulation run executes on the order of billions of events, requiring up to several days or more of computation time.

The common approach to parallelizing discrete event simulations is to model the physical system as a collection of separate physical entities called *physical processes* (PP's). The PP's interact with each other using messages. In the parallel discrete event simulator, each PP is simulated by a *logical process* (LP) which is essentially a sequential discrete event simulator with its own logical clock and events list and the state variables that

---

```
1.    While simulation is not complete
2.      Set logical clock equal to timestamp of first event on events
            list
3.      Remove first event from events list and execute it
4.      Schedule future events as required
5.    Endwhile
```

Figure 1 - Event execution loop for a sequential discrete-event simulator

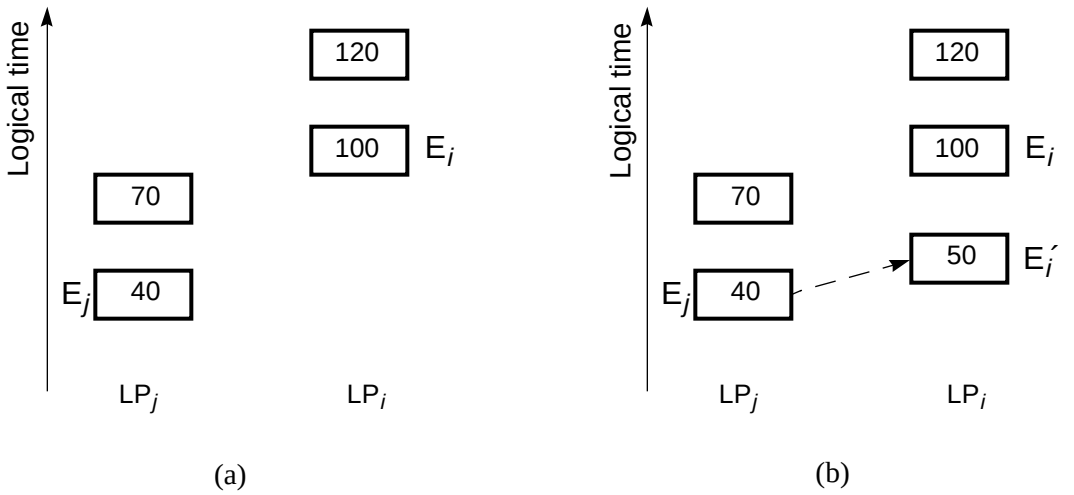


Figure 2 - Synchronization problem in PDES

describe the PP it simulates. The interactions among PP's are captured via the exchange of timestamped messages between LP's over logical *channels* — there is no shared state. The LP's are executed on multiple processors with the goal of reducing the completion time of the simulation. Since the messages exchanged by the PP's usually result in some activity at the receiving PP's, receipt of a timestamped message at an LP usually schedules one or more events or affects the state of the LP receiving the message. This introduces a synchronization problem which is central to PDES.

Consider the example in Figure 2. In Figure 2a, the next scheduled event at LP_i (E_i) has a timestamp = 100 and the next scheduled event at LP_j (E_j) has a timestamp = 40. There are two possibilities regarding the next state: either execution of E_j results in a timestamped message that schedules an event at LP_i (E_i') with timestamp < 100 (Figure 2b), or it does not. The problem that LP_i faces is to decide whether it is safe to execute E_i. On the one hand assume LP_i executes E_i and later, E_j schedules E_i' at LP_i such that E_i' affects the state of LP_i on which E_i executes. In this situation, a *causality error* is said to have occurred since LP_i has violated a causal dependence of E_i on E_i'. Physically a causality error is equivalent to the future affecting the past. On the other hand, LP_i may execute E_i and the execution of E_j may not schedule E_i' *or* E_j may schedule E_i' such that E_i' does *not* affect the state on which E_i executes. In this case, there is no causality error since there is no causal dependence from E_j to E_i. In general, a *causal dependence* is said to exist from event *e* to event *e'* if the execution of *e* affects the state on which *e'* executes. The synchronization problem

in PDES is to determine efficiently when an LP can process events such that at the end of the simulation, *it appears as if* there were no causality errors*. At a high level, this is the problem we are attempting to solve.

PDES is hard because the cause-and-effect relationships in the physical system — which translate to causal dependences in the PDES and therefore dictate the order in which events must appear to be executed — are usually irregular, complex and highly data-dependent (for instance, random numbers play a major role in determining these relationships). This is in sharp contrast with other parallel applications such as numerical computations on large matrices, where the synchronization constraints are regular and fairly predictable.

Clearly, as defined above, causal dependence between events is application-specific. To solve the synchronization problem in a general way, we require a general characterization of *causality*. Note, if *e´* is causally dependent on *e*, the timestamp of *e must* be less than the timestamp of *e´* (cause and effect). Based on this observation, causality may be characterized in general using timestamps. We defer a formal definition to Chapter 3. Timestamp-based causality is captured in the *local causality constraint* [Fuji90] which states that a PDES as described above adheres to the local causality constraint if and only if each LP executes events in non-decreasing timestamp order. It follows that a simulation is guaranteed to be free from causality errors if it adheres to the local causality constraint. Thus, a *causality error* can be defined generally as a violation of the local causality constraint. Note, however, this definition is stronger than actually required: in the example of Figure 2, as mentioned above, E_j may schedule E_i´, but E_i´ may not affect the state on which E_i executes. In this case, executing E_i before executing E_i´ does not adhere to the local causality constraint but is correct, nevertheless.

1.3 Traditional Solutions

Much of the research in PDES has concentrated on solving the synchronization problem described above. Many synchronization algorithms (also called *protocols* in the literature and in this document) have been proposed [Fuji90] that enforce the local causality constraint (with few exceptions). Broadly, PDES protocols have been classified as *conser-*

*Note, it is not *required* that events be executed in the order dictated by the causal relationships. All that is needed is that the overall effect of the execution is equivalent to (i.e. has the same outcome as) the causal order of execution. This topic is discussed in more detail in Chapter 3.

```
1.    While simulation is not complete
2.      Block until next event is safe
3.      Set logical clock equal to timestamp of first event on events
            list
4.      Remove first event from events list and execute it
5.      Schedule future events and send messages as required
6.    Endwhile
```

Figure 3 - Event execution loop for a conservative parallel discrete-event simulator

vative or *optimistic* based on the strategy for enforcing the local causality constraint. As pointed out in [Reyn88], this is only one of nine dimensions along which protocols can be classified. We describe the salient aspects of each.

1.3.1 Conservative Protocols

Historically, the first protocols were conservative. As the name implies, these protocols execute an event only after determining that the event is *safe* — i.e., execution of that event will not result in a causality error in the future. The sequential event execution loop (Figure 1) is modified for conservative parallel execution as shown in Figure 3*. The safety of an event is determined (line 2) generally by exchange of information among the LP's. LP's may send *protocol messages* (as opposed to simulation messages) that carry information that allows other LP's to proceed. Since LP's may block, these protocols are prone to deadlock. Conservative protocols are designed to either avoid deadlocks or detect deadlocks and recover from them.

An example of a conservative approach is the Null-Message (or Link-Time) protocol [ChMi79, PeWM79]. This protocol operates on the property that the timestamps of messages sent along any channel are monotonically increasing. The safety of an event is determined by the arrival of at least one message on every incoming channel. If a message has arrived on every channel, the monotonicity of timestamps over channels implies the message with the smallest timestamp can be safely processed (i.e. converted into an event and executed). This mechanism can lead to significant blocking periods because some channels may carry very little message traffic. To overcome this, the protocol introduces *null messages* — protocol messages that carry only a timestamp and no simulation specific

*Note, this algorithm represents the actions of a single LP.

```
1.    While simulation is not complete
2.      Set logical clock equal to timestamp of first event on events
            list
3.      Remove first event from events list and execute it
4.      Schedule future events and send messages as required
5.      Roll back if necessary
6.      Save state
7.      Collect fossils
8.    Endwhile
```

Figure 4 - Event execution loop for an optimistic parallel discrete-event simulator

information. When an LP sends a simulation message along a particular channel, it sends a null message with the same timestamp as the simulation message along all other outgoing channels. Arrival of a null message on a channel is equivalent to the arrival of a simulation message along that channel for determining the safety of events (since only timestamps are used in this process). Thus, a null message is a guarantee that the LP sending will not send any messages with a smaller timestamp. Although null messages help to unblock LP's, this mechanism can result in a deadlock if the simulation contains a cycle of LP's such that a message may traverse the entire cycle without an increase in timestamp (i.e. if the minimum timestamp increment along every channel in the cycle is zero). The requirement of at least one non-zero minimum timestamp increment restricts the applications for which this scheme can be used.

1.3.2 Optimistic Protocols

Optimistic protocols execute events *without* the guarantee of freedom from causality errors — i.e. they allow the possibility of causality errors. In the event a causality error does occur, the common approach is to recover from the error using a rollback mechanism. Figure 4 shows a typical event execution loop for Time Warp [Jeff85], an optimistic protocol. In *forward execution*, an LP executes events unconditionally as they arrive. Messages sent during forward execution are called *positive messages*. Periodically, an LP saves its state. If a message arrives in an LP's logical past (i.e. with a timestamp smaller than the LP's logical clock) — a *straggler* — a causality error is detected (by the local causality constraint). The LP then performs a *rollback*: (i) it rebuilds its state at the time of the straggler (*rollback time*) using a previously saved state with a timestamp less than or equal to the roll-

back time, and (ii) it sends out *antimessages* to *annihilate* any positive messages sent during the forward processing just rolled back. Rollbacks can *cascade*, since antimessages can cause rollbacks at other LP's if the corresponding erroneous positive messages have already been processed. A rollback initiated by the arrival of a straggler that is also a true message is called a *primary rollback* while one initiated by the arrival of an antimessage is called a *secondary rollback*. Since logical clocks of LP's can move forward as well as roll back, we require a measure of the progress of the simulation. *Global virtual time* (GVT) [Jeff85] is defined for general optimistic systems as the smallest timestamp among all of the following: (i) unexecuted events, (ii) events being executed, and (iii) messages in transit (not yet received and/or processed). GVT is the logical time up to which the simulation is guaranteed to be accurate (free from causality errors) and hence denotes the progress of the simulation. Consequently, GVT is useful for several reasons, including making the effects of event executions visible (especially in interactive simulations). A particularly important use of GVT is in *fossil collection* — reclaiming memory that is no longer relevant to the simulation. Memory objects (events, saved states, antimessages, etc.) with timestamps less than GVT can no longer affect the simulation and can therefore be relinquished. Since state saving can consume enormous amounts of memory, management of memory is crucial in optimistic protocols. Appendix II describes the details of Time Warp and related concepts such as cancellation strategies and event pre-emption.

1.3.3 Discussion

Both conservative and optimistic strategies have their pros and cons. The main advantage of conservative protocols is that they do not incur checkpointing and rollback overheads. However, processes may block due to insufficient information when in fact they could proceed. This *artificial blocking* [Reyn82] introduces *lost-opportunity cost* [SrRe95b]. On the other hand, optimistic protocols do not incur lost-opportunity costs since processes do not block. The main hazard of optimistic protocols is that state saving and rollback costs may degrade performance severely. A number of techniques have been proposed to reduce overheads in the conservative and optimistic approaches.

Studies indicate PDES is a viable technology ([Fuji93] provides a summary of published results). However, performance is usually dependent on the characteristics of the

application being simulated (sometimes even the particular instance of it) and the architecture of the multiprocessor. A consistently efficient protocol that performs well across a wide range of applications still eludes the community. The principal reason for this is that in large part, current approaches do not adapt to the dynamic, unpredictable nature of the synchronization requirements of parallel simulations [NiRe90, Fuji90]. To this end, *adaptive protocols* — those that change the bindings of one or more of their design variables dynamically [Reyn88] — offer a promising approach [ORSA93].

1.4 NPSI Adaptive Protocols

We propose a new class of PDES protocols, the NPSI adaptive protocols, characterized by the use of *near-perfect state information* (NPSI) to adaptively control optimism. NPSI protocols operate by computing an *error potential* (EP) based on NPSI supplied by an asynchronous feedback system, and translating this error potential into control over an LP's optimism. The EP may be thought of as an indicator of the likelihood of a rollback in the near future. One of the key issues in designing NPSI adaptive protocols is identifying the kinds of state information that lead to an accurate EP and thus enhance performance. Our thesis is that an adaptive approach based on low cost near-perfect state information has significant potential to yield a consistently efficient synchronization algorithm.

We describe the Elastic Time Algorithm (ETA) — a very promising NPSI protocol. ETA uses simple state information (logical clock, unreceived message time and next event time) to compute EP and controls optimism by introducing delays proportional to EP between event executions. The delays are asynchronous and independently computed for each LP. We call this the *microadaptive* approach, as opposed to the typical (*macroadaptive*) approach where adaptiveness occurs in the form of relatively infrequent changes to system parameters based on system monitoring. A performance study of ETA shows it to be highly efficient over a wide range of workloads and scalable over moderately parallel systems. The primary contributions of our research are to lay the foundation for NPSI adaptive protocols and to demonstrate their potential to be consistently efficient.

1.5 Thesis Outline

In **chapter 2** we survey previous PDES synchronization algorithms. We analyze the merits and demerits of traditional solutions as well as several recent hybrid approaches that limit optimism.

In **chapter 3** we describe our contributions in the area of critical path analysis of PDES's, resulting from our research into the optimality of NPSI adaptive protocols. The first contribution is a sufficient condition to determine whether a protocol can execute a simulation in less than the critical path time. In addition, we show that a similar condition used in previous studies is necessary but not sufficient. The second contribution is to disprove the claim that super-critical speed requires independence among certain events and analyze the impact of this result on general lower bounds for PDES's.

In **chapter 4** we develop the theory of NPSI adaptive protocols. We motivate the approach, describe the reduction model underlying NPSI protocols, present a framework for the design and study of NPSI protocols, discuss design issues based on this framework and explore several NPSI protocols.

In **chapter 5** we illustrate the use of the reduction model (defined in chapter 4) to compute NPSI. Specifically, we describe *npGVT*, an algorithm to compute accurate GVT in an optimistic PDES and prove its correctness. Also, we describe an implementation of the reduction model and of *npGVT* over this implementation and prove its correctness. These algorithms provide a base for implementing NPSI adaptive protocols.

In **chapter 6** we present the Elastic Time Algorithm (ETA), our NPSI adaptive protocol of choice. We describe a performance study of ETA, including a description of the suite of test workloads and the workload generator developed to realize them. The demonstrated success of ETA over a wide range of workloads is a significant contribution of our work to the field of parallel discrete-event simulation. In addition, we describe a second NPSI adaptive protocol called ZETA, for two reasons: (i) ZETA is an intuitive choice for an NPSI protocol, and (ii) our study of its weaknesses led to the design of the more efficient ETA.

In **chapter 7** we present an analytical comparison of a class of adaptive protocols with Time Warp. This is the first known analytical result concerning adaptive PDES protocols. The class of adaptive protocols to which this analysis applies (called AAWP) includes

several practical protocols, among them NPSI adaptive protocols. Since adaptively controlled optimism is expected to enhance performance (as observed in practice), our analysis is significant because it shows that in theory, it is possible for Time Warp and AAWP's to arbitrarily outperform each other.

In **chapter 8** we discuss tunable parameters in adaptive protocols — parameters whose values must be adjusted dynamically or in an application-specific manner to ensure good performance. We classify tunable parameters, identify metrics to tune them and present an algorithm that uses one of these metrics to tune the value of a parameter in ETA. Performance analysis shows this algorithm is very efficient.

In **chapter 9** we summarize our results, outline areas of future research and comment on the significance, strengths and weaknesses of the NPSI approach.

Chapter 2

Related Work

The synchronization problem remains the central challenge in PDES [Fuji93b, FuNi92, Fuji90]. We analyze the advantages and drawbacks of traditional approaches to synchronization in PDES. Based on this analysis, we believe controlled optimism will be a feature of a consistently efficient protocol. We discuss several recent protocols that limit optimism.

2.1 Analysis of Traditional PDES Protocols

A systematic study of protocol design [Reyn88] revealed nine *design variables* whose bindings can be varied to devise different protocols. The variables are: partitioning, adaptability, aggressiveness, accuracy, risk, synchrony, knowledge embedding, knowledge dissemination and knowledge acquisition. A brief description of each of these variables is provided in Appendix I. Reynolds points out that despite the large number of protocols that have been proposed, only a small portion of the design space has been explored. Specifically, he categorizes several protocols based on the design variables — it is worth noting that adaptability has not been considered in any of these protocols. While all nine variables are important, we restrict our attention to aggressiveness, accuracy, risk and synchrony since these are the main factors that control event processing.

- *Accuracy* guarantees that the output of a PDES appears as if no causality errors occurred.
- *Aggressiveness* allows an LP to execute events without the guarantee of freedom from causality errors.
- *Risk* allows an LP to propagate the results of aggressive execution of events to other LP's.
- *Synchrony* describes the amount of synchronization among LP's. In the PDES context, LP's that synchronize with each other only through the exchange of timestamped messages (as we have described them thus far) are said to be

asynchronous. LP's may also simulate in phases where periods of simulation are separated by periods of global synchronization — called a *synchronous* approach.

Based on these variables, we group traditional protocols into four categories and discuss the limitations of each.

2.1.1 Conservative Protocols

In the conservative approach, an LP does not execute its next event until it can ascertain that no event will be received later with a smaller timestamp (i.e. no causality constraints will be violated by this execution). The observable effect of this approach is that LP's always execute events in timestamp order. Thus, these protocols are accurate, non-aggressive, without risk and asynchronous. Examples of conservative protocols are Null Messages [ChMi79], Deadlock Detection and Recovery [ChMi81], SRADS [Reyn82], and Appointments [NiRe84]. The following critique addresses the more general class of non-aggressive protocols.

The main problem with these protocols is that due to their non-aggressive nature, they tend not to exploit much of the concurrency in the simulation. LP's may be idle for significant amounts of time, unnecessarily waiting for information from other LP's. This phenomenon is called *artificial blocking* [Reyn82]. A clear example of how artificial blocking increases the completion time of non-aggressive simulations is seen in a critical path analysis of the example described in [LiMi90]. Here, a simulation that should take O(1) time takes O(n) time with n LP's. The reason for artificial blocking is the inherently static nature of non-aggressive protocols. Since LP's must detect the *lack* of dependence on events at other LP's, the communication topology of the LP's must be known. The more strongly connected this topology is, the more the artificial blocking. In fact, in the worst case of a completely connected system, the only events that will be executed concurrently are those with identical timestamps (i.e., the simulation is almost sequential).

Blocking introduces the potential for deadlock. Handling of deadlock (either avoidance or detection and recovery) introduces overheads into the simulation. Deadlock avoidance requires the sending of extra messages [ChMi79] to convey lack-of-dependence information. These control messages can proliferate and degrade performance. Distributed

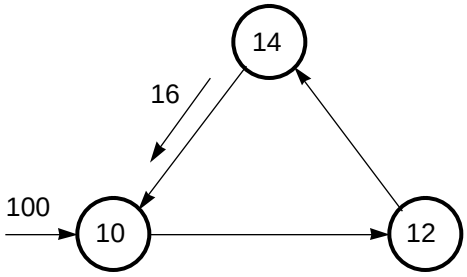


Figure 5 - Sensitivity to lookahead

deadlock detection [ChMi81] is a relatively expensive operation. The performance of these protocols depends on how often deadlocks occur and the cost of detecting and breaking the deadlock.

Finally, the efficiency of non-aggressive protocols is very sensitive to a parameter of the simulated system called *lookahead*. Lookahead is the property of a simulation by which LP's can predict events in their simulated future. The dependence of performance of conservative protocols on lookahead is obvious: the farther an LP can predict into its future, the sooner it can convey lack-of-dependence information to other LP's. Thus, the larger the lookahead, the better the performance. A good summary of the effects of lookahead on the efficiency of conservative protocols is given in [Fuji90]. As a simple example, consider the situation in Figure 5. Here, each of the LP's has a lookahead of 2 logical time units. There are no pending events at any LP except the external event with timestamp 100. Before this event can be executed, the *null* message in transit with timestamp 16 must circulate in the system, its timestamp increasing by 2 logical time units (the lookahead) at each LP, until it equals or exceeds 100. Thus, the message must circulate 14 times before the external event may be executed.

In summary, conservative protocols are useful in restricted situations where there is significant lookahead and the communication topology is sparse.

2.1.2 Optimistic Protocols

These protocols process events aggressively, i.e. they do not wait for a guarantee of freedom from causality errors. In contrast with non-aggressive protocols which detect the *lack* of dependence between events, optimistic protocols *assume* the lack of dependence, detect the *existence* of dependence through detection of causality errors and recover from these errors. By definition, these protocols are aggressive. Since the results of aggressive

execution of events are propagated to other LP's, these protocols are accurate, aggressive, with risk and asynchronous. It is important to note that optimistic protocols are a subset of the more general class of aggressive protocols. While aggressiveness refers only to the execution of events, *optimism* also includes risk, the propagation of aggressive computation to other LP's.

Optimistic protocols face three major problems. First, the time cost of periodic state saving can be very high. The use of special state-saving hardware [FuTG88] can almost eliminate this cost. However, the limited memory capacity of this hardware may become a problem. Other techniques to reduce the time cost of state saving are: (i) saving state less often, and (ii) incremental state saving [PaWi92]. Infrequent state saving tends to increase the depth of rollback while incremental state saving increases the time to rebuild a state after a rollback. Properties of applications that alleviate this cost are: (i) small state size (e.g. fine grained simulations such as logic simulations), and (ii) a kind of "spatial locality" where only small portions of the state are modified over several successive events.

Second, these protocols tend to use much more memory than sequential simulations. This is due to two reasons: state saving and aggressive processing. Memory management schemes are necessary to keep the memory consumption from degrading performance. For saved state, garbage (fossil) collection is used to reclaim states which are no longer needed. Fossil collection requires estimation of *global virtual time*, which is a non-trivial operation. Due to aggressive processing, the future events list of LP's can become very large and cause them to block due to insufficient memory. In these cases, memory must be reclaimed from the future events list by rolling back some LP's. Memory management schemes introduce time overheads such as computation of global values, freeing up memory, sending rollback messages, etc. Even though optimistic protocols can operate with as little memory as required by a sequential simulation using these memory management schemes, a significant amount of memory is required for good performance.

Finally, the most serious problem with optimistic protocols is that under certain situations, the overheads of rolling back incorrect computation can degrade performance. Rollback costs are two-fold: *direct* or *local* rollback costs - incurred while performing rollback operations local to the LP and *indirect* or *cascading* rollback costs. When a causality error is detected at an LP and a rollback is initiated, all other computations spawned by this

incorrect computation must also be rolled back. With such cascading rollbacks, the performance can deteriorate when incorrect computations propagate at least as fast as the rollbacks. In such situations, the parallel simulator spends increasing amounts of time rolling back incorrect computation rather than performing correct computation. Finally, a combination of these two costs can lead to a phenomenon known as *echoing* [LuWS91] which reduces the efficiency of the simulation to zero asymptotically. Echoing occurs when LP's require more time to roll back a certain amount of logical time than to simulate forward the same amount of logical time. In such a case, a set of LP's may become locked in a cycle of cascading rollbacks that remains within that set and is such that the amplitude of each rollback is greater than that of the previous one. The cycle of rollbacks never ends and the simulator takes increasing amounts of real time to make progress.

In summary, the main problem with optimistic protocols is that *they are too optimistic*. This results in high rollback and memory management costs when the application exhibits load imbalance or changing locality of load (for example, logic simulations, Petri Net simulations and simulations of telecommunication networks). Optimistic protocols perform best in situations where state saving and rollback costs are low and sufficient memory is available.

2.1.3 Synchronous Protocols

These protocols take the iterative or synchronous approach. Typically, the simulation occurs in alternating phases of computation (during which LP's simulate events) and synchronization (during which LP's exchange information to enable future events). *Windowing* protocols constitute a large class of synchronous protocols in which LP's synchronize periodically to determine a *window* in the logical future such that events inside this window are safe (thus these protocols form a sub-class of conservative protocols). The rationale is that the overheads of determining the safety of an event are reduced by considering only events within the window. The synchronization phase is usually a barrier synchronization such that all LP's complete the current computation phase before proceeding to the next one. Thus, these protocols are accurate, non-aggressive, without risk and synchronous. Examples include lookahead-based windows [Luba89, AyRa92], Conditional Events [ChSh89] and YAWNS [Nico93].

The main limitation of synchronous protocols is the determination of the size of the logical time window. The optimal size depends on the application and its identification requires user expertise. Performance can be very bad when the window is either too small or too large. Further, these protocols suffer some of the disadvantages of other non-aggressive protocols: inherently static nature, sensitivity to lookahead and to a much lesser extent, overheads of conveying lack-of-dependence information.

2.1.4 Error-Tolerant Protocols

Error-tolerant protocols allow inaccuracies in the simulation in a controlled manner so as to reduce rollback costs by reducing the number of rollbacks (thus they form a subclass of optimistic protocols). When an LP detects a causality error, it may allow the error to persist if the overall effect on the simulation output is not significant. Thus, these protocols are potentially inaccurate, aggressive, with risk and asynchronous.

The main problem with this approach is that it is not general enough to be applied to a wide range of simulations. It is applicable only in situations where the output of the simulation has sufficient variance to hide the effects of the inaccuracy; for example, in Monte Carlo simulations. These protocols have not been thoroughly explored as yet [Movv93, Theo84] and some issues such as the effect of inaccuracies on the computed output measures are yet to be investigated.

2.1.5 Critical Path Analysis

With the lack of a consistently efficient protocol, a natural question to ask is “What is the best we can do?” That is, given a particular simulation run, what is the smallest amount of real time required to execute that particular simulation run? Of course, we must also ask “Does such a lower bound exist?” Knowing the answers to these questions, we may strive to develop protocols that can be shown to achieve or at least approach the above bound (if it exists).

Previously, critical path analysis has been used to derive this lower bound. Based on the causal relations between events, a *dependence graph* can be constructed for any simulation run. This is a weighted acyclic directed graph over which a *critical path* may be defined as the path (or paths) from a source to a sink with the maximum cumulative weight. It has been shown that the time to execute the critical path (critical path time) is a lower

bound on the completion time of any simulation using a non-aggressive protocol [LiLa91]. Typically, this lower bound is a loose one due to the static nature of non-aggressive protocols (see the artificial blocking example mentioned in Section 2.1.1). Initially, it was thought that this lower bound applies to *all* simulations. However, it was shown later that certain aggressive protocols are capable of beating it, i.e. they can complete a simulation in less than the critical path time. This is possible because aggressiveness allows an LP to guess its computation before its critical time. Thus, there exist aggressive protocols for which the critical path time does not serve as a lower bound.

2.2 Protocols With Limited Optimism

Conservative and optimistic protocols lie at opposite ends of a spectrum of possibilities for aggressiveness and risk. Given the static nature of conservative protocols, their dependence on lookahead and their potential inability to achieve even the critical path lower bound, the optimistic approach is widely believed to be the better technique for extracting concurrency in a simulation and consequently, more likely to yield a consistently efficient protocol [Fuji90]. However, since uncontrolled optimism (such as in Time Warp) is prone to poor performance, an approach with controlled optimism is desirable. Several protocols have been proposed that limit optimism in Time Warp. We categorize these previous approaches broadly, based on the criterion for limiting optimism, and discuss each category.

2.2.1 Window Based

The earliest protocols with limited optimism applied the synchronous approach to optimistic protocols. Between synchronizations, LP's execute those events that have timestamps within the agreed-upon window. This ensures that all LP's remain in the same region of logical time and advance through the simulation together. Since the difference between the logical clocks of any two LP's is bounded, *echoing* (Section 2.1.2) cannot occur. Similarly, it is also possible to limit the lengths of rollback chains and therefore control the cascading of rollbacks.

Several aggressive windowing algorithms have been proposed: Moving Time Windows [SoBW88], Filtered Rollback [LuWS89], Window based throttling [ReJe89], Unified Distributed Simulation System [McAf90], Bounded Time Warp [TuXu92], Aggressive

Global Windowing Algorithm [Dick93], and Breathing Time Warp [Stei93]. Most of these algorithms follow the same basic principle described above. The differences are in the methods used to determine the time window. Notable among these are UDS and Breathing Time Warp. The former uses independent windows for each LP. The latter uses a fixed size window based on the *number* of events rather than the logical time span to control the risk of each LP and a separate logical time window to control the aggressiveness of each LP.

As with non-aggressive windowing protocols, the primary limitation of aggressive windowing protocols is the determination of the window size, which can affect performance significantly. The window must be small enough to limit the propagation of incorrect computation but large enough to not limit the propagation of *correct* computation. A common window for all LP's could unnecessarily restrict some LP's. The optimal size will usually depend on the nature of the application and even the particular instance of it. In addition, the window size must change dynamically to account for changes in load distribution among the LP's. All of these requirements make the selection of the window size a difficult problem.

2.2.2 Space Based

Another technique limits optimism based on spatial boundaries rather than temporal boundaries. In general, the LP's are divided into *clusters*, each of which operate optimistically internally. Interaction among clusters is without risk (i.e. messages are exchanged only after it is determined they are safe). Erroneous computation is limited to a cluster so that when a rollback occurs, the length of the rollback chain will generally be smaller. Note, these protocols employ the *partitioning* design variable [Reyn88] in that the set of LP's is partitioned and the interaction of LP's within a partition is different from the interaction of LP's across partitions. Examples of this general technique are Hierarchical Rollback [Gima89] and the Local Time Warp [RaAT93] approach. A variation on this technique is used in Composite ELSA [ArSm92] where LP's are either conservative or optimistic. Knowledge is embedded in messages [Reyn88] to manage interaction between the two kinds of LP's.

The single main pitfall of this approach is that physical limits on the propagation of messages cannot guarantee the absence of severe cascading of rollbacks (for example, see

the CHASE workload in Section 6.3.3.2). While the lengths of rollback chains will be smaller in general, echoing is still possible.

Of more interest is a special class of space-limited aggressive protocols in which each cluster consists of a single LP. Here, each LP executes events aggressively but does not propagate the results of such execution to other LP's — i.e., interaction among LP's is without risk. Thus, rollbacks are local to each LP and echoing and cascading rollbacks are avoided. Examples of this technique are SRADS with local rollback [DiRe90], Speculative Computing [Mehl91], Breathing Time Buckets [Stei91] and Risk-free Time Warp [Bell93]. The main problem with this approach is that since risk is zero for each LP, the benefits of aggressive processing are limited only to individual LP's. Thus, this approach tends to be too conservative.

In summary, while some perceive the space-based technique as combining the best of the conservative and optimistic techniques, we believe it includes the pitfalls of both. This is due to two reasons: (i) risk-free interaction between clusters introduces the problems of conservative protocols: potential for deadlock, sensitivity to lookahead and overheads of communicating lack-of-dependence information, and (ii) aggressive processing introduces state saving and rollback costs. While rollback costs are very low in the special case of one LP per cluster, they can be very expensive in the more general case.

2.2.3 Penalty Based

In this approach, each LP's optimism is controlled by its recent behavior. Based (usually) on rollback behavior, some LP's are penalized (and consequently block) while others are favored (and consequently continue executing events). The key to good performance is to accurately predict the future based on its observed behavior. Three protocols have been proposed in this class. In Penalty-Based Throttling [ReJe89], the authors penalize any LP that sends an antimessage. This is injudicious, since an LP that is sending an antimessage is one that is rolling back and consequently, will be among those that are farthest behind in logical time. Adaptive Time Warp [BaHo90] also penalizes an LP that has been rolling back very often in the recent past. However, the penalty is incurred when the LP is processing forward rather than soon after a rollback. Predictably, Reiher and Jefferson report very poor performance while Ball and Hoyt have reported (slightly) better perfor-

mance than Time Warp. A third protocol is Randomized Self-synchronization [Madi93] in which each LP *estimates* the logical clock values of the other LP's. Based on the difference between these estimates and its own logical clock value, it either proceeds aggressively (if it is lagging behind) or suspends processing for some time (if it is far ahead). The assumptions that obtaining clock information from all LP's can be time-expensive and hence, LP's use a randomized algorithm to estimate other clocks. Obviously, the efficacy of this approach lies in the accuracy of these estimations.

In summary, penalty based limiting of optimism seems very promising. However, so far, it has been done based either solely on local state or on estimates of non-local states. We believe that, to achieve good performance LP's must dynamically react to changing system state, which is best done by actually obtaining the required non-local state and considering it along with local state.

2.2.4 Knowledge Based

The basic philosophy of this approach is to contain the propagation of incorrect computation as soon as it is determined to be incorrect. When an LP suffers a primary rollback, any aggressive processing it has propagated to other LP's is potentially incorrect. Therefore, it broadcasts a message to all affected LP's indicating the potentially erroneous nature of their execution. These LP's then limit their optimistic processing. Thus, this approach is based on rapid repair upon detection of an error rather than anticipation of errors. There are two examples in this category: Wolf [MaWM88], which takes a conservative approach and stops *all* potentially incorrect computation, and Filter [PrSu91], which stops only those computations that are exactly known to be incorrect. Availability of extra knowledge to identify incorrect computations exactly introduces high message processing costs (to store and process the extra information) and higher memory requirements. Further, to be effective, this scheme requires an efficient broadcast mechanism with the broadcast messages having higher priority than normal messages. Finally, the effectiveness of this technique is limited since it is reactive rather than proactive — by the time the primary rollback occurs, the incorrect computation may have proliferated so that rollbacks will be required to recover from the errors.

2.2.5 Probabilistic

The MIMDIX operating system [MaHF92] incorporates *probabilistic synchronization*. Here, a special process called a *genie* is responsible for the synchronization. Periodically, it sends a synchronization message to all LP's probabilistically, which cause them to synchronize to the timestamp of the message. In other words, those LP's that are ahead of this timestamp are rolled back. By keeping the timestamp of the synchronization message close to GVT, all LP's can be kept temporally close to each other, thus reducing the risk of cascading rollbacks. The main problem with this approach is that it does not restrict only incorrect computations. Also, it requires an efficient, high priority broadcasting mechanism. Finally, the probability controlling the synchronization must be estimated carefully so that, on the one hand, the overheads of synchronization do not dominate the simulation while on the other hand, LP's do not become too aggressive on the other. This estimation requires user expertise.

2.2.6 Discussion

Several of the approaches described above have demonstrated better performance than Time Warp with unlimited optimism under certain situations. This supports the intuition that limiting optimism reduces rollback costs and thus improves performance. However, we expect that all of these protocols will have limited performance in general due to one or more of the following: (i) the criterion for limiting optimism (window size, cluster size, penalty thresholds, probabilities, etc.) is predetermined (ii) the decision to limit optimism is based solely on local history (iii) the LP's operate synchronously.

2.3 State Based Protocols

Recently, three protocols have been proposed which we categorize as *state based* protocols. They differ from those listed above in two significant ways: first, they are adaptive in that the LP's continually adjust their optimism, and second, adaptive decisions are based on state information which, although available locally, is directly affected by the actions of other LP's. These protocols are similar to NPSI protocols. The first two protocols [HaTr94, FeTr94] are similar to each other in that they both utilize information about channels to decide when and for how long LP's should wait. Intuitively, it may be argued that our approach has an advantage over these because LP's in NPSI protocols can receive infor-

mation from all of their predecessors whereas channels provide information only about immediate predecessors. The benefit comes from the fact that in a channel-based protocol, information has to percolate through the predecessors of an LP before it reaches that LP. During this time, the LP may have moved farther ahead than it should have. Moreover, the protocols in [HaTr94, FeTr94] cannot determine if a predecessor is rolling back until they receive a null- or anti-message conveying that information. NPSI protocols are able to determine this information earlier. The third protocol in this class [DaFu94] uses memory consumption as the basis for limiting optimism. The protocol limits the memory consumption of LP's adaptively and consequently, also limits their optimism. It is based on a memory management protocol such as Cancelback or Artificial Rollback [LinY92a]. We take the opposite approach in that NPSI adaptive protocols limit the optimism of LP's directly and consequently limit their memory consumption. Our approach has the potential for completely eliminating the need for costly memory management schemes.

2.4 Summary

The optimistic approach is inherently more capable of extracting concurrency in a simulation than the conservative approach. However, uncontrolled optimism can degrade performance severely due to high rollback and memory management costs. Consequently, controlled optimism is a desirable feature. We have categorized protocols that limit optimism, based on the criterion for limiting optimism and reasoned about the pros and cons of each category. Three recent protocols (state-based) are of particular interest since they use state information to make adaptive decisions and are thus similar to the NPSI approach.

Chapter 3

Super-critical Speed

The search for a consistently efficient protocol is turning towards adaptiveness. An analysis of the optimality of (adaptive) protocols must begin by answering the question "What is optimal?" Our investigation of this question revealed two interesting points: first, due to the phenomenon of *super-critical speed*, there is no consensus on optimality of protocols, and second, some published results on super-critical speed are flawed. This chapter describes the contributions we have made to the understanding of super-critical speed and lower-bounds for PDES's. Specifically, (i) we present a correct sufficient condition for a protocol to be super-critical and show that a similar condition proposed previously is necessary but not sufficient, and (ii) we disprove the claim that super-critical speed requires independence among events and study the consequent implications on lower bounds. An understanding of these issues is essential to the successful application of critical path analysis to PDES. This work appears in [SrRe95a].

3.1 Overview

Critical path analysis is a technique to derive lower bounds on the completion time of PDES's [Livn85]. Berry and Jefferson [BeJe85] applied critical path analysis to PDES and argued that the critical path time is a lower bound on the completion time. This application of critical path analysis is particularly interesting because of the somewhat counter-intuitive result that it is possible for certain simulations to complete in less than the critical path time, a phenomenon we call *super-critical speed.* We say a protocol is super-critical if it is possible for at least one simulation using that protocol to complete in less than the critical time. In [Berr86, JeRe91] it was shown that certain variants of the Time Warp protocol [Jeff85] are super-critical. In particular, [JeRe91] presented a criterion for super-criticality and used it to show that four protocols were super-critical. These results showed that the critical path time is not a lower bound for all PDES's. Lin and Lazowska [LiLa91] proposed a lower bound that applies to all PDES's. However, this lower bound is a very loose one since it requires that each LP guess all of its computation correctly. These early analyses

defined inter-event dependence based on the timestamps of events and messages. This definition has the drawback that it creates artificial dependences between some pairs of events. Gunter [Gunt94a] proposed an enhanced definition of dependence and argued subsequently that independence is necessary for super-criticality. Also, he derived a new lower bound which is tighter than the one presented in [LiLa91].

Our contributions to this area of PDES research are two-fold:

i) We present a sufficient condition for a protocol to be super-critical. We show the condition presented in [JeRe91] to be necessary but not sufficient for super-criticality. The condition we present has been used in [Gunt94a] but was presented without formal support [Gunt94b]. In a result developed independently, we establish the truth of the sufficient condition.

ii) We show, by example, that Gunter's enhanced definition of dependence is not sufficient to capture all forms of super-criticality. Specifically, super-critical speed is also possible when an LP guesses correctly a dependence on a message that it has not yet received. Thus the claim that super-criticality requires independence is invalidated. We study the impact of this result on lower bounds derived using critical path analysis.

Critical path analysis is an active area of research in PDES. Tools based on such analysis [Lin93] are helpful in estimating whether sufficient parallelism exists in a simulation to justify the effort of parallelization. An understanding of the limitations of critical path analysis (as illustrated by insights described here) is important for successful application of these tools.

3.2 Critical Path Analysis

We assume the PDES consists of a set of logical processes (LP's), P_1, P_2 ... P_n and each LP executes on its own processor[1]. Each P_i represents a sequence of simulated events. In the case of aggressive protocols, critical path analysis applies only to committed events. The timestamp of an event e is denoted by $V(e)$. Each LP may send messages to other LP's

1. The lower bound may change if multiple LP's execute on a processor since more than one LP may have an executable event at the same time. The issue of optimal scheduling when multiple LP's are assigned to a single processor is addressed in [LinY92b].

as a result of executing events. In addition to simulation-specific information, a message contains a send-time which equals the logical clock of the LP sending the message and a receive-time which is greater than or equal to its send-time. When the message is received, the receiving LP schedules an event with timestamp equal to the receive-time of the message. This model is called the *message-initiating* model. With this model, we define the following relations on events:

i) Event *e* is the *predecessor* of event *e'* (or *e'* is the *successor* of *e*) if: (i) *e* and *e'* are executed by the same P_i, (ii) $V(e) < V(e')$, and (iii) there is no other event *e"* in P_i such that $V(e) < V(e") < V(e')$. We denote the predecessor of an event *e* as *pred*(*e*).

ii) Event *e* is the *antecedent* of event *e'* if the execution of *e* sends a message which schedules *e'*. Note *e* and *e'* may be executed by the same LP. We denote the antecedent of an event *e* as *ante*(*e*).

If defined, *pred*(*e*) and *ante*(*e*) are unique for a given event *e*. Also, an event can be the predecessor of at most one event but the antecedent of more than one event. We define the relation $\rightarrow$ as $e \rightarrow e'$ (*e* immediately affects *e'*) if either $e = pred(e')$ or $e = ante(e')$. The transitive closure $\Rightarrow$ of $\rightarrow$ induces a partial ordering on the events in the simulation as follows: $e \Rightarrow e'$ if there exists a sequence of events $e = e_{(0)}, e_{(1)}, \ldots e_{(n)} = e'$ such that $e_{(i)} \rightarrow e_{(i+1)}$ for all $0 \leq i < n$. Thus, in the general case, the relation $\Rightarrow$ defines the dependence among events: if $e \Rightarrow e'$, *then the execution of e' depends* on the execution of *e*. It is important to note this definition of dependence is based on the timestamps of events and not on application-specific relationships. If event *e'* is semantically dependent on event *e*, then we must have $V(e) < V(e')$. On the other hand, it is possible to have two semantically unrelated events *e* and *e'* such that $V(e) < V(e')$. Thus the $\Rightarrow$ relation is a pessimistic definition of dependence since it assumes dependences that may not exist.

A particular parallel discrete event simulation *run* (an execution of a simulation program with a given set of input values) may be represented using a *space-time* diagram as in Figure 6. The points in this 2-dimensional plane represent events that were executed in the simulation run. The two co-ordinates of each event are the LP at which it was executed (the space co-ordinate) and its logical timestamp (the time co-ordinate). Arrows are

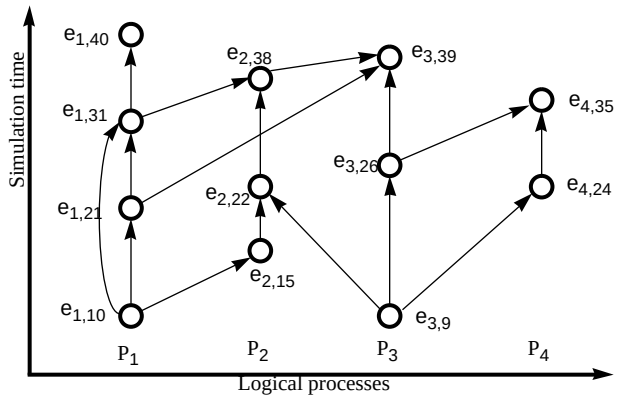


Figure 6 - Space-time diagram

used to represent the $\rightarrow$ relation among events. For example, $e_{1,10}$ is the predecessor of $e_{1,21}$ and the antecedent of $e_{2,15}$. Events that have no predecessors are called *initial* events. These are the first events to be executed at any LP. Further, initial events that have no antecedents (e.g., $e_{1,10}$ and $e_{3,9}$) are called *start* events.

With each event *e*, we associate an amount of real time required to execute that event, T(*e*). For the sake of simplicity, we ignore all overheads associated with inter-LP communication. These overheads can be incorporated easily into critical path analysis if required. If *start(e)* is defined to be the real time at which the execution of event *e* is started, then *complete(e)* = *start(e)* + T(*e*) is the time at which the execution of event *e* completes. The *critical time,* crit (*e*), of each event *e* is defined as follows:

$$\text{crit}(e) = \text{MAX}\{\text{crit}(\text{ante}(e)), \text{crit}(\text{pred}(e))\} + T(e)$$

where the simulation is assumed to start at real time zero and crit (ante (*e*)) and crit (pred (*e*)) are defined to be zero if ante (*e*) and pred (*e*) are not defined respectively. crit (*e*) is the earliest time the event *e* can complete execution under the assumption that no dependences are violated (i.e., if $e \Rightarrow e'$, *e* is executed before *e'*). Consequently, the largest value of crit (*e*) among all events in the simulation run will give us the lower bound on the completion time of the simulation run under the same assumption. Events that have this maximum critical time are called *final* events. An example of the computation of critical times is shown in Figure 7. The first number beside each event *e* is T(*e*) and the second number is crit (*e*).

A *critical path* is defined from a start event to a final event as follows:

i) Every final event is on a separate critical path;

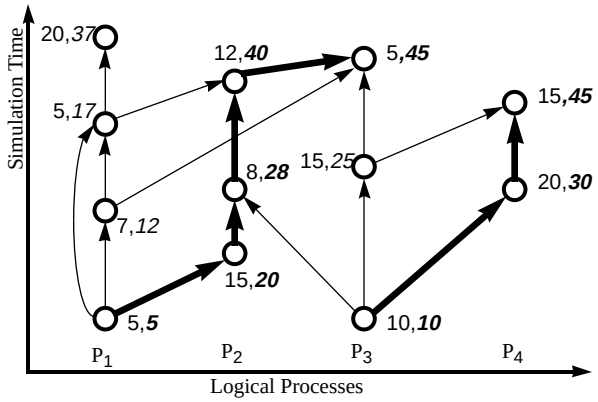


Figure 7 - Critical paths

ii) If *e* is on a critical path and crit (ante (*e*)) ≤ crit (pred (*e*)) then if pred (*e*) exists, it is on the critical path;

iii) If *e* is on a critical path and crit (pred (*e*)) ≤ crit (ante (*e*)) then if ante (*e*) exists, it is on the critical path.

We have highlighted the two critical paths in Figure 7. The maximum value of crit(*e*) is called the *critical path time*.

Self-Initiating Model

Another commonly adopted model for PDES's is the self-initiating model [Nico91] wherein each LP schedules its own next event — a message does not schedule an event. However, a message does affect the state of the receiving LP. Thus messages may cause rollbacks. It is easy to see that the definitions above apply to this model as well if we redefine the *antecedent* of an event as follows:

> Event *e* is the *antecedent* of event *e´* if the execution of *e* sends a message *m* such that *e´* is the event that is executed immediately after the arrival of *m*.

3.3 When is a Protocol Super-critical?

The critical path time defines a lower bound on the completion time of a simulation under the assumption that events are executed in the order specified by the dependence relation ⇒. However, it is only required that the overall effect of the simulation be the same as if the events were executed in that order. Therefore, if an LP guesses correctly, it may execute certain events out of order while keeping the simulation accurate. By guessing correctly on the critical path, it is possible to complete the simulation in less than the critical

path time. This phenomenon was called *super-critical speed-up* in [JeRe91] but we refer to it as *super-critical speed* since the critical path time is an absolute quantity. For a simulation run to be super-critical, the act of "guessing correctly" must occur on every critical path of the simulation.

It is important to note the distinction between a super-critical simulation run and a super-critical protocol. A super-critical run is a particular execution of a simulation that completes in super-critical time. A protocol is said to be super-critical if it is possible for a simulation run (at least one) using that protocol to be a super-critical run. By its very nature, a general PDES protocol cannot guarantee that LP's will guess correctly; it can only *enable* them to do so. Even so, it is desirable to be able to determine whether a protocol has this capability or not. To do so, we require a sufficient condition for super-criticality, **SC** such that if a protocol allows **SC** to be true in a simulation run, then the protocol permits the simulation run to be super-critical (i.e. the protocol is super-critical). We may then prove protocols to be super-critical by showing that they allow **SC** to be true in at least one simulation run. In [JeRe91], the authors present one such condition. We show that their condition is merely necessary (i.e. super-critical protocols will satisfy it) and we present the actual sufficient condition.

3.3.1 Condition for Enabling Super-critical Speed

Recall that for any event *e*, the following are defined:

- start (*e*): the real time at which the execution of *e* commences.
- complete (*e*): the real time at which the execution of *e* completes.
- crit (*e*): is the earliest time at which *e* can complete if no dependences are violated and is given by:

$$\text{crit}(e) = \text{MAX}\{\text{crit}(\text{ante}(e)), \text{crit}(\text{pred}(e))\} + T(e)$$

The super-criticality condition of [JeRe91], which we call **C**, is stated below:

C: There must be at least one pair of events *e* and *e'* on every critical path such that $e \rightarrow e'$ and complete (*e*) > start (*e'*)

We show by construction that the fact that **C** is true for a simulation run does not imply the run can complete in super-critical time.

Consider conceptually extracting a critical path and laying it on the real-time line as shown in Figure 8a. Each block of time (shaded rectangle) corresponds to the execution of an event on the critical path. Dots indicate that intermediate events have been left out for brevity. Now consider the critical path of a particular simulation run in which the condition **C** is satisfied. This condition could have been satisfied in one of the two ways shown in Figure 8. In Figure 8b, **C** is satisfied because the execution of $e_{(i+1)}$ commenced on time (at crit ($e_{(i)}$)) but the execution of $e_{(i)}$ completed late (after crit ($e_{(i)}$)). Note $e_{(i)}$ may have completed late due to several reasons. For instance, it may be that just prior to this execution of $e_{(i)}$, the LP executing $e_{(i)}$ may have guessed incorrectly and consequently rolled back, causing the delay of $e_{(i)}$. We have depicted this by a single delay (gap in the shading) at some point earlier than the execution of $e_{(i)}$. The LP executing $e_{(i+1)}$ guessed correctly so that the execution of $e_{(i+1)}$ was not rolled back when $e_{(i)}$ completed after crit ($e_{(i)}$). By only observing $e_{(i)}$ and $e_{(i+1)}$, the two events that satisfy **C**, we cannot conclude that the simulation can complete in super-critical time. In Figure 8c, **C** is satisfied because the execution of $e_{(i+1)}$ commenced early (before crit ($e_{(i)}$)) while the execution of $e_{(i)}$ was on time (completing at crit ($e_{(i)}$)). This occurs because the LP executing $e_{(i+1)}$ guessed correctly so that it is not

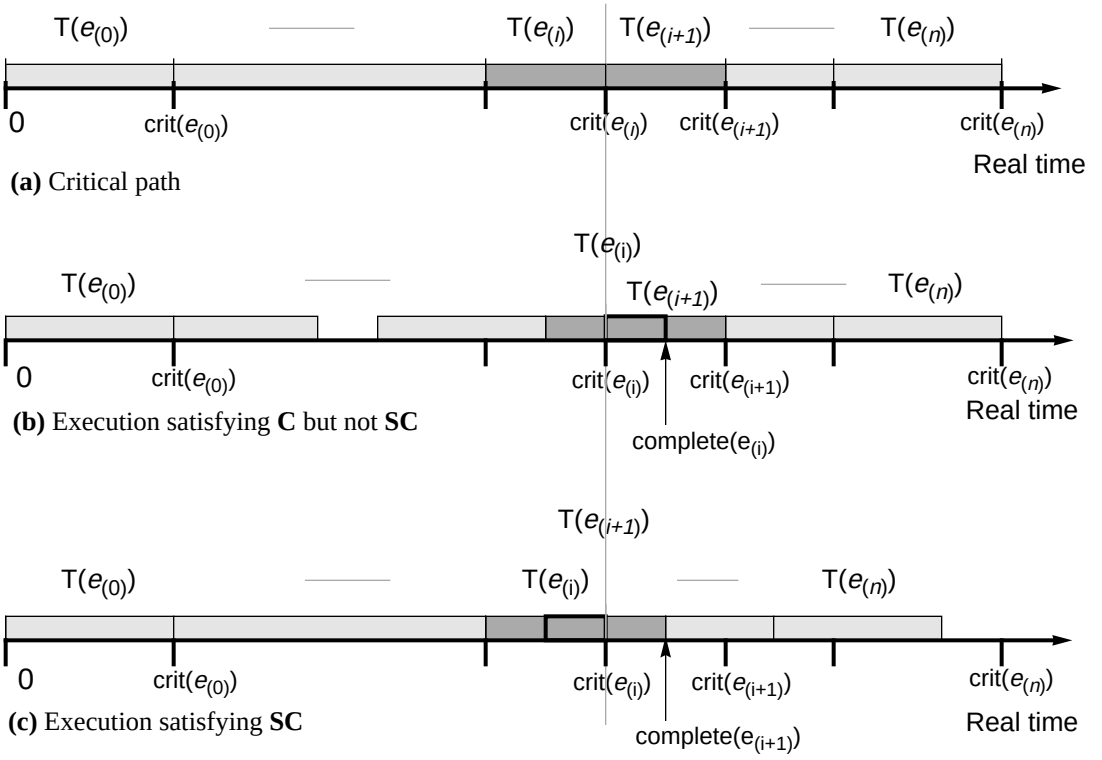


Figure 8 - Super-critical speed

rolled back when $e_{(i)}$ completes execution at crit ($e_{(i)}$). If every event on this critical path following $e_{(i+1)}$ is executed immediately after its preceding event completes (as depicted), the simulation will complete before crit ($e_{(n)}$), as shown (assuming this is the only critical path).

In [JeRe91], the authors consider only the scenario of Figure 8c and not that of Figure 8b. By itself, condition **C** specifies nothing about the absolute completion times of *e* and *e'*, which are essential to super-critical speed since the maximum critical time is an absolute quantity. Indeed, the scenarios of Figure 8b and Figure 8c were generated simply by moving events along the real time line while maintaining their relative timing relations. Clearly, crit (e) is the quantity that must link the condition for super-critical speed with the absolute timings of events since it defines whether an event is early, on time or late. Therefore, we claim that the actual sufficient condition for a protocol to be super-critical is:

SC: There must be at least one event *e* on every critical path such that complete (*e*) < crit (*e*)

Note, this condition is satisfied in Figure 8c (complete ($e_{(i+1)}$) < crit ($e_{(i+1)}$)) but not in Figure 8b, thus distinguishing the two. The intuition behind this condition is very simple: at least one event on every critical path must complete before its critical time so that there is the possibility of all events following it on the critical path to complete before their critical times and thus for the simulation to complete before the maximum critical time. The key is that crit (*e*) (and not the starting or completion time of event *e* relative to its adjacent events on the critical path) defines the earliness or tardiness of event *e*. Thus we have shown by construction that the fact that a simulation run satisfies **C** does not imply that it can complete in super-critical time whereas the fact that a simulation run satisfies **SC** does imply that it can complete in super-critical time. Of course, even in the situation of Figure 8c, it is possible for the simulation to complete after crit ($e_{(n)}$), if at least one event following $e_{(i+1)}$ is sufficiently tardy.

If a protocol is super-critical, then by definition, there exists a simulation run using that protocol such that the final events on every critical path complete before their critical times. Trivially therefore, super-criticality implies that the protocol allows **SC** to hold for at least one simulation run. Thus **SC** is also a necessary condition for super-criticality.

Revisiting Figure 8c, if **SC** is true on a particular critical path, we can show by contradiction that **C** is true for $\hat{e}$ and e where e is the earliest event for which **SC** is true on the critical path and $\hat{e} \rightarrow e$. Thus, **SC** $\Rightarrow$ **C**. Since **SC** is necessary for super-criticality, **C** is also a necessary condition for a super-critical protocol.

In [JeRe91], the authors use **C** to show that the four protocols: Time Warp with Lazy Cancellation, Time Warp with Lazy Reevaluation, Time Warp with Phase Decomposition and Space-Time Simulation are capable of super-critical speed. Elsewhere [SrRe93b] we have shown that all of these protocols satisfy **SC** and therefore are indeed, super-critical.

Finally, a common misconception seems to be that a protocol that allows LP's to guess events is super-critical. A distinction must be made between all events and the ones that are actually committed. Time Warp with aggressive cancellation allows LP's to guess events but does not commit events that complete before their critical times (i.e. it does not allow LP's to guess committed events). Accordingly, it has been established [LiLa91] that Time Warp with aggressive cancellation is not super-critical. In words, this is what **C** claims: if a protocol allows LP's to guess committed events correctly, then it is super-critical. However, that is not sufficient. We have shown that the protocol must allow LP's to guess committed events correctly *and* before their critical times.

3.4 Super-criticality and Independence

Observation: There are two phenomena that may result in super-critical speed:

i) when an LP correctly guesses that no messages will arrive that affect an event;

ii) when an LP correctly guesses the effect on an event, of one or more messages that *will* arrive;

In the first case, super-critical speed is possible because dependence (the relation $\Rightarrow$ in Section 3.2) is defined based on timestamps of events and messages. Under this definition, an event e at some P_i is dependent on a message m if m schedules an event e' at P_i such that $V(e') < V(e)$. However, it is possible that the execution of e' affects only a part of the state of P_i upon which the execution of e does not depend, i.e., the outcome of executing e is independent of whether e' is executed or not. Recognizing this intra-process concurrency, Gunter [Gunt94a] formalizes the notion of independence of states and messages on events, which we reproduce here for convenience:

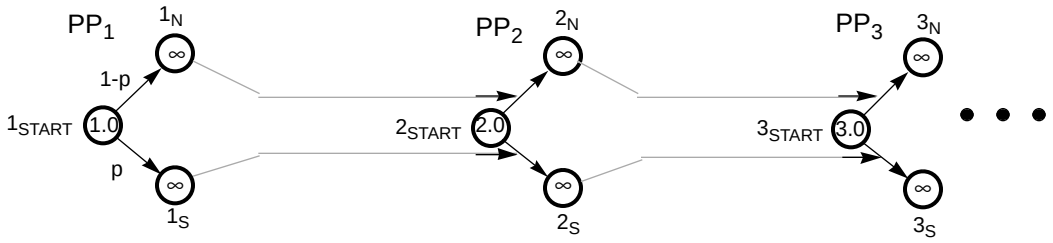


Figure 9 - A super-critical simulation without independence

Definition: A state (message), denoted S (*m*), created by event *e* is said to be independent of a set of events **E** if **E** is a subset of the set of all events **E'** such that *e* ⇒ **E'** and the S (*m*) resulting when all of the events in the simulation are properly executed is the same as the S (*m*) when all of the events in **E** are not executed.

In particular, a message *m* is independent of an event *e* if the same message *m* is generated irrespective of whether *e* is executed or not. It follows that a message *m* is dependent on an event *e* if *m* is not independent of *e*. Using this enhanced definition of dependence, Gunter derives two results: (i) independence is necessary for super-critical speed, and (ii) a new definition of the critical path and hence a new lower bound which is tighter than that of [LiLa91]. By capturing semantic dependences accurately, this new definition of dependence accounts for super-critical speed that occurs when LP's guess only the *lack* of dependence. However, it fails to capture super-critical speed that may occur when LP's guess the *existence* of dependence.

To understand how super-critical speed may occur when LP's guess the existence of dependence correctly, consider the physical system shown in Figure 9. The system consists of *n* physical processes, PP_i, each of which begins in a starting state i_{START}. At time 1.0, PP_1 transitions to state 1_S with probability *p* or to state 1_N with probability 1-*p*. At time 2.0, PP_2 transitions to state 2_N if PP_1 has transitioned to state 1_N earlier or to state 2_S if PP_1 has transitioned to state 1_S earlier. Note there is no default action for PP_2: it knows that PP_1 has transitioned earlier. Similarly, at time 3.0, PP_3 transitions to state 3_N if PP_2 has transitioned to state 2_N earlier or to state 3_S if PP_2 has transitioned to state 2_S earlier. The remaining processes also behave similarly. Pictorially, the dependences among the transitions are shown by the dashed lines in Figure 9.

Assume this system is simulated by *n* logical processes, P_i, one for each PP_i. After a state transition, each P_i (except P_n) sends a single message m_i to P_{i+1} indicating the direc-

tion in which it has transitioned (North or South). Recalling the definition of dependence above, clearly each m_i is dependent on m_{i-1} and the event that caused m_{i-1}. Now consider an execution of this simulation using Time Warp with Lazy Cancellation. Each P_i, $(2 \leq i \leq n)$ has an event which will cause it to transition out of i_{START}. Since these P_i's have not yet received the transition messages (m_i) from their predecessors, they have to guess the next state to which they must transition. Let us assume that each P_i decides to transition to state i_S and generates m_i informing its successor of the transition. At the same time, P_1 executes its only event, decides (probabilistically) to transition to state 1_S and sends m_1 accordingly. Since every P_i has guessed its incoming message correctly, no antimessages are generated (because of Lazy Cancellation) and the simulation is complete when all m_i have been received. The entire simulation run takes O(1) time whereas, under the new definition of dependence, the critical path time is O(n). Thus, this simulation run has completed in super-critical time. In this example, the super-critical speed comes from processes P_2 through P_n. The events they execute and the messages they generate are all dependent on other events and messages (in other words, there are no independent messages or states). Thus, Gunter's claim that super-critical speed requires independence is disproved.

It is natural to ask if the example we have presented is realistic (i.e. has any practical counterparts). We believe it is because one can imagine a simulation where LP's statistically estimate the probability of transitioning along a particular arc (p in Figure 9). If p is high and an LP may guess that the next transition will be along that arc. Finally, our example also serves to demonstrate that while super-criticality is possible, it may not be observable in practice because it may occur only in brief phases during the simulation but not for the entire simulation. Specifically, if only a subset of the P_i $(2 \leq i \leq n)$ guess their computation correctly, the rollbacks induced could cause the simulation to take longer than the critical path time even though that subset of the processes completed their execution in super-critical time.

3.5 Lower Bounds

From our example, it is evident there are simulations where Time Warp with Lazy Cancellation may complete in super-critical time, even with the new definition of the critical path in [Gunt94a]. It follows that the lower bound derived in [Gunt94a] based on this

critical path is *not* a lower bound for Time Warp with Lazy Cancellation (in fact for any super-critical protocol). Our insight suggests that irrespective of how accurately we capture the semantic dependence among events, protocols such as Time Warp with Lazy Cancellation (and others listed in [JeRe91]) still have the capability to complete in super-critical time. Under the best circumstances, each LP can guess all of its dependences correctly (as is the case in our example above), completing its execution in an amount of time equal to the sum of the execution times of all of its events. We are thus led to the conclusion that the lower bound of [LiLa91] (which was stated as a lower bound only for Time Warp with Lazy Cancellation but applies to all protocols):

$$\mathrm{LB} = MAX_{\mathrm{P}_i}\Big(\sum_{e \in \mathbf{E}_{\mathrm{P}_i}} \mathrm{T}(e)\Big)$$

where $\mathbf{E}_{\mathrm{P}_i}$ is the set of events executed by $\mathrm{P_i}$, remains the best known general lower bound for all PDES's. Moreover, it suggests that critical path analysis will not be able to improve upon this general lower bound. However, this lower bound is of limited significance because it is unachievable for any realistic simulation. Consequently, it is important to note that Gunter's lower bound, which is a tighter one, applies to *particular simulation runs* using protocols in which LP's do not guess the existence of dependence. It is difficult to incorporate this restriction into the protocol itself, because the nature of an LP's guessing depends on the application and is generally not specified in a protocol.

3.6 Summary

A protocol is said to be super-critical if there exists at least one simulation that *can* complete in less than the critical path time using that protocol. Since several protocols have been shown to be super-critical, a sufficient condition for super-criticality is desired which may be used to determine whether protocols of interest are super-critical or not. We have established such a sufficient condition. Also, we have shown that a condition used in a previous study to demonstrate the super-criticality of four protocols is not sufficient but necessary. Our observation that super-critical speed is possible when LP's correctly guess the dependence of some events on unreceived messages disproves a previous claim that super-critical speed requires independence. We have described the implications of this contradiction on lower bounds on completion times of simulations.

Chapter 4

NPSI Adaptive Protocols

Traditional solutions to the synchronization problem in PDES have, in large part, failed because they cannot adapt under circumstances in which they should. Currently, the direction offering the most promise is one in which the optimism of LP's is controlled adaptively. We describe a new and unique class of protocols - those in which LP's dynamically adapt their behavior to changes in the simulation using low-cost near-perfect system state information. We call these *NPSI (near-perfect state information) adaptive protocols*. The work described here appears in [SrRe95b].

4.1 Motivation

We seek a consistently efficient protocol — one that can be shown to perform well over a wide range of applications. As noted in Chapter 2, an optimistic approach is inherently more capable of extracting the parallelism inherent in simulations and thus offers the most potential towards our goal. However, optimistic protocols are not without their problems. These problems can be alleviated by controlling optimism (as demonstrated by previous efforts described in Chapter 2). NPSI adaptive protocols are optimistic protocols in which optimism is controlled based on near-perfect state information.

4.1.1 Effects of Controlling Optimism

Optimistic protocols incur three time costs: state saving, rollback and memory management. Limiting optimism (aggressiveness and risk) introduces a fourth cost, *lost opportunity cost*, characterized by the potential loss in performance when an LP stops executing events or sending messages even though it is safe for it to continue. The time cost of state saving is a function of the size of the state that is modified by events and the frequency of state saving, rather than the level of optimism and thus can be ignored here. To obtain good performance, protocols that control optimism must minimize the following cost function:

```
total cost = rollback cost +
             memory management cost +
             lost opportunity cost
```

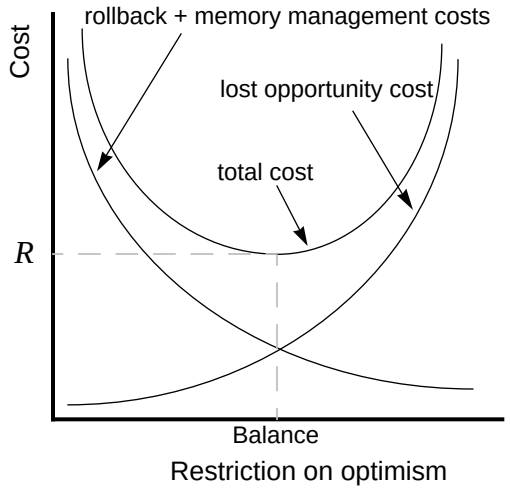


Figure 10 - Trade-off introduced by controlling optimism

While limiting optimism tends to decrease the first two costs, it also tends to increase the third, leading to the trade-off shown in Figure 10. Protocols that control optimism must attempt to achieve a good balance in this trade-off. The *best possible performance* for protocols with controlled optimism is characterized by $R = 0$, where R is the residual total cost when the balance is reached. $R = 0$ when controlled optimism eliminates rollback and memory management costs without adding lost opportunity cost.

4.1.2 Measure of Success

From the discussion above, a good measure of success for protocols that control optimism is how significantly they reduce R, the residual cost at the balance point in Figure 10. Yet another measure is their performance relative to that of a protocol that does *not* control optimism, such as Time Warp. Time Warp is the protocol that has been applied to the most applications with encouraging results [Fuji93a]. However, it suffers the drawbacks of optimistic protocols noted earlier. Consequently, there are several applications for which Time Warp does not perform well. By showing that NPSI protocols consistently outperform Time Warp over a wide range of workloads that includes those for which Time Warp performs well and those for which it does not, we will have demonstrated consistent good performance.

4.2 Near-Perfect State Information

The trade-off of Figure 10 must be balanced properly in order to obtain good performance. To do so, protocols must attempt to distinguish incorrect computations from correct ones and limit the propagation of the former while allowing the latter to proceed. PDES's are known to be very dynamic in nature [NiRe90, Fuji90], i.e. the locality of events in the system changes as the simulation progresses. Typically, this is due to the fact that simulated systems have some form of information flow which is translated in the simulator into a causal chain of events among LP's. Since the propagation of such chains is based on probabilistic decisions and input parameters of the simulation, it is impossible to determine the flows a priori (except in special cases). Thus, the synchronization requirements of PDES's are usually irregular, highly data-dependent and dynamic.

We believe the two key requirements for a protocol to be consistently efficient are that it is adaptive and that it uses feedback from the simulation to control optimism. Ideally, these requirements are satisfied by providing LP's with perfect state information, such that any relevant change in system state is visible instantaneously. However, this is impossible to achieve in practice due to various latencies in computing distributed snapshots. We believe, and this work supports our belief, that a good approximation of perfect state information is desired. We take the direct approach by assuming the existence of an asynchronous dynamic *feedback system*. This system operates asynchronously with respect to the LP's and provides them with near-perfect information at low cost. We describe the computational model for the feedback system, and discuss how it may be realized in practice.

4.2.1 Reduction Model

The assumed model of computation, which we call the *reduction model* is shown in Figure 11. The LP's exchange timestamped simulation messages through the host communication network. In addition, each LP encodes the relevant parts of its state into a set of values called an *input state vector* (ISV). These ISV's are processed by an NPSI Calculator to produce *output state vectors* (OSV) for each LP as below:

> Let $\mathbf{ISV}_1$, $\mathbf{ISV}_2$, ... $\mathbf{ISV}_n$ be the input state vectors submitted by the n LP's such that $\mathbf{ISV}_i = \langle V_i^1, V_i^2, ..., V_i^m \rangle$ where m is the size of each LP's state vector. Let θ_k, $1 \le k \le m$ be binary associative operators. Then for each LP_i the NPSI Cal-

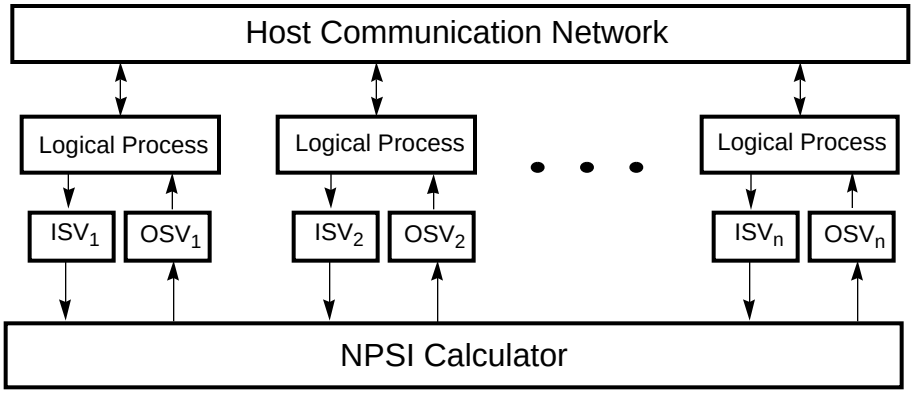


Figure 11 - Model of computation for NPSI adaptive protocols

culator computes $\mathbf{OSV}_i$ = <TS_i^1, TS_i^2, ..., TS_i^m> where $TS_i^k := \theta_k<V_l^k>$ $\forall l \in P_i$, $1 \le k \le m$ and P_i is the predecessor set of LP_i.

The predecessor set of LP_i is defined as follows. A *communication graph* describes the communication among LP's. The nodes of this graph are the LP's. A directed arc exists from LP_a to LP_b if LP_a can send a message to LP_b. The predecessor set of LP_i is the set of LP's, LP_j such that there exists a path from LP_j to LP_i in the communication graph. Note, a cycle containing LP_i implies LP_i is a predecessor of itself. Since the predecessor set can be different for different LP's, the information computed by the NPSIC is called *target-specific information* [PaRe93]. In the special case where the communication graph is strongly connected (i.e. there exists a path between every ordered pair of nodes), the computed information corresponds to global information. We note that the above definition of the predecessor set is pessimistic in that an arc exists from LP_a to LP_b in the communication graph if there is a possibility that LP_a sends a message to LP_b — in actual fact, it may be that LP_a never sends a message to LP_b. The predecessor set is only one possible set over which to compute target-specific information — it is conceivable that target-specific information is computed over some subset of the predecessor set.

The LP's and the NPSI Calculator (NPSIC) are completely asynchronous with respect to each other. State vectors are used to force the interactions between them to be atomic: an LP must provide a *complete* ISV and must read a *complete* OSV. However, it is not required that every LP_i reads every OSV_i generated by the NPSIC. We have established elsewhere [RePS93] that the atomicity criterion is sufficient to provide sequential consis-

tency, a property that facilitates the design of provably correct synchronization algorithms using this model (see Chapter 5).

An LP may generate a new ISV at any time. Similarly, it may read its OSV at any time. The NPSIC operates in *reduction cycles*, reducing a set of state vectors, one from each processor, to a set of output state vectors, one for each LP, in each cycle. Subsequent reduction cycles re-use state vectors for those LP's that do not present new state vectors. It is assumed that the time required for each reduction cycle is neglibly small when compared with the time required for other operations in the simulation such as event execution, state saving, message sending, rollback, etc. Thus, this system provides almost instantaneous target-specific snapshots of the system in a reduced form at very low cost to the LP's. We assume the reduction model because we believe it is sufficiently powerful to provide the kinds of information required in most PDES's [RePS92]. Chapter 5 describes an algorithm we have designed to compute global virtual time [Jeff85] rapidly, accurately and frequently in an aggressive PDES, using the reduction model.

4.2.2 Practical Issues

The reduction model can be realized in practice. On a shared memory architecture, a set of processes may be dedicated to the task of computing OSV's using ISV's stored in shared memory that are updated by other processes participating in the simulation. The OSV's can then be read back from the shared memory. Lin [LinJ92] describes techniques to compute target-specific information for use with conservative protocols. These techniques can be extended to aggressive protocols. Another example of asynchronous computation using non-local information is the GVT algorithm of [FuHy95]. An implementation of this algorithm on a KSR-2 shared-memory multiprocessor takes on the order of 100 microseconds for each GVT computation with very little computation overhead (1 microsecond) [Hybi95]. These performance results indicate that it is possible to compute NPSI on shared-memory multiprocessors.

One way to realize the reduction model on distributed memory architectures is to use a high-speed asynchronous reduction network such as the *parallel reduction network* (PRN) described in Section 5.5.1. The PRN is a global reduction network, one in which binary, associative operations (minimum, summation, etc.) are used to reduce state infor-

mation from all LP's. Our experience with the design, construction and testing of the PRN [RePS92] suggests that a production version of such a network can operate at very high speeds (less than 20 nanoseconds per stage in the tree). These low latencies, combined with its pipelined, tree structure make such a network scalable up to thousands of processors. We have used the PRN to implement and study the NPSI algorithms proposed here. Chapter 6 presents the results of this performance analysis.

It is important to note that reduction networks have been proposed and constructed in practice, to support global computations such as barrier synchronization, summation, determining maxima and parallel prefix computation [Ston90, Hosh89, CrKn28, Blel89]. One such network is the *control network* in the CM-5 [Ponn93]. It is used to perform non-local data distribution operations such as broadcasting, combining (reduction and parallel prefix), bit-wise operations and barrier synchronizations, very rapidly. For bit-wise logical OR operations, it can operate asynchronously with respect to the processors, much as our PRN. Thus, the control network appears to be a suitable platform on which to implement the reduction model. Similarly, the Cray T3D [Cray93] includes support circuitry in the form of a tree to perform rapid barrier synchronization. Although this tree is incapable of performing general reductions, the T3D supports several reduction operations across processors, in software [Cray95]. This is made possible by the combination of a distributed shared memory and very low latency access to this shared memory. These reduction functions may be used to implement the reduction model.

A global reduction network implements a special case of the general reduction model in which all of the OSV's are identical and are computed using information from all ISV's. The general model can be implemented using a *target-specific reduction network* — one that computes OSV_i using a subset of all ISV_i's where the subsets can be different for the various LP's. While a target-specific reduction network has not been constructed as yet, the results of [Panc94] indicate that it is possible to do so.

4.3 NPSI Adaptive Protocols

NPSI adaptive protocols are optimistic protocols that control the aggressiveness and risk of LP's dynamically using near-perfect state information. There are two phases in the design of NPSI adaptive protocols:

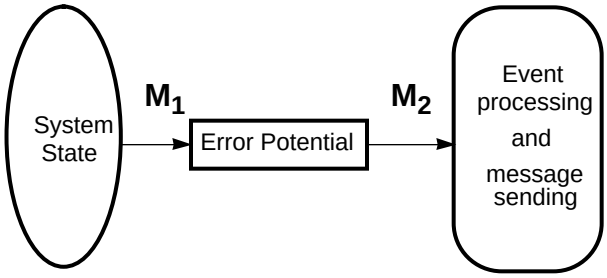


Figure 12 - General framework for adaptive protocols

- identifying the information on which the decision to limit optimism is to be based;
- designing the mechanism that translates this information into control over an LP's optimism;

Clearly, there are numerous choices for each of these. In order to facilitate independent study of each, we uncouple them by introducing a quantity called *error potential* (EP_i) associated with each LP_i. The value of EP_i is used to control LP_i's optimism. The framework we propose is shown in Figure 12. The NPSI adaptive protocol keeps each EP_i up-to-date as the simulation progresses, by evaluating M_1 at high frequency using state information it receives from the feedback system. Similarly, M_2 dynamically translates these new values of EP_i into delays in the event execution and communication rates. Different protocols may be devised by changing M_1 and M_2. The goal of our research is to design mappings M_1 and M_2 such that their combination forms an adaptive protocol that performs well consistently.

4.3.1 M_1 - Computing EP

To achieve optimal performance, LP's must be able to identify computations that will be rendered incorrect in the future and limit their propagation. Obviously, this requires the ability to predict the future which is difficult at best. In the framework described above, the error potential is a way of labeling computation as *potentially incorrect*. EP_i is a number that indicates the likelihood of LP_i's computation becoming incorrect in the near future: the higher the value of EP_i, the higher this likelihood. The key to consistently good performance is to devise an M_1 that will predict the nature of the LP's computation (i.e. whether that computation will be rolled back or not) accurately most of the time. An inaccurate M_1

can produce a low value of EP when the computation is erroneous, resulting in higher roll-back costs, or a high value of EP when the computation is correct, resulting in higher lost opportunity cost.

While designing M_1, we must keep in mind the reduction model of Section 4.2.1. This model requires that we encode the relevant parts of an LP's state into a set of values (a state vector). We list some possible mappings for computing EP_i below. In the following, GVT refers to *global virtual time* [Jeff85] defined as the minimum of the logical clocks of all LP's and the timestamps of any messages that have been sent but not yet incorporated into the events list of the receiving LP's.

i) EP_i = number of local events executed with timestamps > GVT. This mapping incorporates two concepts: (a) the cost of a rollback is generally proportional to the number of events rolled back, and (b) no LP can roll back to a logical time smaller than GVT. Thus, the EP will be proportional to the maximum possible rollback depth. Unfortunately, this mapping does not capture the *likelihood* of an LP rolling back. For instance, consider two LP's, one at logical time 100 having executed 2 events with timestamps > GVT and another at logical time 50 having executed 10 events with timestamps > GVT. This mapping will restrict the latter even though the former is more likely to be rolled back. The situation is further deteriorated if the second LP is about to send a message to the first. This reasoning suggests that logical time should be included in the mapping. We note that this mapping is equivalent to the Breathing Time Buckets algorithm [Stei93].

ii) $EP_i = MT_i$ - minimum MT among all LP's, where we define MT_i (*minimum time)* as the smaller of (i) the logical clock of LP_i and (ii) the minimum timestamp of all messages sent by LP_i which are either in transit or have not yet been processed by the receivers. Note, the minimum of the MT's of all LP's is GVT. The rationale is that if an LP is far ahead of others in logical time and may receive messages from them, it is quite likely to be rolled back. This mapping remedies the problem described in (i) above. It requires that GVT be computed quickly and often as is done by the algorithm described in Chapter 5.

iii) EP_i = logical clock of LP_i - GVT. Since the logical clock of an LP_i reflects its current execution state more accurately than its MT_i and in fact, may be much larger, mapping (ii) above may not penalize some LP's sufficiently. Thus, this mapping is expected to perform better than mapping (ii).

iv) A further improvement over mapping (iii) would be to use the *next event time* (η_i) of an LP_i rather than its logical clock, i.e. $EP_i = \eta_i$ - GVT. It is reasonable to expect this change to improve performance since η_i is a predictor of LP_i's next action while LP_i's logical clock only describes its current state. Unfortunately, the LP with minimum η_i cannot determine the fact that its next event is safe (when there are no messages in transit) and therefore will continue to wait for an amount of time proportional to its EP_i (which can be arbitrarily long) even though it can proceed. Thus, this mapping has the disadvantage that it can lead to unpredictably long periods of unnecessary waiting (lost opportunity).

v) A general optimization for all of the above is to use *target-specific virtual time (TSVT)* [PaRe93] rather than GVT. $TSVT_i$ for an LP_i is the GVT computed over only the predecessor set of LP_i in the communication graph (Section 4.2.1). Since the event or message with timestamp equal to GVT may be one that cannot affect LP_i's execution, $TSVT_i$ is a tighter lower bound on the rollback horizon for LP_i than the global GVT. This principle can be extended to other types of information as well.

vi) EP_i = logical clock of LP_i - the mean of the MT's of all LP's. If it is observed that $TSVT_i$ is usually too pessimistic (i.e. LP_i generally does not roll back that far), this variant may yield better performance.

vii) EP_i = logical clock of LP_i - MIN_i (MT_j + cumulative lookahead from LP_j to LP_i), where the minimum is computed over all LP_j that are predecessors of LP_i. It has been clearly established that performance and lookahead in simulations are positively correlated [Fuji90]. Here, we enhance mapping (iii) by incorporating lookahead.

viii) This mapping is based on the view that the likelihood of an LP rolling back correlates with whether its immediate predecessors are rolling back. Define MP_i to be the minimum of the logical clocks of those LP's that are immediate predecessors of LP_i and are rolling back (note, the logical clock values are observed *after* the rollback is initiated, i.e. after the clock has been rolled back). Then, $EP_i = \infty$ if $MP_i \geq$ logical clock of LP_i; else $EP_i = 0$. Thus, this mapping restricts an LP_i while any of its predecessors roll back to a time smaller than LP_i's current logical clock. While this mapping will reduce secondary rollbacks (i.e. those caused by antimessages), it ignores primary rollbacks (caused by stragglers).

ix) $EP_i = MAX_i$ {memory usage of LP_j}, where the maximum is computed over all LP_j to which LP_i can send a message directly. The mapping is designed to prevent a situation in which an LP_i floods a successor LP_j with messages due to uncontrolled aggressive processing, thus causing LP_j to block due to insufficient memory. Time expensive memory management schemes such as Cancelback [LinY92a] are currently used to repair such situations. With this mapping, a throttling effect is created when memory is scarce and removed when memory becomes available again.

x) EP_i is computed based on the rollback history of LP_i. Note, this mapping is unique in that EP_i is based solely on local state information. By keeping some history information at each LP, it may be possible to compute the potential for an error in the near future based on the recent and distant past. Some form of filter must be devised which associates different weights with the recent and distant pasts, giving more weight to the recent past. Of course, the implicit assumption here is that local history is a good predictor of the local future.

Note, each of the mappings proposed above is dynamic in the sense that the mapping is continuously evaluated as the simulation proceeds, thus updating the value of EP_i. In order to design a consistently efficient protocol, we must devise an M_1 that estimates EP_i reliably. However, any one of the above mappings by itself may not be able to do so: we can construct a scenario for each wherein the mapping will unnecessarily restrict some

LP's. For instance, an LP that is only a sender of messages but not a receiver, will be throttled unnecessarily by any M_1 that is based on differences in logical times or event execution rates. Theoretically, such an LP need never be throttled since it can never roll back. Practically, it must be throttled since, otherwise it may cause its successors to consume all available memory to store the messages it sends and consequently block. Such LP's must use mapping (ix). More generally, the LP's may be partitioned into subsets with a different M_1 being applied to each subset. Further, the choice of M_1 itself may be made dynamic. It is also possible to change the grouping of LP's dynamically. Finally, instead of using a single mapping for each subset, we may employ a weighted combination of several mappings. This is an important advantage of the general framework we have proposed: we are able to combine multiple approaches into a single protocol.

So far, we have assumed that the error potential is associated with the LP's (i.e., EP_i for LP_i). Another possibility is to associate error potentials with messages. Typically, a message causes an event to be scheduled, the execution of which results in a new message being sent out. Thus, we can identify *causal chains* in the simulation and associate error potentials with them. This may allow finer control of optimism.

4.3.2 M_2 - Controlling Aggressiveness and Risk

M_2 is a function that translates a value of EP into control over the aggressiveness and risk of an LP. It must be designed such that higher values of EP_i result in less aggressiveness and/or risk at LP_i. A simple scheme is to establish a *threshold* such that if the value of EP exceeds this threshold, event execution and communications are suspended until EP falls below the threshold again. A more sophisticated scheme would reduce the event execution and communication rates gradually as EP increases. This deceleration can be achieved by inserting delays at appropriate points. M_2 would then be a function that maps EP to a wall-clock time delay.

An LP that sends out messages to other LP's based on aggressive computation is said to have risk. Since risk contributes to rollback costs, some mechanism must be included to control an LP's risk also, based on its error potential. This mechanism could be similar to the one used to control the LP's aggressiveness: an LP could either stop sending messages while EP exceeds a threshold value or it could gradually decrease the rate at

which it sends out messages as EP increases. It is also possible to have a common mechanism controlling both aggressiveness and risk. By controlling the risk of LP's, we can generate any hybrid scheme between the two extremes: an approach with unlimited risk such as Time Warp, and a risk-free approach such as SRADS with Local Rollback [DiRe90].

4.4 Summary

We have introduced a new class of adaptive protocols, the NPSI adaptive protocols. These are unique in that they employ near-perfect state information in dynamically controlling the aggressiveness and risk of LP's. We have motivated the NPSI approach, described the reduction-based computational model assumed by it and discussed how this model may be realized in practice. We have described a framework for designing NPSI adaptive protocols, consisting of two mappings, M_1 and M_2. Finally, we have explored design issues regarding these mappings.

Chapter 5

Near-Perfect GVT

The reduction model assumed for computing near-perfect state information has been introduced earlier. We describe how this model may be used to produce state information required by adaptive protocols. Specifically, we present an algorithm (called *npGVT*) to compute GVT in an aggressive PDES. Since *npGVT* is based on the reduction model, it produces very accurate (near-perfect) values of GVT at low cost. Thus, it can be invoked frequently during the simulation. These features make *npGVT* amenable for use in adaptive protocols (other approaches are not). *npGVT* serves as a base for implementing NPSI adaptive protocols and is used to compute the mapping M_1 for ETA, the NPSI adaptive protocol described in Chapter 6. We describe *npGVT2P*, an implementation of *npGVT* on a global reduction network. We prove the correctness of both algorithms. *npGVT2P* has been published in [SrRe93a].

5.1 Why GVT?

Global virtual time (GVT) is an important value required by the LP's in any aggressive protocol. GVT represents the simulated time up to which all LP's have simulated correctly and beyond which, all LP's have simulated speculatively (i.e., may be incorrect). This means that no LP can ever be rolled back to a time smaller than the value of GVT. Thus GVT defines a commitment horizon, i.e., when GVT is greater than or equal to the timestamp of an event, that event cannot be rolled back again and therefore may be *committed*. Consequently, GVT is critical in interactive and real-time simulations. GVT is also used for several other purposes in aggressive PDES's. The most important of these is fossil collection. Since LP's are aggressive, they must save state periodically. However, all states with timestamps smaller than GVT may be reclaimed since they are "unreachable". Other uses of GVT include termination detection and gathering statistics. Finally, GVT (and its target-specific counterpart, TSVT) appears to be a useful value on which to base adaptive decisions to limit optimism (Section 4.3.1).

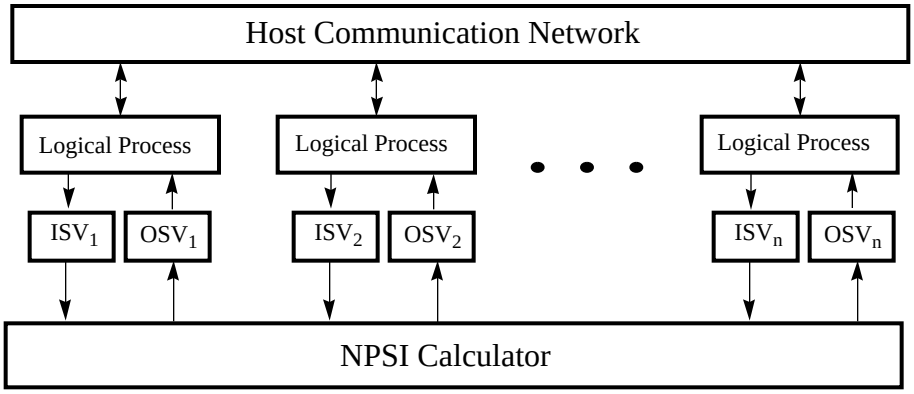


Figure 13 - Model of computation for adaptive protocols

Since computing GVT corresponds to obtaining a distributed global snapshot of the system, it is a non-trivial problem. Several GVT computation algorithms have been proposed. However, most of these are time consuming, which has two consequences: (i) the computed values of GVT tend to be "stale" (i.e. not close to the actual value at the time of determination), and (ii) the cost of the algorithm precludes frequent invocations. These two factors make such approaches infeasible for use in adaptive protocols. We describe *npGVT*, our GVT computation algorithm that, being based on the reduction model of Section 4.2.1, produces accurate values of GVT at low cost.

5.2 GVT Computation Algorithm

For exposition, we describe *npGVT* as applied to the Time Warp protocol — it is equally applicable to *any* aggressive protocol. We assume familiarity with Time Warp and associated terminology — Appendix II provides an overview. Recall the reduction model of Section 4.2.1, repeated in Figure 13 for convenience. For the present, let us assume that this model computes global reductions rather than target-specific reductions as described in Section 4.2.1 (i.e., $OSV_1 = OSV_2 = ... = OSV_n$ and each OSV is computed using information from *all* ISV's). Following the description of *npGVT*, it will be apparent that if this assumption is relaxed, the same algorithm computes TSVT, the target-specific counterpart of GVT.

Our strategy is two-fold: (i) identify the values to be supplied in the state vectors by each LP, and (ii) design the algorithm to maintain these values consistently at each LP such that the reduced output of the NPSI Calculator provides GVT. For historical reasons, we

refer to individual elements of the state vectors as *T-values*. By definition, GVT is the minimum of two values: the minimum of the logical clocks of all LP's and the minimum unreceived message time among all LP's. Correspondingly, each LP maintains two *T*-values: σ_i, the logical clock value of LP_i, and υ_i, the smallest among the timestamps of all messages LP_i has sent out that have not yet been received or incorporated into the receiver's events list.

Given these *T*-values, the NPSI Calculator computes:

$\sigma' = MIN_{LP_i}(\sigma_i)$: the minimum of all the logical clocks

$\upsilon' = MIN_{LP_i}(\upsilon_i)$: the minimum unreceived message time

and GVT is computed as $GVT = MIN(\sigma', \upsilon')$.

Timestamps of antimessages must also be part of the GVT computation. To see why, consider a system of two LP's with LP_0 having σ_0 at 1000, LP_1 having σ_1 at 1500 and no messages in transit. Therefore, GVT has a value of 1000. Now, LP_0 sends LP_1 an antimessage with timestamp 1000 and then proceeds to execute its next event with timestamp 2000. While the antimessage is in transit, if antimessages are not acknowledged, we have σ_0 at 2000, σ_1 at 1500 and both υ_0 and υ_1 at ∞, giving a GVT of 1500. When the antimessage is finally received by LP_1, σ_1 falls to 1000, bringing GVT down to 1000. This is an error since GVT must be strictly non-decreasing. The error occurs because the smallest timestamp in the system, which is the timestamp of the antimessage, is momentarily absent from the GVT computation algorithm. The problem is solved by considering antimessages also in computing υ_i.

In order to maintain υ_i, an LP that sends a message must be informed when the intended receiver receives the message and incorporates it into its events list. To achieve this, *npGVT* acknowledges each message received by an LP using the two-phase protocol described in [Panc92]. The acknowledgment protocol, which operates over the reduction system rather than the native communication system, uses two more *T*-values, ρ_i and τ_i. Each of these is a 3-tuple comprised of the timestamp of the message being acknowledged, a globally unique message identifier (this is required since there may be multiple messages with the same timestamp) and a size field used for batched acknowledgments described later[1]. Thus, each LP_i has a four-element state vector consisting of the *T*-values: σ_i, υ_i, ρ_i

and τ_i. One way of generating globally unique message identifiers in a distributed manner with no overhead is by having the identifier consist of three fields: the LP number of the sender, the LP number of the receiver and a sequence number. Each LP maintains one sequence counter for each LP that it sends messages to. When an LP sends a message to another LP, it increments the counter for that receiver and uses its value in the message identifier. Thus, the message identifiers form a contiguous sequence for each ordered pair of communicating LP's. Each LP_i also maintains two lists: the *outstanding message list* contains messages sent by LP_i for which acknowledgments have not yet been received, and the *unacknowledged message list* contains messages received by LP_i that it has not yet acknowledged. The lists are sorted in ascending order of the sequence numbers in the message identifiers.

Figure 14 lists the algorithm *npGVT*. It has two concurrent procedures, PROCESS and DO_ACK. The PROCESS procedure invokes three subroutines, SEND_MSG, RCV_MSG and ROLLBACK as required. Note, RCV_MSG is called as a subroutine by PROCESS rather than being a concurrent procedure invoked when a message is received. This is because we assume that messages do not preempt event execution. We show later how to incorporate event preemption into this algorithm.

The PROCESS procedure implements all Time Warp specific operations. It unconditionally executes events on the events list. The state of the LP is saved periodically. After the execution of each event, the procedure checks for the arrival of new messages during this execution. If any new messages have arrived in the LP's logical past (stragglers), PROCESS calls ROLLBACK to perform the rollback. During a rollback, the logical clock of the LP is rolled back to the timestamp of the straggler. We have chosen to employ aggressive cancellation — we show later that lazy cancellation is easily accommodated in this algorithm. At the end of a rollback, the PROCESS procedure checks to see if any new stragglers have arrived during the rollback process that will cause the LP to roll back further. If so, ROLLBACK is called again. Thus, ROLLBACK is invoked repeatedly until all of the newly received messages are in the LP's logical future (i.e., the LP has rolled back far

1. It is assumed that the NPSI Calculator performs selective reduction operations (such as MIN) only on the timestamp field of *T*-values with multiple fields. The values in the other fields of the global output will equal the corresponding values of the input *T*-value that "wins" the selection.

```
PROCESS:   IF     there are events in the events list
           THEN   local_clock_i := σ_i := timestamp of next event;
                  Execute event; Perform SEND_MSG if required;
                  Optionally save state;
           WHILE  there are newly received messages with timestamps < σ_i
                  Perform ROLLBACK;
           FOR    each new message
                  Perform RCV_MSG;
           Optionally collect fossils;

SEND_MSG:  [IF    message_time < υ_i
           THEN   υ_i := message_time]
           Add message to outstanding list;
           Send the message;
           IF     message_sign > 0
                  Add antimessage to output list;

RCV_MSG:   IF     message_sign > 0
           THEN   Insert event into events list;
           ELSE   Delete the positive event;

           [IF    ρ_i = {∞, Φ, 0}
           THEN   ρ_i := (message_time, message_id, 1)
           ELSE   Add message to unacknowledged message list];

ROLLBACK:  local_clock_i := σ_i := rollback time;
           Roll back the execution of all events with time > rollback time;
           Restore state from the last time it was saved before rollback time;
           Rebuild the state up to rollback time if required;
           FOR    each antimessage with timestamp > rollback time
                  Perform SEND_MSG;
                  Delete it from the output list;

DO_ACK:    IF     [τ′ = ρ_i AND ρ_i ≠ {∞, Φ, 0}]            -- sender has seen ack
           THEN   Remove next batch to be acknowledged
                      from unacknowledged message list;    -- if one exists
                  Set ρ_i to acknowledge this batch;

                  [IF   ρ′ has been sent to this LP
                  THEN  τ_i := ρ′;
                        IF  ρ′ messages are in outstanding message list]
                        THEN Remove the acknowledged batch from outstanding message list;
                            IFtimestamp of acknowledged batch = υ_i
                            THENυ_i := smallest timestamp in outstanding message list
           ELSE   τ_i := {∞, Φ, 0} ;
```

Figure 14 - *npGVT*

---

enough). These messages are now incorporated into the events list of the LP in the RCV_MSG subroutine. Note, the ROLLBACK procedure calls SEND_MSG to send the anti-messages. This guarantees that the timestamps of antimessages are accounted for in the value of υ_i.

Two-phase Acknowledgment

Since LP's may not observe all of the state vectors emerging from the NPSIC, an LP sending an acknowledgment to another through the reduction system cannot assume that the latter will observe it. To overcome this problem, messages are acknowledged using a two-phase scheme: in the first phase, the sender of the message being acknowledged determines that the receiver of the message is acknowledging the message and in the second phase, the receiver determines that its acknowledgment has been observed by the sender. To acknowledge a message sent to it by LP_s, LP_r initiates the first phase by setting its ρ_r to the timestamp and message identifier of the received message. We refer to some ρ_i having a value other than $\{\infty, \Phi, 0\}$ (the null value) as a *primary acknowledgment* (or simply *acknowledgment*) and the act of setting ρ_i as *submitting a primary acknowledgment*. The timestamp of a primary acknowledgment is the same as that of the message which it is acknowledging. For now, assume that the timestamp of ρ_r is the smallest among all other acknowledgments submitted simultaneously. Thus, ρ' will equal ρ_r. When this is observed by LP_s (and it is guaranteed to ultimately do so, since LP_r does not change ρ_r), it knows that its message to LP_r is being acknowledged and the first phase is complete. LP_s initiates the second phase by setting its τ_s to ρ'. As with ρ_i, we refer to some τ_i having a value other than $\{\infty, \Phi, 0\}$ as a *secondary acknowledgment* and the act of setting τ_i as *submitting a secondary acknowledgment*. Since an LP submits a secondary acknowledgment only after observing a primary acknowledgment, every secondary acknowledgment corresponds to a primary acknowledgment and the timestamp of a secondary acknowledgment is equal to that of its primary acknowledgment. After some time, τ' will equal τ_s and therefore ρ_r. This is ultimately observed by LP_r (because LP_s does not change τ_s) which therefore determines that LP_s has processed its acknowledgment. LP_r then removes its acknowledgment by changing ρ_r. After some delay, this change is reflected in the output of the reduction network as a change in ρ'. When LP_s observes this change, it knows that LP_r has removed its acknowledgment and so it *relinquishes* the secondary acknowledgment by setting τ_s to $\{\infty, \Phi, 0\}$ and the second phase is complete.

At any time during the two phases described above, a new acknowledgment with a timestamp smaller than ρ_r may be submitted by some LP other than LP_r. As a result, ρ' will equal this new acknowledgment. This change in ρ' may occur either before or after

LP_s observes the old ρ'. In the first case, the first phase of the protocol is preempted; this situation is the same as having several acknowledgments with the new acknowledgment having the smallest timestamp. When the new acknowledgment completes, ρ_r again becomes the acknowledgment with the smallest timestamp and the first phase is restarted. In the second case, LP_s has observed the primary acknowledgment but the second phase is preempted. LP_s observes that its message is no longer being acknowledged and consequently relinquishes the second acknowledgment by setting τ_s to $\{\infty, \Phi, 0\}$. After the new acknowledgment completes, ρ' reverts to ρ_r. When LP_s observes this, it resumes the second phase by resubmitting its secondary acknowledgment and the second phase ultimately completes. Thus, acknowledgments are effectively nested.

Batched Acknowledgments

From the description above, it is clear that when multiple acknowledgments are submitted at the same time, only the one with the smallest timestamp proceeds. This serialization of acknowledgments is alleviated by an enhancement in which a batch of messages is acknowledged using a single physical acknowledgment. This is done using the third components (the batch-size field) of ρ_i, τ_i, ρ' and τ'. An acknowledging LP searches its unacknowledged message list for a batch of received messages with contiguous sequence numbers and sets its ρ_i to the 3-tuple {the smallest timestamp in the batch, the message identifier of the starting message in the batch, the number of messages in the batch}. The LP receiving the acknowledgment considers the entire batch of messages to be acknowledged and deletes them from its outstanding message list.

The `DO_ACK` procedure consists of two parts. The first `IF` statement implements the part of the two-phase protocol executed by an LP that receives a message while the second `IF` statement implements the part executed by an LP that sends a message.

Event Preemption and Lazy Cancellation

We indicate how two optional features of Time Warp (event preemption and lazy cancellation) can be incorporated into *npGVT*. In Figure 14, the event processing procedure (`PROCESS`) checks to see if new messages have arrived after the execution of each event. When one or more of the messages received during the execution of an event has a smaller timestamp than that of the event, an efficient implementation will preempt the execution of

the event (since it will be rolled back anyway) rather than wait for the execution to complete. To incorporate this into *npGVT*, the `RCV_MSG` subroutine is activated as a concurrent procedure with `PROCESS` and `DO_ACK` whenever the LP receives a message. In addition, at the start of the `RCV_MSG` procedure, a check is made to determine if the received message is in the LP's logical past and if so, the `ROLLBACK` procedure is called. In effect, each iteration of the `WHILE` and `FOR` loops in `PROCESS` is transformed into a concurrent invocation of `RCV_MSG`.

With lazy cancellation, upon rolling back, an LP sends out antimessages only as required. In Figure 14, the `ROLLBACK` subroutine implements aggressive cancellation, wherein antimessages are sent out during a rollback regardless of whether they are needed or not. To implement lazy cancellation, the `FOR` loop is removed from the `ROLLBACK` subroutine. Instead, the `PROCESS` procedure must now perform a check after every event execution, to see if any antimessages need to be sent out. If so, the `SEND_MSG` subroutine is called to send the antimessage.

5.3 Correctness

In this section, we establish the correctness of *npGVT*. We are concerned only with temporal correctness. We assume that the simulators are functionally accurate. By temporal correctness we mean that none of the committed events violate causality constraints. The following criteria define (temporal) correctness of a parallel simulation:

i) each LP must ultimately execute events in strictly non-decreasing timestamp order [Fuji90]

ii) the simulator must make progress if the application being simulated makes progress

The second criterion incorporates the concepts of freedom from deadlock as well as termination.

5.3.1 Correctness of a GVT Computation Algorithm

An *ideal time warp system* is one in which the LP's obtain all the information they need instantaneously. For instance, in an ideal system, when an LP receives a message sent to it by another, the sending LP becomes aware of the fact that the message has been received at the same instant the receiving LP receives the message. Obviously, no imple-

mentation can be ideal but every implementation has a corresponding ideal system. Timestamps often remain in an implementation for some time after they have disappeared from the ideal system. Returning to our previous example, typically, the sending LP is notified of the receipt of the message by some form of acknowledgment. Since this acknowledgment takes time to accomplish, the message remains outstanding for a longer period of time in the implementation than in the ideal system. This *lag* is especially true of implementations in which the GVT computation proceeds asynchronously with the simulation, i.e., the simulation is not suspended when GVT is computed. As a result, the GVT computed by an implementation is usually an approximation of the value of GVT in the corresponding ideal system. At any instant of real time *t*, we distinguish between the actual value of GVT, $GVT_a(t)$, and the computed value of GVT, $GVT_c(t)$, which is the value of GVT made available by the implementation at time *t*.

Since GVT is the commitment horizon, it is essential that the value of GVT not be overestimated at any time during the simulation. Otherwise it is possible to commit events that may be rolled back. Further, to ensure progress of the simulation, the estimated value of GVT must increase as the simulation proceeds. The correctness criteria for a GVT computation algorithm can now be stated as below:

Criterion 1: $GVT_c(t) \leq GVT_a(t)$ for all times *t*.

Criterion 2: If $GVT_a(t)$ increases at the instants of real time t_i, i=0,1,...,

then for each *i* there exists a finite *u* such that $u \geq t_i$ and $GVT_c(u) \geq GVT_a(t_i)$.

5.3.2 GVT Computation Model

The general reduction model of Figure 13 can be restated for computing GVT as in Figure 15. The GVT Calculator operates asynchronously, in reduction cycles that take Δ time units each. A new reduction cycle is initiated every δ time units, i.e., δ is the time required by the GVT Calculator to accept a single set of state vectors. Note, it is possible to have $\Delta \geq \delta$ by pipelining the GVT Calculator.

Each LP presents a new state vector to the GVT Calculator whenever an event occurs that causes a change in $GVT_a(t)$ (for example, executed event, received message, sent message, etc.). The GVT Calculator takes these state vectors and computes

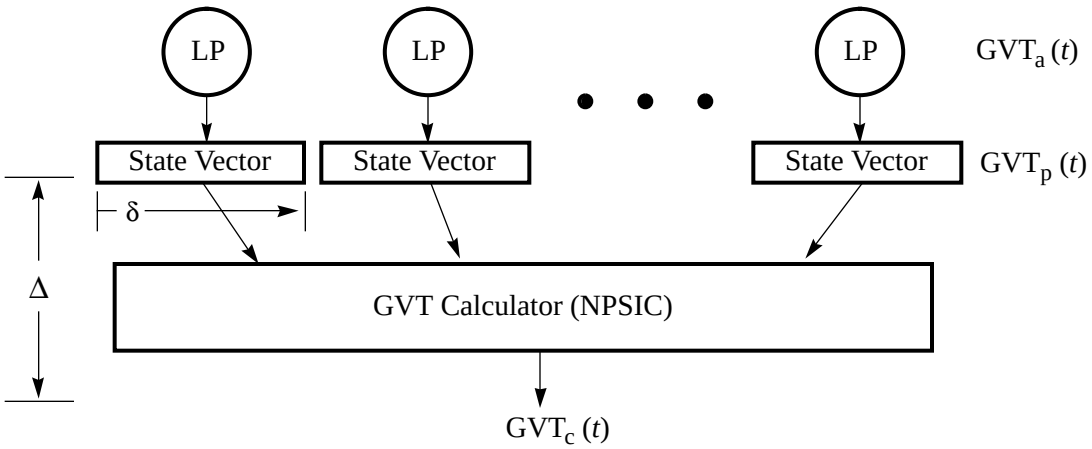


Figure 15 - GVT Computation Model for *npGVT*

GVT_c (t+Δ). If a new state vector is not submitted by any LP between two successive reduction cycles, the GVT Calculator simply reuses the old state vector for that LP.

5.3.3 Useful Properties

Before proving the correctness of *npGVT*, we present some properties of the reduction model and of *npGVT*. These will be used in the proofs that follow. Each of these properties can be trivially shown to be true. A short name is associated with each property and will be used along with the property number to refer to the property.

5.3.3.1 Properties of the Reduction Model

P1 *reduction operation*

Let $\mathbf{V}_1$, $\mathbf{V}_2$, ... $\mathbf{V}_n$ be the state vectors submitted by the n LP's such that $\mathbf{V}_i = <V_i^1, V_i^2, ..., V_i^m>$ where m is the size of each LP's state vector. Let θ_i $1 \le i \le m$ be binary associative operators. Then in one reduction cycle the GVT Calculator computes $\mathbf{G} = <G^1, G^2, ..., G^m>$ where $G^i := \theta_i <V_1^i, V_2^i, ..., V_n^i>$, $1 \le i \le m$ and **G** is the output state vector of the GVT Calculator. For the algorithms in this chapter, $m = 4$ and $\theta_1 = \theta_2 = \theta_3 = \theta_4 = MIN$. Recall that we assume a *global* reduction model instead of a *target-specific* one, for simplicity.

P2 *input cycle time*

A reduction cycle (i.e. a fresh computation of GVT) is started every δ time units.

P3 *reduction cycle time*

Each reduction cycle takes Δ time units to complete and $\Delta \geq \delta$.

P4 *periodic read*

Each LP reads the output of the GVT Calculator periodically, i.e. no LP will ignore the output of the GVT Calculator for an arbitrary amount of time.

5.3.3.2 Properties of *npGVT*

P5 *set clock*

σ_i always equals `local_clock`$_i$.

P6 *set unreceived message time*

When LP_i sends a message, if υ_i is greater than the timestamp of the message, υ_i will be set to the timestamp of the message; otherwise, there is no change in υ_i.

P7 *update unreceived message time*

When LP_i receives the primary acknowledgment for a message with timestamp υ_i, that message is removed from LP_i's outstanding message list and υ_i is set to the smallest among the timestamps of the remaining messages in its outstanding message list.

P8 *unreceived is minimum*

At any LP_i, υ_i is always equal to the smallest of the timestamps of the messages in its outstanding message list. This follows directly from P6 (set unreceived message time) and P7 (update unreceived message time)

P9 *message deletion*

A message is deleted from LP_i's outstanding message list only when LP_i receives the primary acknowledgment for that message.

P10 *set clock on receive*

When LP_i receives a message, σ_i is made less than or equal to the timestamp of the message and the message is incorporated into LP_i's events list before the primary acknowledgment of the message is initiated.

5.3.3.3 Properties of the `DO_ACK` Procedure

P11 *unique selection*

Given a set of primary acknowledgments submitted to the GVT Calculator, there is one among them (which has the smallest

timestamp among them, by P1 (reduction operation)) that is selected by the GVT Calculator.

P12 *secondary acknowledgment*

When LP_i observes a primary acknowledgment being sent to it, it submits the corresponding secondary acknowledgment within a finite, bounded amount of time. Otherwise, ρ_i is set to $\{\infty, \Phi, 0\}$.

P13 *persistent primary acknowledgment*

Once any LP_i submits a primary acknowledgment to the GVT Calculator, that acknowledgment is removed only when LP_i observes the corresponding secondary acknowledgment.

5.3.4 Correctness of *npGVT*

npGVT maintains two *T*-values at each LP_i, corresponding to the two component values of GVT: σ_i, the logical clock and υ_i, the smallest unreceived message time. In addition, *npGVT* performs acknowledgments of messages using two more *T*-values: ρ_i and τ_i. These four *T*-values define a *state vector* for each LP. We show next that *npGVT* satisfies the two correctness criteria of Section 5.3.1.

5.3.4.1 Criterion 1

Define $GVT_p(t)$ to be the value of GVT that can be *potentially* calculated at the input side of the GVT Calculator (Figure 15), i.e., given the set of state vectors presented to the GVT Calculator at time *t*, $GVT_p(t)$ takes on the value of GVT defined by these state vectors. To show that *npGVT* satisfies criterion 1, we prove three lemmas below.

LEMMA 1 If $GVT_p(s) \le GVT_a(s)$ for all times *s* at which a reduction cycle is started, then $GVT_c(t) \le GVT_a(t) \forall t$.

PROOF If a reduction cycle is started at time *s*, its result becomes available at the output of the GVT Calculator at time $s+\Delta$ (P3 (reduction cycle time)). By P2 (input cycle time), the next reduction cycle is started at time $s+\delta$. Since this cycle also takes Δ time units to complete, a fresh result becomes available at time $s+\delta+\Delta$. Continuing thus, we see that a fresh (not necessarily different) value of $GVT_c(t)$ is computed every δ units of time. Thus, $GVT_c(t)$ is a step function with the separation in time between steps being an integral multiple of δ. Consider the two (arbitrarily chosen) consecutive steps above which

form the time interval [s+Δ, s+δ+Δ). GVT_a (t) cannot decrease in this interval (because of its non-decreasing nature, by assumption of the functional correctness of *npGVT*). It follows that $GVT_c(t) \le GVT_a(t) \ \forall t$ if $GVT_c(s + \Delta) \le GVT_a(s + \Delta)$ for those times s+Δ at which a fresh result is computed at the output of the GVT Calculator. By design of the GVT Calculator, $GVT_c(s + \Delta) = GVT_p(s)$ and by assumption, $GVT_p(s) \le GVT_a(s)$. Thus, $GVT_c(s + \Delta) \le GVT_a(s)$. By the non-decreasing property of GVT_a (t), $GVT_a(s) \le GVT_a(s + \Delta)$. ■

We digress briefly to explain some notation. All of the *T*-values considered here (i.e., σ_i, υ_i, etc.), as well as $GVT_a(t)$, $GVT_p(t)$ and $GVT_c(t)$ are functions of real time. We use the time parameter throughout the paper only for $GVT_a(t)$, $GVT_p(t)$ and $GVT_c(t)$. For the other entities (especially σ_i and υ_i), we use the time parameter when it is required for clarity and omit it otherwise.

npGVT is designed such that any action by an LP that may cause GVT_a (t) to change is preceded by a change in the *T*-values being presented to the GVT Calculator that reflects that change. For instance, before sending a message, the timestamp of the message is incorporated in the υ_i of the sending LP. Similarly, when an LP receives a message, the timestamp of the message is incorporated into its σ_i before the message is acknowledged. This being the case, the following invariant is maintained by the algorithm:

LEMMA 2 At any instant t, there is at least one LP_k with $\sigma_k(t)$ or $\upsilon_k(t)$ less than or equal to GVT_a (t).

PROOF By definition, $GVT_a(t)$ equals either the `local_clock` of some LP_k at time t or the timestamp of a message sent by some LP_k to some other LP_l that has not yet been incorporated into LP_l's events list at time t (or both). In the first case, P5 (set clock) guarantees that $\sigma_k(t)$ will equal GVT_a (t) and the Lemma is true. In the second case, P10 (set clock on receive) guarantees that the primary acknowledgment of the message is issued only after the message is incorporated into LP_l's events list, i.e., $\sigma_l(t)$ is made equal to GVT_a (t) (it cannot be less than GVT_a (t)). Thus, for the entire time the message is "in transit", P9 (message deletion) ensures that the message in transit is also present in LP_k's

outstanding message list and P8 (unreceived is minimum) guarantees that the $\upsilon_k(t)$ will be less than or equal to $GVT_a(t)$. ■

LEMMA 3 $GVT_p(t) \leq GVT_a(t) \ \forall t$

PROOF $GVT_p(t)$ is defined as the GVT computed from the state vectors presented to the GVT Calculator at time *t*, i.e.:

$$GVT_p(t) = MIN_{LP_i}(\sigma_i(t), \upsilon_i(t))$$

which implies that $GVT_p(t) \leq \sigma_k(t)$ and $GVT_p(t) \leq \upsilon_k(t)$ for all LP_i. The lemma follows immediately from the truth of the invariant of Lemma 2 and these inequalities. ■

THEOREM 1 $GVT_c(t) \leq GVT_a(t) \ \forall t$

PROOF From Lemma 1, Theorem 1 is true if $GVT_p(t) \leq GVT_a(t)$ for those times *t*, at which the GVT Calculator starts a reduction cycle. From Lemma 3, $GVT_p(t) \leq GVT_a(t)$ $\forall t$. ■

Note the proof above covers the case when old state vectors are overwritten by new ones before they are processed by the GVT Calculator. In this case, the set of values of $GVT_p(t)$ processed by the GVT Calculator is a subset of the set of all the values assumed by $GVT_p(t)$. Since Lemma 2 is valid for all values of $GVT_p(t)$, in the case of overwrites it will be valid for those values which are processed by the GVT Calculator and Theorem 1 is true.

5.3.4.2 Criterion 2

Since acknowledgments are key to the progress of υ' and therefore $GVT_c(t)$, we first show that the DO_ACK procedure acknowledges messages correctly.

LEMMA 4 The DO_ACK procedure acknowledges messages such that if there are a finite number of unacknowledged messages with timestamps less than or

equal to the timestamp of a given unacknowledged message, that message is eventually acknowledged.

PROOF We consider acknowledgments under two situations:

Case I: Fixed set of acknowledgments

Assume first that a non-empty set of acknowledgments (*A*) submitted to the GVT Calculator does not change over the entire time taken to perform a single two-phase acknowledgment. By P11 (unique selection), one of these (call it P_W, an acknowledgment from LP_R to LP_S) will appear in the state vector output by the GVT Calculator. In addition, P_W will have the smallest timestamp in the set. By P13 (persistent primary acknowledgment) and our assumption of an unchanging acknowledgment set, P_W will continue to appear in the GVT Calculator's output. P4 (periodic read) guarantees that LP_S will observe P_W and by P12 (secondary acknowledgment), will submit the corresponding secondary acknowledgment (call it S_W from LP_S to LP_R) in finite time. P12 (secondary acknowledgment), P13 (persistent primary acknowledgment), P4 (periodic read) and our assumption of an unchanging acknowledgment set guarantee that after some finite time, the only secondary acknowledgment being submitted to the GVT Calculator will be S_W. After time Δ, S_W will appear in the output of the GVT Calculator and will continue to do so because P_W also appears in the output. P4 (periodic read) guarantees that LP_R will observe S_W and by P13 (persistent primary acknowledgment), will remove its primary acknowledgment, P_W. After time Δ, P_W will no longer appear in the GVT Calculator's output. When this fact is observed by LP_S, it will stop sending its secondary acknowledgment, S_W (P12 (secondary acknowledgment)). Thus, when a set of acknowledgments is submitted to the GVT Calculator and this set does not change for the duration of one acknowledgment, then a unique one of these (having the smallest timestamp) will complete in finite time. In the rest of this proof and the paper, if P_X is a primary acknowledgment, we refer to the two-phase process initiated by the submission of P_X as the *P_X acknowledgment.*

Case II: Changing set of acknowledgments

Consider the case where the set of acknowledgments (*A*) changes when the two-phase protocol is in progress. By P13 (persistent primary acknowledgment), the only change can be the submission of a new acknowledgment (call it P_N) by some LP_N. Thus we

have a new set of acknowledgments, $A' = A \cup P_N$. This new arrival can have one of two effects: it either replaces P_W as the winning acknowledgment or it does not. The second case is uninteresting as it does not affect the two-phase acknowledgment described earlier (i.e., we can apply the argument of Case I to show that the P_W acknowledgment will complete in finite time). When P_N replaces P_W in the GVT Calculator's output, we say the P_W acknowledgment is *preempted*. P_N may preempt either the first phase or the second phase of the P_W acknowledgment. We consider each separately.

Case II(a): first phase preempted

The first phase of the P_W acknowledgment completes when LP_S observes P_W. If the first phase is preempted, then LP_S has not observed P_W. The effect of this is that the first phase has not been started at all. Since P_N is the winning acknowledgment now, the P_N acknowledgment goes through to completion in some finite time. When LP_N stops sending P_N, the set of acknowledgments reverts to A from A'. Now the situation is exactly that before the arrival of P_N. Thus, the P_W acknowledgment will restart and complete in finite time.

Case II(b): second phase preempted

If P_N replaces P_W after LP_S has observed P_W (and hence submitted S_W), the second phase of the P_W acknowledgment is preempted. This may happen either after or before LP_R has observed S_W in the output of the GVT Calculator. In the first case, all that remains of the second phase is for LP_S to stop sending S_W. By P12 (secondary acknowledgment), this will indeed happen in a bounded amount of time since P_W has been replaced by P_N. Since LP_R has already seen S_W, it will no longer submit P_W and the P_W acknowledgment is complete. In the second case, after LP_S observes that P_N has replaced P_W (which it eventually will, by P4 (periodic read)) it will stop sending S_W (P12 (secondary acknowledgment)). The P_N acknowledgment now goes to completion. At the end of this, when LP_N removes P_N, P_W again becomes the winning acknowledgment (the set of acknowledgments is again A). After some finite time, LP_S observes this and resubmits its secondary acknowledgment, S_W (by P12 (secondary acknowledgment)). The P_W acknowledgment now completes in finite time.

Note the arguments of Case II may be applied recursively (i.e., if a preempting acknowledgment is itself preempted). Thus the two-phase acknowledgment protocol

allows acknowledgments to be preempted such that they are resumed later (i.e., the acknowledgments are properly nested). A necessary condition for an acknowledgment to preempt another is that the timestamp of the former must be less than or equal to that of the latter. Consequently, for an acknowledgment to be infinitely delayed (preempted), there must be an infinite number of messages with timestamps less than or equal to its own timestamp. ■

THEOREM 2 If $GVT_a(t)$ increases at the instants of real time t_i, i=0,1,..., then for each i there exists a finite u such that $u \geq t_i$ and $GVT_c(u) \geq GVT_a(t_i)$.

PROOF We prove this theorem by contradiction. Assume that $GVT_c(t) < GVT_a(t_i)$ for all $t \geq t_i$ and some i. Since $GVT_c(t_i) = GVT_p(t_i-\Delta)$, this means that $GVT_p(t) < GVT_a(t_i)$ for all $t \geq t_i$. By definition of $GVT_p(t)$, this implies that at every instant of time $t \geq t_i$, there must be at least one LP_k such that either $\sigma_k(t) < GVT_a(t_i)$ or $\upsilon_k(t) < GVT_a(t_i)$. Since $\texttt{local\_clock}_k(t) \geq GVT_a(t_i)$ $\forall LP_k$ and $\forall t \geq t_i$, it follows from P5 (set clock) that $\sigma_k(t) \geq GVT_a(t_i)$ $\forall LP_k$ and $\forall t \geq t_i$. The condition $GVT_a(t_i) > GVT_a(t_j)$ $\forall j < i$ implies that there are only a finite number of unacknowledged messages at any LP with timestamps less than or equal to $GVT_a(t_{i-1})$. Lemma 4 guarantees that all of these messages are acknowledged in finite time. P7 (update unreceived message time) then ensures that any LP's υ_i can remain less than $GVT_a(t_i)$ for only a finite amount of time. The assumption is thus contradicted. ■

In summary, we have proved that *npGVT* satisfies the two correctness criteria for any GVT computation algorithm:

i) The computed GVT is always less than or equal to the actual value of GVT

ii) The computed GVT advances if the simulation advances.

We have thus described a correct method of computing and disseminating critical synchronization information rapidly using the reduction model to support NPSI adaptive protocols.

5.4 Global vs. Target-Specific

There are two aspects to the GVT computation scheme described above: *npGVT*, the algorithm that maintains a pair of T-values (σ_i and υ_i) consistently at each LP_i, and a global reduction model that computes global minima corresponding to these two T-values. A global reduction model was assumed in Section 5.2 since GVT is a global value. It is possible to use *npGVT* in conjunction with the target-specific reduction model of Figure 13. In this case, *npGVT* computes the target-specific analog of GVT, defined for each LP_i as follows:

$$TSVT_i = MIN_{LP_j}(\sigma_j, \upsilon_j) \quad \forall LP_j \in \text{ the predecessor set of } LP_i$$

The proofs of correctness of *npGVT* can be extended to the case where it is used with a target-specific reduction model.

5.5 An Implementation of *npGVT*

Implementation of *npGVT* requires that we first realize the reduction model of Figure 15. As noted earlier, one way of realizing this model is to use an asynchronous reduction network. We describe briefly one such network designed and built at the University of Virginia [RePS93]. This network and its associated hardware components are referred to as the *framework hardware* (for historical reasons).

5.5.1 Framework Hardware

The hardware configuration is shown in Figure 16. The main components of this hardware are the *parallel reduction network* (PRN) and the *auxiliary processors* (AP). The PRN implements the NPSIC. For each *host processor* (HP), there is an AP which is a general purpose processor such as the HP itself. The HP's perform simulation specific tasks (executing events, sending and receiving messages, saving state, etc.) while the AP's perform the synchronization tasks (i.e., they are responsible for submitting state vectors to the PRN, reading the output of the PRN and executing synchronization algorithms). When an HP causes its state to change (by executing an event, receiving a message, etc.) it communicates this change to its AP. The AP incorporates such changes into state vectors and submits them to the PRN for reduction. The AP also reads the globally reduced output state vector produced by the PRN and makes selected subsets of this available to the HP.

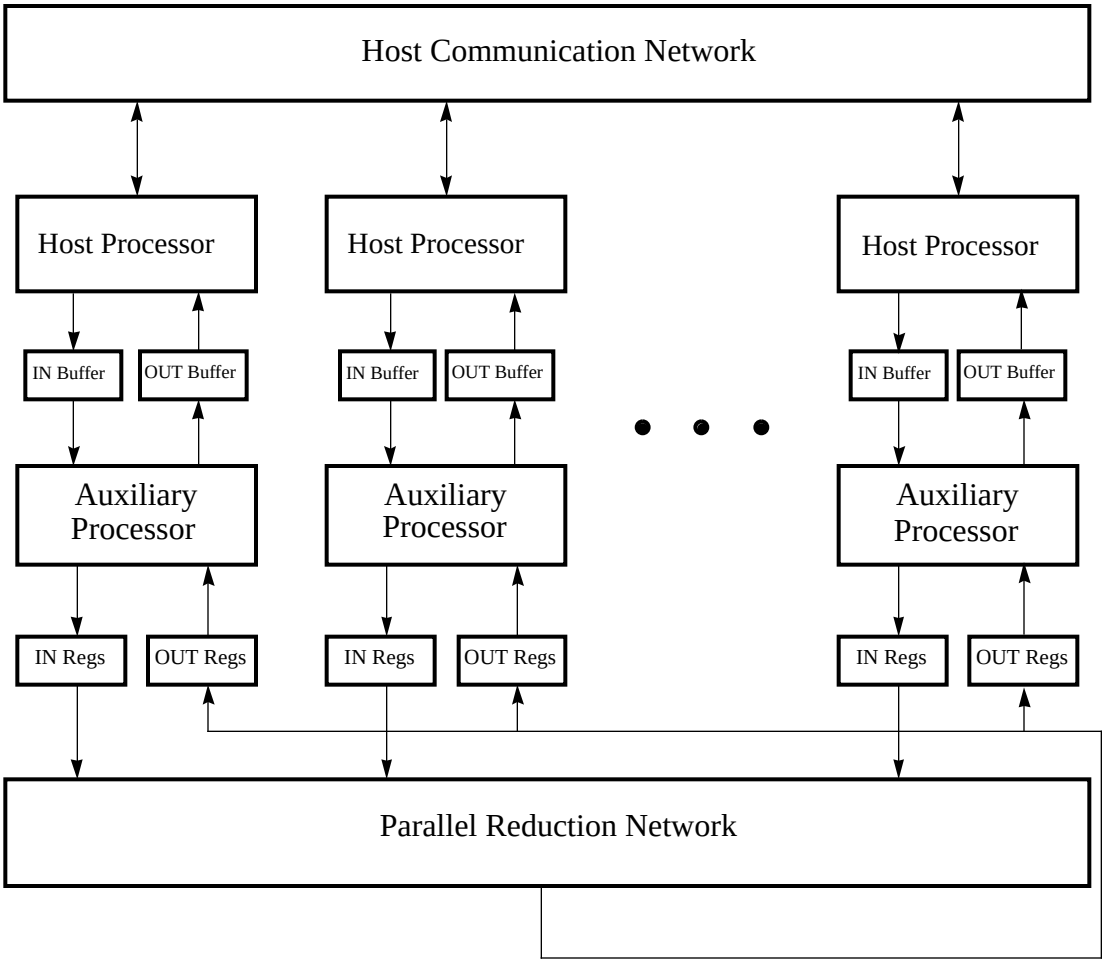


Figure 16 - Realizing the reduction model

The interface between the HP and the AP consists of two unidirectional channels (implemented as buffers). For information flowing from the HP to the AP, it is required that none of it be lost and that the AP process the information in the order sent out by the HP. For these two reasons, the IN Buffer is a FIFO. In the other direction, the only requirement is that the most recent version of the AP's output be available to the HP. Accordingly, the OUT Buffer consists of a single cell buffer, which is repeatedly overwritten atomically by the AP each time it presents new data to the HP. We note that this configuration is one of many possible implementations of the general reduction model. In particular, the choice of having dedicated processors for simulation and synchronization tasks and their asynchronous operation with respect to each other introduces latencies in the data path that are absent in the general model. This requires minor modifications to *npGVT,* described in Section 5.5.2.

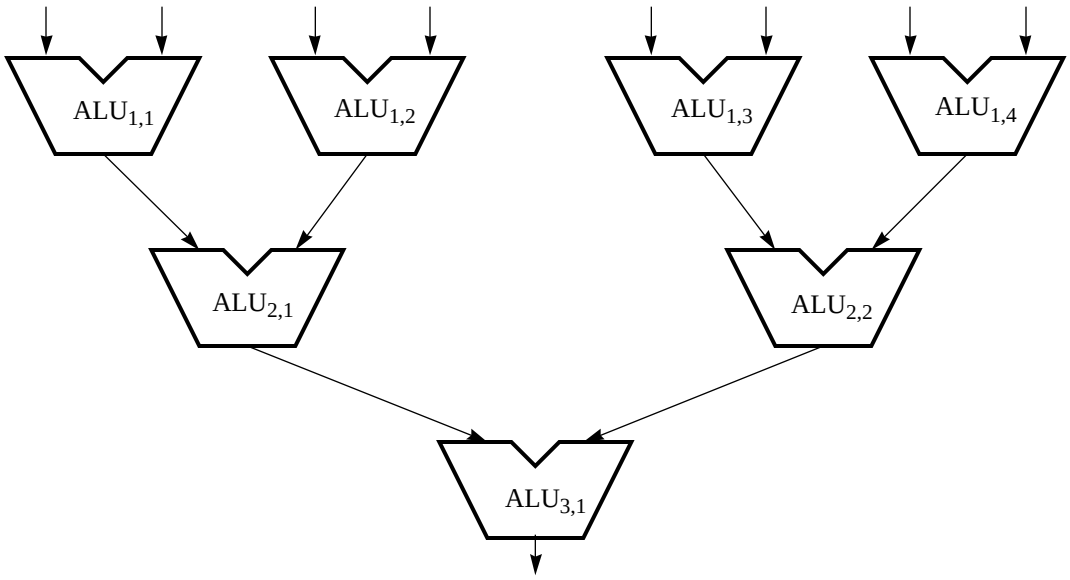


Figure 17 - A Parallel Reduction Network

The PRN operates in reduction cycles, reducing a set of state vectors, one from each processor, to a single output state vector in each cycle. Subsequent reduction cycles re-use state vectors for those AP's which do not present new state vectors. To satisfy these requirements, the design includes a custom interface between the AP's and the PRN. The AP's write new state vectors to the IN registers and read globally reduced state vectors from the OUT registers. Likewise, the PRN reads state vectors from the IN registers, reduces them and writes the global output to the OUT registers. The IN and OUT registers are comprised of three sets of registers each which provide the isolation between the AP's and the PRN. The detailed operation of these register interfaces is described in [RePS93].

The PRN is a binary tree of height $\log_2 n$, where n is the number of processors. Figure 17 shows a PRN for eight processors. Each node of the tree contains a general purpose ALU which performs reduction operations on its two operands. The reductions are performed in parallel across all the ALU's. The PRN interfaces with each processor through the IN and OUT registers. At each LP, the IN registers present a state vector (which may be the same as the one used in the previous cycle) to the PRN. The PRN starts the reduction by reading the first elements of each state vector and reducing them pair-wise at the top row of ALU's. These ALU's then pass the reduced values down to the second row of ALU's. While the second row of ALU's reduces these values, the top row reduces the second ele-

ments of each state vector. Thus, the PRN operates in a pipelined fashion. The time required for a set of values to pass through a single PRN stage is called the *minor cycle time*, while the time required for the top row ALU's to read all the elements of the state vectors is called the *input cycle time*. Note that an input cycle consists of *m* minor cycles, where *m* is the size of the state vectors. Because of the pipelined nature of the PRN, it takes $\log_2 n$ minor cycles for the global reduction of the first element of a set of state vectors and thereafter, reductions of the remaining *m*-1 elements emerge from the PRN one minor cycle apart. Thus, a reduction cycle, which is the total time required to produce a globally reduced output state vector from a set of *n* input state vectors of size *m* each, is $(\log_2 n + (m\text{-}1))*c$, where *c* is the minor cycle time.

Besides the ALU, each PRN node has some additional logic to accommodate *T*-values that have multiple components (such as ρ_i and τ_i). Each value submitted for reduction is accompanied by a *tag* which is expected to carry the other components of a *T*-value. At each ALU, the tags of the two operands are brought to a selector switch. The choice of which tag is propagated down to the next stage is made by a control signal from the ALU. For example, if the ALU performs a selective operation such as a *minimum* on the values, the tag of the smaller of the two values will proceed downward. In non-selective operations, the choice can be arbitrary but should be deterministic.

5.5.2 Implementing *npGVT* on the Framework Hardware

The framework hardware replaces a single processor with two processors: an HP to execute application specific tasks and an AP to execute the synchronization tasks. In order to execute *npGVT* of Figure 14 on the framework hardware, the synchronization aspects must be separated from the application specific ones. Figure 18 shows *npGVT2P*, which consists of two algorithms, one to be executed on each HP and one on each AP. We make the assumption that each LP is allocated its own HP-AP pair of processors in order to simplify the description of the algorithm as well as the correctness proofs in the next section. The algorithm can be extended easily to allow more than one LP per pair of processors. We use HP_i and AP_i to refer to the HP-AP pair on which LP_i is executing. In *npGVT2P*, the HP maintains the logical clock, the events list and the output list (list of antimessages) while

```
HOST_PROC: IF     there are events in the events list
           THEN   local_clock = timestamp of next event
                  Enqueue (NEW_CLOCK, local_clock);
                  Execute event; Perform SEND_MSG if required;
                  Optionally save state;
           WHILE  there are newly received messages with timestamps < local_clock
                  Perform ROLLBACK;
           FOR    each new message
                  Perform RCV_MSG;
           Optionally collect fossils;

SEND_MSG:  Enqueue (SENT_MSG, message_time, message_id);
           Send the message;
           IF     message_sign > 0
                  Add antimessage to output list;

ROLLBACK:  local_clock := rollback time;
           Enqueue (NEW_CLOCK, local_clock);
           Roll back the execution of all events with time > rollback time;
           Restore state from the last time it was saved before rollback time;
           Rebuild state up to rollback time if required;
           FOR    each antimessage in output list with time > rollback time
                  Perform SEND_MSG;
                  Delete it from the output list;

RCV_MSG:   IF     message_sign > 0
           THEN   Insert corresponding event into events list;
           ELSE   Delete the positive event;
           Enqueue (RCVD_MSG, message_time, message_id);
```

Figure 18 - *npGVT2P*: Host Processor Algorithm

---

the AP maintains the four *T*-values — σ_i, υ_i, ρ_i and τ_i, the outstanding message list and the unacknowledged message list.

Recall that the communication channel between an HP and its AP is functionally a FIFO. An HP communicates with its AP by inserting a tagged communication into the FIFO with the tag indicating the nature of the communication. In the algorithm for the HP, this is indicated by the statement `Enqueue (tag, value`$_1$`, value`$_2$`, ...)`. The AP is responsible for three major tasks:

- it must read the output of the PRN and propagate relevant portions to the HP
- it must process as soon as possible the tagged communications that its HP inserts into the FIFO (since speed of reduction is critical)
- it must perform acknowledgments of messages through the PRN

It is important that each of these tasks is performed frequently and none gets delayed arbitrarily. To satisfy these requirements, the AP's algorithm is a single loop. The AP

```
AUX_PROC:  Read the PRN output;
           Write global state vector to HP interface if it has changed;
           Perform DO_ACK;
           IF     FIFO is not empty
           THEN   Get next entry from FIFO;
                  CASE (entry_type):
                      NEW_CLOCK:  σ_i := new_clock_value;

                      SENT_MSG:   IF message_time < υ_i
                                  THEN υ_i := message_time;
                                  Add message to outstanding message list;

                      RCVD_MSG:   [IF ρ_i = {∞, Φ, 0}
                                  THEN ρ_i := (message_time, message_id, 1)]
                                  ELSE Add message to unacknowledged message list;

DO_ACK:    IF     [τ′ = ρ_i AND ρ_i ≠ {∞, Φ, 0}]
           THEN   Remove next batch to be acknowledged
                      from the unacknowledged message list;
                  Set ρ_i to acknowledge this batch;

           [IF    ρ′ has been sent to me
           THEN   IF  messages in ρ′ batch are not in outstanding message list
                  THEN τ_i := {∞, Φ, 0} ;            -- ignore primary ack
                  ELSE τ_i := ρ′];
                      Mark acknowledged messages as "deleted";
                      Remove any other messages sent to the same LP which have been marked
                          "deleted" from outstanding message list;
                      IF    timestamp of acknowledged batch = υ_i
                      THEN  υ_i := smallest timestamp among messages not marked
                            "deleted" in outstanding message list;
                  ELSE τ_i := {∞, Φ, 0} ;
```

Figure 18 (continued) - *npGVT2P*: Auxiliary Processor Algorithm

begins each iteration by reading the PRN output and propagating relevant portions to the HP if required. Having obtained the new global values, the AP checks for acknowledgments by executing the DO_ACK procedure. Finally, it removes one tagged communication from the FIFO (if there is one) and processes it.

5.6 Correctness of *npGVT2P*

Recall that *npGVT2P* is derived from *npGVT* by distributing the work between the HP and the AP. This organization introduces latency in the data path for synchronization data (Figure 19), i.e., when an HP changes state so that GVT_a (t) changes, this change is not reflected immediately in the state vector presented to the GVT Calculator. Moreover, the HP and the AP are asynchronous. This means that after an HP enqueues a communication in its FIFO, it continues processing without waiting for the AP to process that communication. This is the main difference between the computational models for *npGVT2P* and

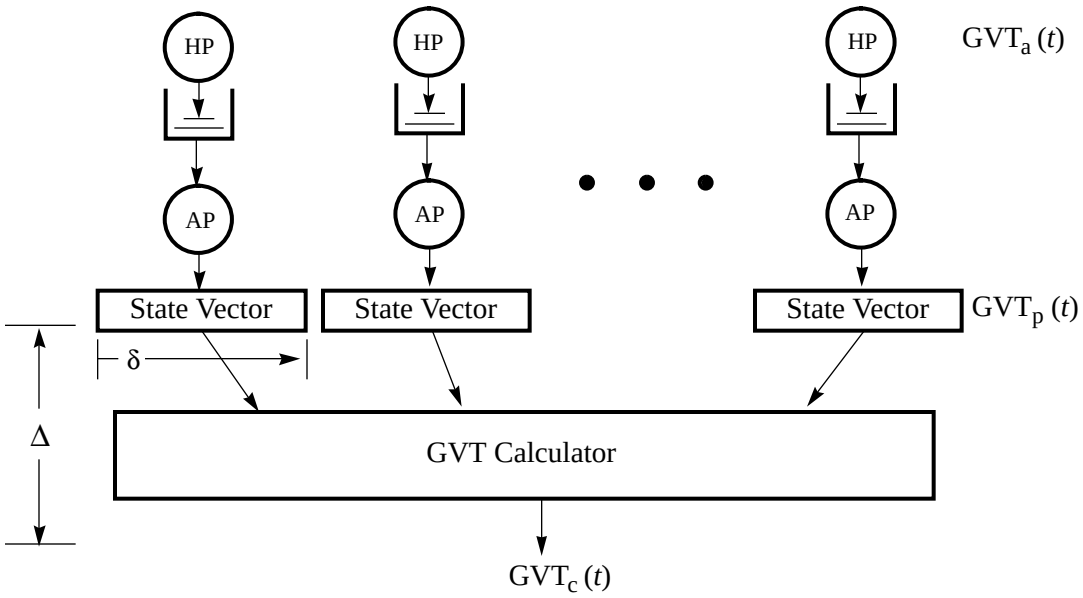


Figure 19 - GVT Computation Model for *npGVT2P*

npGVT. For instance, in *npGVT*, the time of the next event is always reflected in σ_i (and therefore GVT$_p$ (*t*)) before the HP starts executing the event, while in *npGVT2P*, σ_i may be set to the time of the event after the event has completed execution. In this section, we show that correctness of *npGVT2P* requires only a minor modification to the `DO_ACK` procedure of *npGVT*. As with *npGVT*, the correctness of *npGVT2P* is determined by the two correctness criteria of Section 5.3. We show next that *npGVT2P* satisfies these two criteria.

5.6.1 Properties of the Framework Hardware

As before, we use some properties of *npGVT2P* (which are trivially true) to prove its correctness. These properties supplement the general properties of Section 5.3.3. It is easily verified that these general properties are true for the PRN-based implementation of the reduction model in Figure 19 and for *npGVT2P*.

P14 *no loss*

No communication from the HP to the AP is lost.

P15 *finite delay*

No communication in a FIFO remains unprocessed by the AP for an arbitrary amount of time. This assumes that the communications are enqueued at an average rate that is lower than the average rate at which the communications are processed.

5.6.2 Criterion 1

Define the *minimum time* at time t, $\mu_i(t)$ of AP_i as:

$$\mu_i(t) = MIN(\sigma_i(t), \upsilon_i(t))$$

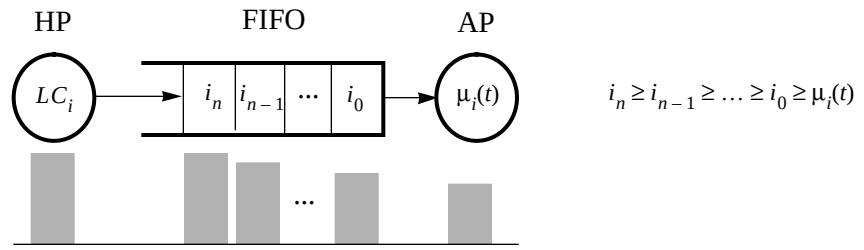
THEOREM 3 $GVT_c(t) \leq GVT_a(t)\ \forall t$

PROOF Comparing Figure 19 with Figure 15, we see that the relation between $GVT_c(t)$ and $GVT_p(t)$ has not changed, i.e. Lemma 1 holds here as well: if $GVT_p(s) \leq GVT_a(s)$ for all times s at which a reduction cycle is started, then $GVT_c(t) \leq GVT_a(t) \forall t$. We are thus required only to prove that $GVT_p(s) \leq GVT_a(s)$ for those times s at which a new reduction cycle is initiated. Once again, we do this by showing that *npGVT2P* maintains the invariant of Lemma 2, which is restated here using our definition of μ_i above:

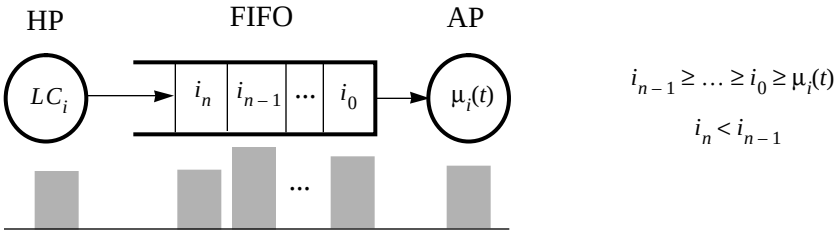
I: At any instant t, there is at least one LP_i such that $\mu_i(t) \leq GVT_a(t)$.

For any t, $GVT_a(t)$ is the timestamp of one or both of the following: (i) an event being executed (or just executed) at some HP, or (ii) a message in transit. We see that in *npGVT2P*, an HP always enqueues a `NEW_CLOCK` communication with the timestamp of the next event before executing that event. Similarly, an HP enqueues a `SENT_MSG` communication before sending a message. As a result, at any time t, we see that the FIFO of at least one LP (LP_i) will have a tagged communication with timestamp $GVT_a(t)$. With regard to this communication, there are two possibilities: either the AP of the corresponding HP has processed the tagged communication at t or it has not. In the first case, we see from *npGVT2P* (the `NEW_CLOCK` and `SENT_MSG` cases of the `CASE` statement in the AP's main procedure) that $\mu_i(t) \leq GVT_a(t)$ and the invariant will hold. In the remainder of this proof, we analyze the second case wherein the FIFO of some LP contains a communication with timestamp $GVT_a(t)$ and this communication has not yet been processed by the AP.

We use the pictorial notation of Figure 20 to show the timestamps present in an HP-AP pair and the FIFO between them. In this notation, vertical bars are used to indicate the relative values of the timestamps of the corresponding entities (HP, FIFO element or AP). The height of the bar is proportional to the value of the timestamp, i.e. the larger the timestamp, the taller the bar. The bar for the HP depicts the value of the HP's logical clock (LC).



(a) LP without rollback

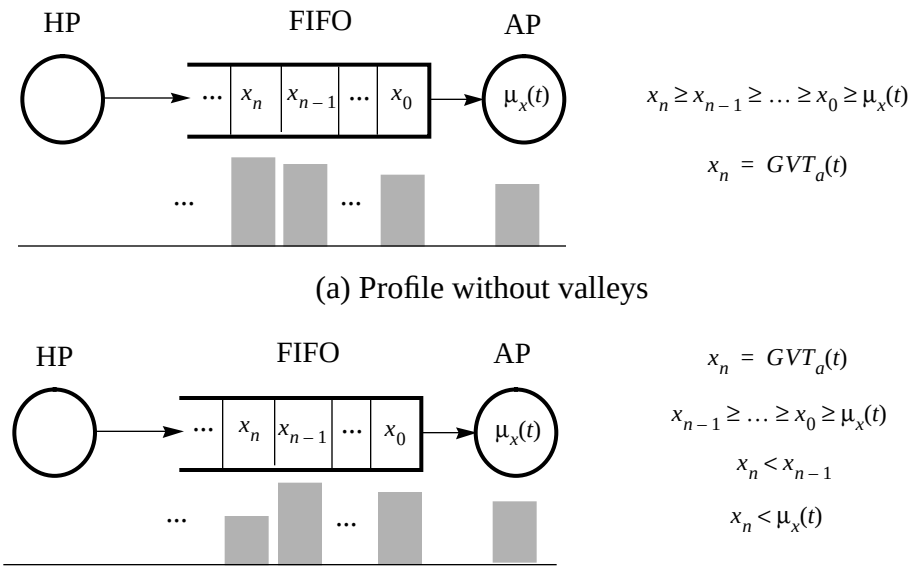


(b) LP with rollback

Figure 20 - Timestamp profile of LP_i

For each element in the FIFO, the bar depicts the timestamp of the communication enqueued (we consider only those elements that have a value enqueued in them and use i_j to symbolically denote the timestamp of the *j*'th communication from the head of the FIFO of LP_i). Finally, the bar for the AP depicts the AP's minimum time, μ_i. Figure 20a shows an LP that has been processing forward in logical time in the recent past, while Figure 20b shows an LP that has suffered a rollback recently. We refer to a dip in the timestamps in a FIFO (such as i_{n-1} to i_n) as a *valley.* The effect of a rollback is to create a valley in the LP's FIFO. With regard to an element in the FIFO, we say that all of the elements closer to the AP are *ahead* of it. When a particular timestamp is irrelevant, it is omitted.

At time *t*, consider the LP (LP_x) with a communication with timestamp GVT_a (*t*) enqueued in its FIFO. The FIFO of LP_x can have one of the two profiles shown in Figure 22, with x_n = GVT_a (*t*) (strictly speaking, the FIFO can have more than two profiles, depending on the number of valleys in the profile; however, there are only two interesting types: those with valleys ahead of the communication with timestamp GVT_a (*t*) and those without). If LP_x does not have any valleys ahead of this communication, then we see from Figure 22a that $\mu_x(t) \leq GVT_a(t)$ and the invariant will hold. The interesting case arises when the communication with timestamp GVT_a (*t*) is present in LP_x's FIFO due to a recent rollback (Figure 22b). In this case, it may happen that $\mu_x(t) > GVT_a(t)$. For the invariant to



(a) Profile without valleys

(b) Profile with a valley

Figure 21 - Possible profiles

be true, there must be some other LP_r, such that $\mu_r(t) \leq GVT_a(t)$. We show now that this is the case.

Consider the LP (LP_y) that sent the straggler message that caused the rollback in LP_x corresponding to the valley in Figure 22b (in the case where LP_x's profile has more than one valley, consider the valley closest to the communication with timestamp $GVT_a(t)$, but ahead of it in the FIFO). HP_y will have enqueued a `SENT_MSG` communication with timestamp $GVT_a(t)$ into its FIFO before sending the straggler. If this communication has been processed by AP_y, then $\upsilon_y(t) \leq GVT_a(t)$ (since AP_x has not yet acknowledged the straggler message) and thus $\mu_y(t) \leq GVT_a(t)$. If not, there are two possible situations: (i) LP_y is not the root of the rollback chain to LP_x or (ii) it is the root of the rollback chain to LP_x. We consider the two cases separately.

Case (i): Assume LP_y is not the root of the rollback chain to LP_x (i.e., LP_y has rolled back recently) and the `SENT_MSG` communication has not been processed by its AP so that its profile is similar to that shown in Figure 22b. Then LP_y's situation is similar to that of LP_x. As before, we proceed further up the rollback chain and examine LP_y's predecessor. This process terminates in one of two ways: (a) we reach a pair of LP's in the chain, LP_i and LP_j such that LP_i is before LP_j in the chain and AP_i has processed the `SENT_MSG`

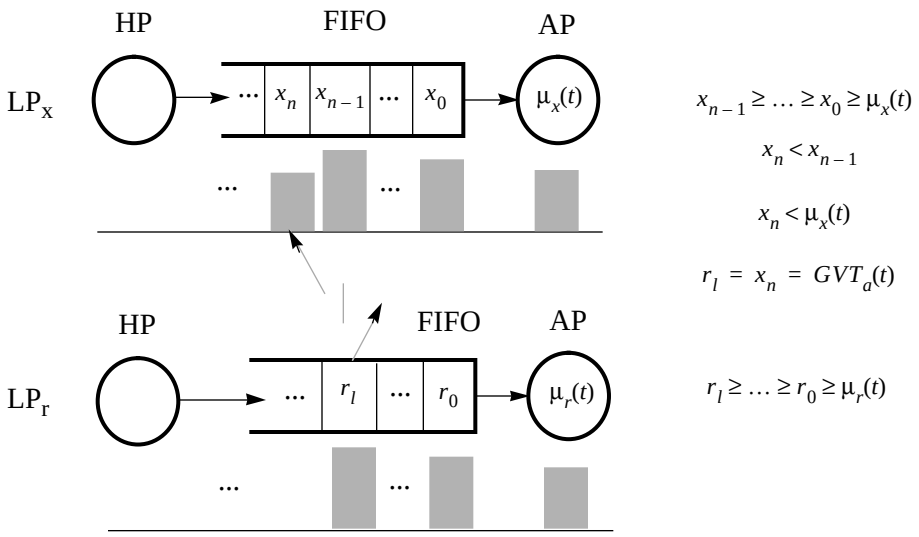


Figure 22 - Profile of the root of the rollback chain

communication from HP_i or (b) we reach the root of the rollback chain. In case (a), as explained earlier, since AP_j has not acknowledged the straggler message, LP_i will have $\mu_i(t) \leq GVT_a(t)$. Case (b) is the same as case (ii) above, which we consider next.

Case (ii): If the root of the rollback chain (LP_r) has a FIFO profile as shown in Figure 22[1], then the invariant holds since $\mu_r(t) \leq GVT_a(t)$. On the other hand, if its FIFO profile is as shown in Figure 23, then it has suffered a rollback recently, whose effects are still in its FIFO (r_p) and consequently, the invariant may not be satisfied by LP_r. However, LP_r is now in a situation very similar to that of LP_x and we apply our logic again, traversing backwards along the second rollback chain to LP_r. We show next that this process will terminate.

Consider the process again. We start with the LP that has the minimum timestamp in its FIFO. As we proceed up the rollback chain, we mark each LP that we visit. Note that each time a marked node is revisited, we move closer to the AP in the FIFO (because we are following a causal chain and 'A caused B' implies 'A occurred earlier than B'). Assume

1. A dashed arrow is used to connect the tagged communication in a FIFO corresponding to a rollback-causing straggler message, with the valley due to that rollback in the succeeding LP. For instance, r_l is the `SENT_MSG` communication corresponding to the message sent by LP_r that started the cascading rollback. A vertical dotted line is used to indicate that intermediate LP's have been omitted for compactness.

now that we reach the root of this chain and the root has a valley ahead in its FIFO, due to another rollback chain. We proceed up this new rollback chain, once again marking visited nodes. Here again, if we revisit previously marked nodes, we will visit them at a point closer to the AP, each time. Since the number of processors is finite (we cannot keep marking unmarked nodes indefinitely) and the number of cells in any FIFO ahead of the first element visited is finite, this procedure must ultimately terminate in one of the following ways:

i) We reach an LP_i whose AP has seen the `SENT_MSG` communication from its AP and therefore has $\mu_i(t) \le GVT_a(t)$.

ii) We reach the root of a rollback chain (LP_i) whose FIFO profile has no valleys (Figure 22) ahead of the communication corresponding to the rollback-causing straggler, in which case LP_i has $\mu_i(t) \le GVT_a(t)$.

Note that the entire argument for the truth of the invariant relies upon property P14 (no loss) of Section 5.6.1. Given that the invariant is true, Lemma 3 holds for *npGVT2P* as well and the proof of Theorem 3 parallels that of Theorem 1. ■

5.6.3 Criterion 2

As noted earlier, the addition of the FIFO requires a modification to the `DO_ACK` procedure as can be seen by comparing the corresponding procedures of Figure 14 and Figure 18. In *npGVT*, the timestamp and identifier of a message are always inserted into the outstanding message list prior to sending the message. In *npGVT2P*, the HP sends the mes-

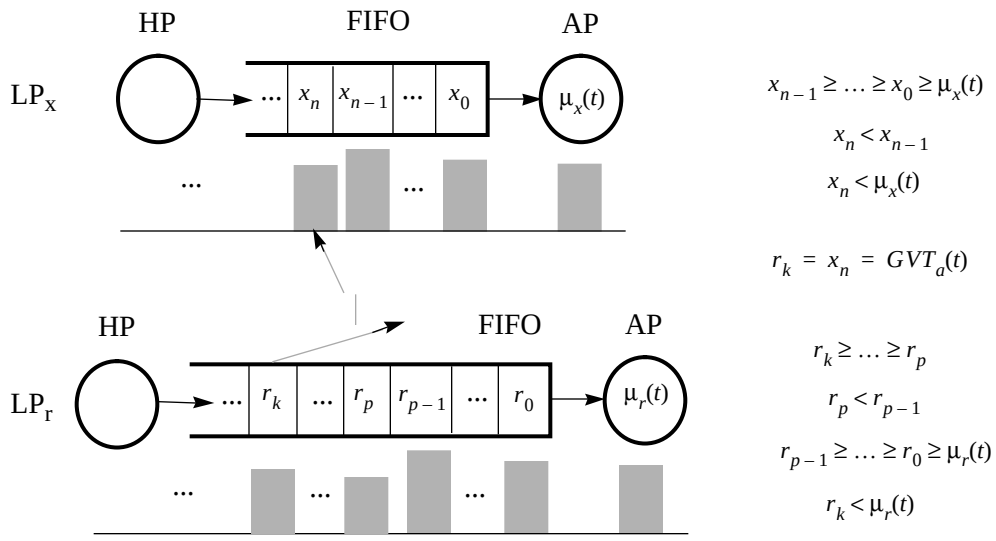


Figure 23 - Profile of the root of a rollback chain with a valley

sages while the AP maintains the outstanding message list and the two processors are asynchronous. This leads to a race condition in the following situation:

It is possible (though extremely unlikely given the speed of the framework hardware) that HP_i sends out a message to HP_j and enqueues a SENT_MSG communication to AP_i such that the message is received by HP_j and the primary acknowledgment for the message is initiated by AP_j before AP_i processes the SENT_MSG communication. As a result, it may happen that AP_i observes AP_j's primary acknowledgment emerging from the PRN, but does not have the corresponding message identifier in its outstanding message list. With *npGVT*, the DO_ACK procedure executing at AP_i assumes that the message was in the outstanding message list earlier but has been removed because its primary acknowledgment has been observed previously (i.e., it simply assumes that the second phase of the acknowledgment for that message had been preempted). Therefore it submits the corresponding secondary acknowledgment. When AP_j observes this secondary acknowledgment, it removes its primary acknowledgment and the two-phase acknowledgment is completed. Finally, when AP_i processes the SENT_MSG communication, it inserts the message identifier into its outstanding message list where it remains indefinitely, since AP_j will never acknowledge that message again. In such a case, υ_i (and hence $GVT_c(t)$) will not advance beyond the timestamp of that message.

The problem here is that the DO_ACK procedure in *npGVT* is unable to distinguish the case where a message (or a batch of messages) has appeared in the sender's outstanding message list and subsequently been deleted due to the receipt of the corresponding primary acknowledgment from the case where the message has simply not appeared in the sender's outstanding message list because the corresponding SENT_MSG communication has not yet been processed by the AP. The capability to distinguish these cases is important because in the former case, the sender must submit the secondary acknowledgment while in the latter, it must not (to avoid the race condition described previously). To solve this problem, in *npGVT2P* we mark messages for which a primary acknowledgment has been received as "deleted" but do not physically remove them from the list. Thus, an AP executing *npGVT2P* ignores any primary acknowledgments it receives for messages that are not present in its outstanding message list but submits the secondary acknowledgment for those that are present, whether marked "deleted" or not. To keep the sizes of the outstanding mes-

sage lists small, messages marked “deleted” must be removed from the lists periodically. It is clear that property P13 (persistent primary acknowledgment) from Section 5.3.3.3 is true for the DO_ACK procedure of *npGVT2P* as well. Consequently, when AP_i receives a primary acknowledgment from AP_j, it is guaranteed that all previous acknowledgments from AP_j have completed. Thus, whenever AP_i receives a primary acknowledgment from AP_j, it removes all other messages in its outstanding message list sent to AP_j that have been marked “deleted”.

With regard to property P11 (unique selection), since the only difference between *npGVT* and *npGVT2P* is the method of submission of primary acknowledgments, this property is true for the DO_ACK procedure of *npGVT2P* as well. For *npGVT2P,* P12 (secondary acknowledgment) is true only if the corresponding message is present in LP_i’s outstanding message list when the primary acknowledgment is received. If the message is not present in the outstanding message list, the AP ignores the primary acknowledgment until the message is inserted. However, once a message has been sent by an HP, P14 (no loss) and P15 (finite delay) guarantee that the corresponding SENT_MSG communication will be processed and therefore this message will be inserted into the corresponding AP’s outstanding message list within finite time. Thus, P12 holds for the DO_ACK procedure of *npGVT2P* as well. Since the proof of Lemma 4 uses only properties P4 and P11 through P14, Lemma 4 holds for *npGVT2P* as well.

THEOREM 4 If $GVT_a(t)$ increases at the instants of real time t_i, i=0,1,..., then for each i there exists a finite u such that $u \geq t_i$ and $GVT_c(u) \geq GVT_a(t_i)$.

PROOF We prove this theorem by contradiction. Assume that $GVT_c(t) < GVT_a(t_i)$ for all $t \geq t_i$ and some i. Since $GVT_c(t_i) = GVT_p(t_i\text{-}\Delta)$, this means that $GVT_p(t) < GVT_a(t_i)$ for all $t \geq t_i$. By definition of $GVT_p(t)$, this implies that at every instant of time $t \geq t_i$, there must be at least one LP_k such that either $\sigma_k(t) < GVT_a(t_i)$ or $\upsilon_k(t) < GVT_a(t_i)$. Since $\texttt{local\_clock}_k(t) \geq GVT_a(t_i)$ $\forall LP_k$ and $\forall t \geq t_i$, there are only a finite number of NEW_CLOCK communications in each LP’s FIFO, followed by a NEW_CLOCK communication with value $\texttt{local\_clock}_k(t_i)$. By P15 (finite delay), $\sigma_k(t) \geq GVT_a(t_i)$ $\forall t \geq t_i + \alpha$ for some finite delay α. The condition $GVT_a(t_i) > GVT_a(t_j)$ $j < i$ implies that there are only a finite number of unacknowledged messages with timestamps less than

or equal to $GVT_a(t_{i-1})$, either in the form of `SENT_MSG` communications in the FIFO's or in the unacknowledged message lists of the AP's. By P15 (finite delay) and Lemma 4, all of these get acknowledged in finite time. Analogous to P7 (update unreceived message time), *npGVT2P* has the property that when an AP_i receives a primary acknowledgment for a message, that message is marked accordingly and is considered as being deleted for future computations of υ_i. Thus, when AP_i receives a primary acknowledgment for a message with timestamp υ_i, the new value of υ_i is computed by excluding the message just acknowledged. This ensures that $\upsilon_k(t) \geq GVT_a(t_i) \;\; \forall t \geq t_i + \beta$ for some finite delay β. Consequently, $\forall t \geq MAX(\alpha, \beta)$, and the assumption is contradicted. ■

5.7 Properties of $GVT_c(t)$

In this section, we describe three interesting properties of the GVT computed by *npGVT*. For brevity, we discuss these only for *npGVT*, but each of these properties can be shown to be true for *npGVT2P* as well.

5.7.1 $GVT_c(t)$ is Non-decreasing

Criterion 1 for the correctness of a GVT computation algorithm requires only that $GVT_c(t) \leq GVT_a(t) \;\; \forall t$. Here, we show a stronger property about $GVT_c(t)$, namely that in addition to being less than or equal to $GVT_a(t)$, $GVT_c(t)$ is strictly non-decreasing (i.e. $t_1 > t_2 \Rightarrow GVT_c(t_1) \geq GVT_c(t_2)$). The GVT computation model for *npGVT* is shown in Figure 15. We see from the figure that $GVT_c(t) = GVT_p(t-\Delta)$. It is clear that if the values of $GVT_p(t-\Delta)$ at those times $t-\Delta$ when a reduction cycle is started are strictly non-decreasing, then $GVT_c(t)$ will be non-decreasing for all times t. We now show that this is the case.

Consider a change in $GVT_p(t)$ from one reduction cycle ($GVT_p(t_0) = G$) to the next ($GVT_p(t_0 + \delta) = \hat{G}$). For exposition, we refer to T-values contributing to $\hat{G}$ using the $\wedge$ symbol and to those contributing to G without it (i.e., $\sigma_i(t_0 + \delta) = \hat{\sigma}_i$, $\upsilon_i(t_0 + \delta) = \hat{\upsilon}_i$, $\sigma_i(t_0) = \sigma_i$ and $\upsilon_i(t_0) = \upsilon_i$). By definition, $\hat{G}$ is the minimum of the smallest $\hat{\sigma}_i$ and the smallest $\hat{\upsilon}_i$. In each case, we consider all possible ways in which these two T-values may be assigned.

Case (i): $\hat{G} = \hat{\sigma}_i$ for some *i*

- LP_i finished processing an event and computed $\hat{\sigma}_i$ from the events list. $\hat{\sigma}_i$ must be at least as large as σ_i since the events list is sorted in non-decreasing order. We have $G \le \sigma_i \le \hat{\sigma}_i = \hat{G}$.
- LP_i received a straggler or an antimessage from LP_j, causing a rollback. In this case, $\hat{\sigma}_i < \sigma_i$, but since antimessages are also acknowledged, the straggler will be outstanding at LP_j. Thus $G \le \upsilon_j = \hat{\sigma}_i = \hat{G}$.

Case (ii): $\hat{G} = \hat{\upsilon}_i$ for some *i*

- LP_i received an acknowledgment, and therefore set its $\hat{\upsilon}_i$ to the new smallest timestamp in its outstanding list. By P8 (unreceived is minimum) (Section 5.3.3.2), we have $\upsilon_i \le \hat{\upsilon}_i$ and since $G \le \upsilon_i$ we have $G \le \hat{G}$.
- LP_i sent a message and set $\hat{\upsilon}_i$ to the timestamp of the message. Here $\hat{\upsilon}_i < \upsilon_i$. However, since a message is sent only after executing an event (i.e., after setting σ_i), we have $\hat{\upsilon}_i = \sigma_i$ and $G \le \sigma_i$. Thus $G \le \hat{G}$.

5.7.2 GVT_c (*t*) Tracks GVT_a (*t*)

A necessary condition for GVT_c (*t*) to be correct is that it is never greater than GVT_a (*t*). An algorithm that estimates a lower bound on GVT satisfies this condition. The GVT computed by *npGVT* exhibits a much stronger property. From Figure 15 and the discussion on the operation of the reduction network in Section 5.5.1, it is clear that if GVT_a (*t*) (and hence GVT_p (*t*)) assumes at most one new value between successive reduction cycles, GVT_c (*t*) will assume exactly the same values as GVT_a (*t*), albeit lagging behind it. If GVT_a (*t*) assumes more than one new value between successive reduction cycles, then only the last one will be seen in GVT_c (*t*). In this case, GVT_c (*t*) assumes a subset of the values of GVT_a (*t*), once again lagging behind it. In either case, the values of GVT_c (*t*) are numerically the same as those of GVT_a (*t*) and not mere lower bounds.

5.7.3 GVT_c (*t*) Approaches GVT_a (*t*)

A corollary of the progress criterion (Section 5.3.4.2) is that if the simulation is halted at any time, GVT_c (*t*) will equal GVT_a (*t*) in some finite time. Assume that after the simulation has run for some time, all LP's stop their processing and retain the values of their

`local_clock`'s. With a reliable host communication network, all messages in transit will be received eventually and incorporated into the events lists of the receiving LP's. Now, since there are only a finite number of messages to be acknowledged, after some finite time, the υ_i of each LP will become ∞. At this point, GVT_p (t) will equal the minimum of the σ_i of all the LP's and after time Δ, GVT_c (t) will equal GVT_a (t). This property is particularly important when using *npGVT* in NPSI adaptive protocols because these protocols allow LP's to temporarily suspend execution of events. Specifically, it leads to an interesting and important feature of the NPSI adaptive protocol described in Chapter 6.

5.8 Summary

We have presented an example of computing NPSI using the reduction model of Chapter 4. The algorithm in this chapter (*npGVT*) computes GVT in aggressive protocols. We have proved the correctness of *npGVT*. We note that GVT computation is only an example — *npGVT* can be (and has been) extended to compute other kinds of information. *npGVT* is a foundation for the design of NPSI adaptive protocols. We have presented an implementation of *npGVT* on a hardware realization of the reduction model and proved its correctness. Finally, we have described three properties of the GVT computed by *npGVT* which benefit adaptive protocols.

Chapter 6

Elastic Time Algorithm

We describe the Elastic Time Algorithm (ETA), an NPSI adaptive protocol based on the general framework proposed in Chapter 4. A performance study of ETA is described. This study demonstrates clearly that ETA consistently outperforms Time Warp (both time and space) across a wide range of workloads. No other synchronization algorithm has been shown to be so successful. We consider the success of ETA a significant contribution to the PDES community.

We describe ZETA, a forerunner of ETA. While ZETA is an intuitive first choice for an NPSI protocol (others have proposed similar techniques), a performance study shows that it is not as effective as ETA.

Early work in this chapter appears in [SrRe95b].

6.1 Elastic Time Algorithm

Recall the general design framework for NPSI adaptive protocols described in Chapter 4. The framework consists of two mappings, M_1 and M_2. M_1 computes an error potential (EP_i) for each LP_i. M_2 translates EP_i into control over LP_i's optimism. Both M_1 and M_2 are evaluated dynamically at high frequency during the simulation. Based on this framework, an NPSI protocol can be described by specifying the two mappings M_1 and M_2.

6.1.1 M_1: Computing EP_i

Define the following:

σ_i = logical clock of LP_i — timestamp of the event either just executed or being executed currently.

υ_i = minimum unreceived message time of LP_i, i.e. it is the smallest among the timestamps of all messages sent by LP_i that have not yet been received and/or incorporated into the events list of the receivers.

η_i = next event time of LP_i, defined as: σ_i while LP_i executes an event; otherwise, the smallest among the timestamps of events scheduled to be executed at LP_i.

$\alpha_i = MIN(\eta_i, \upsilon_i)$

$\alpha_i' = MIN_{LP_k}(\alpha_i) \ \forall LP_k \in$ the predecessor set of LP_i

M_1 is the following function:

$$M_1: EP_i = MAX \{\eta_i - \alpha_i', 0\}$$

Note, α_i' constitutes the state information obtained dynamically using the reduction model (Section 4.2.1). This mapping is a variation of mapping (iv) in Section 4.3.1 — it uses α_i' to compute EP_i rather than GVT (or TSVT). We call α_i' the *minimum future time* (MFT) since it is the minimum (over the predecessor set) of the next event times and the timestamps of any messages in transit. Thus α_i' can be thought of as representing the time of the next event relevant to LP_i (i.e. the next event that can affect LP_i). Note, υ_i is included in computing α_i' since a message in transit may ultimately cause an event to be scheduled at LP_i. Since η_i is LP_i's next event time, the difference between the η_i and α_i' is an indicator of how far ahead (in logical time) LP_i's next scheduled event is of the next event that can possibly affect LP_i. It is interesting to note the relationship between TSVT and α_i':

$$TSVT_i = MIN_{LP_k}(\sigma_i, \upsilon_i) \ \forall LP_k \in \text{the predecessor set of } LP_i$$

$$\alpha_i' = MIN_{LP_k}(\eta_i, \upsilon_i) \ \forall LP_k \in \text{the predecessor set of } LP_i$$

From the above definitions, we see that $TSVT_i$ represents the current state (σ_i) of the simulation while α_i' represents the immediate future (η_i). Intuition suggests, therefore, that α_i' will prove more effective in controlling optimism.

Recall that mapping (iv) of Section 4.3.1 has the drawback that the LP with the next *safe* event (i.e. the event that is guaranteed not to be rolled back) is unable to determine this fact. Potentially, this leads to arbitrarily long periods of unnecessary waiting. The use of α_i' instead of $TSVT_i$ in ETA solves this problem. $\alpha_i' > \eta_i$ implies there are no impending events that will affect LP_i's next scheduled event — i.e., LP_i's next event is safe. Correspondingly, the mapping above sets EP_i to zero in this case, effectively terminating any waiting LP_i may be engaged in.

The intuition behind ETA's M_1 is that if an LP is far ahead of others, it is likely to be rolled back soon and therefore should be slowed down. The name ETA derives from the following analogy: one may imagine an LP and its predecessors as pins moving along the logical time line with an elastic band around them. The farther an LP moves away from the rest, the slower its progress due to the restraining pull of the elastic band. When the LP farthest behind moves forward, the restraint on the LP farthest ahead is reduced so that it may quicken its pace again.

6.1.2 M_2: Controlling Optimism

Given a value of EP_i computed by M_1, M_2 is the following linear function:

$$M_2: \delta_i = s \cdot EP_i$$

where *s* is a scaling factor (we defer discussion on *s* to Chapter 8). The event processing loop of an LP using ETA (i.e., the event processing loop of an LP using Time Warp modified to incorporate M_1 and M_2) is shown in Figure 24. Some points to note:

```
9.    σi = 0; ηi = timestamp of first scheduled event
10.   While simulation is not complete
11.     σi = ηi; Process next event{schedule future events and send messages as
            required}
12.     Compute new ηi
13.     Start wait timer
14.     do
15.      Receive and buffer messages
16.      Exit from loop if
17.             there is a message that will cause rollback
18.             or if the message has timestamp = TSVT
19.      Process a message (if any) and recompute ηi
20.      Update αi´, EPi and δi
21.      Read wait timer
22.     while wait timer value < δi
23.     If necessary, roll back and recompute ηi
24.     Process messages and recompute ηi
25.     Save state
26.     Collect fossils
27.   Endwhile
```

Figure 24 - Event execution loop of ETA

i) The blocked state is not opaque in that while in this state, LP_i receives messages from its input channels (line 7) and checks for two kinds of messages: (lines 8-10): (i) a message that may cause it to roll back, and (ii) a message with timestamp $TSVT_i$. In each case, an action is warranted immediately — a rollback in the former and processing the message with timestamp TSVT (since there can be no messages with timestamps < TSVT) in the latter — and therefore, the waiting is aborted.

ii) LP's process messages in the blocked state. This reduces lost opportunity cost in the following situation. Assume an incoming message at LP_i has a timestamp smaller than η_i. Since this message may schedule an event at LP_i, η_i may decrease in processing the message, thus reducing EP_i (which would have been abnormally high if the message had not been processed — resulting in lost opportunity cost). Also, processing messages as soon as possible reduces the time for which they are considered unreceived and thus accelerates the progress of α_i'.

iii) EP_i and δ_i are updated frequently in the blocked state.

iv) The waiting scheme is not memoryless. The wait timer is started only once at the beginning of the blocked state. While LP_i is in the blocked state, η_i is constant while α_i' increases monotonically (shown in Appendix III). Thus, δ_i decreases. LP_i interprets each new value of δ_i as an estimate of the amount of time it should have waited since the start of the waiting period. When this value becomes smaller than the time it has actually waited, LP_i exits the blocked state.

These properties of M_2 suggest a simple loop structure for the blocked state, as shown in Figure 24 (lines 5-14). This M_2 provides direct control over an LP's aggressiveness only. The LP's risk is controlled only to the extent that while in a blocked state, the LP does not send out messages. Clearly, it is possible (and perhaps desirable) to have a separate mechanism to limit risk.

6.1.3 Properties of ETA

ETA is adaptive because the aggressiveness and risk of LP's can change dynamically. ETA follows what we call the *microadaptive* approach wherein the adaptiveness is fine-grained — it occurs after each event execution. In contrast, most of the previously proposed protocols that limit optimism follow a *macroadaptive* approach wherein the control settings

are fixed for significant periods of time and changed between periods. A microadaptive approach is inherently more capable of reacting efficiently to rapid changes in system behavior. For example, it is well known that most of the activity in a logic simulation is localized to a small part of the circuit and this locality changes (rapidly) as the simulation proceeds. In this case, the LP's simulating the part of the circuit where the activity is currently concentrated would be heavily loaded and would thus simulate at a slower rate while the remaining LP's would be lightly loaded and would thus surge ahead in logical time, simulating events that will be rolled back, in all likelihood. As the load locality changes among LP's, ETA adapts by restraining those LP's currently far ahead in logical time (i.e. the lightly loaded ones).

ETA is a hybrid between the optimistic and conservative approaches in that, based on the value of *s*, it can mimic either approach. When *s*=0, δ_i=0 irrespective of the value of EP_i. In this case, each LP executes exactly one iteration of the wait-loop in Figure 24 in each iteration of the main event execution loop and thus, ETA is essentially identical to Time Warp. Consider very large values of *s*. As *s* tends to infinity, δ_i becomes very large even for small values of EP_i. However, as noted earlier, the LP (LP_i) with the next safe event has $EP_i = 0$, and consequently, $\delta_i = 0$. LP_i terminates its waiting, executes its next event and then recomputes η_i. This new value of η_i is subsequently incorporated into the α's of LP_i's successors so that the LP (LP_j) with the next safe event computes $EP_j = 0$, terminates its waiting and executes its next event. Although it appears as if events are being executed sequentially, the use of target-specific information to compute the error potential implies multiple LP's may execute events concurrently. In our example above, it may happen that LP_i's new next event is also safe and thus, LP_i may execute this event concurrently with LP_j's event execution. Thus, events are executed in a manner similar to that in the Null-Messages conservative protocol [ChMi79]. It is interesting to note that common enhancements to reduce lost opportunity costs in conservative protocols (such as the use of lookahead [Fuji90]) can be applied to ETA as well, by incorporating them into M_1.

Putting ETA in perspective with previous protocols, we observe that it is similar to window based protocols with two significant differences:

- It is completely asynchronous - there are no barrier synchronizations to negotiate windows;
- Each LP's logical time window may be considered infinite but the event execution rate decreases as the LP moves farther away from the base of the window.

6.2 Implementation Issues

ETA requires each LP to maintain two values: η_i and υ_i. These values are used by the feedback system to provide α_i' to each LP_i. By definition, η_i equals σ_i, the logical clock of LP_i when LP_i executes an event. Since the GVT algorithm of Chapter 5 (*npGVT*) maintains σ_i and υ_i correctly, it can be extended to maintain η_i as well. We observe in Figure 24 that η_i can change whenever any of the following occur: (i) an event execution completes (this event may have scheduled future events), (ii) a rollback occurs (the current event changes), and (iii) a message is received (the message may schedule a future event). η_i may be maintained correctly by executing the following statement:

```
if there is a next event, η_i = timestamp of next event; else η_i = ∞
```

at the following three points in *npGVT* (Figure 14): (i) after executing an event (and scheduling any future events and/or sending messages as required) in function `PROCESS`, (ii) after rolling back and restoring the state in function `ROLLBACK`, and (iii) after processing each message in function `RCVMSG`. In the case of *npGVT2P* (Figure 18), the host processor executes the above statement at the corresponding places and sends the value of η_i to the auxiliary processor.

As noted in Section 4.2.2, a target-specific reduction model can be implemented using a target-specific reduction network. However, since such a network does not exist yet, we emulate it using a global reduction network using a scheme proposed in [Panc94]. This scheme allocates a portion of the global OSV to each LP_i. Only those LP's in the predecessor set of LP_i submit values into the corresponding portions of their ISV's — all other LP's submit identity values that do not contribute to the reductions (for example, ∞ for a MIN operation).

6.3 Performance Analysis

We describe a two-part performance study of ETA. The first part was conducted using an implementation of ETA on a four-processor system, while the second part focused on demonstrating scalability using a simulation of systems with larger numbers of processors.

6.3.1 Performance Criteria

We use the two performance measures described in Section 4.1.2 — performance relative to Time Warp and minimization of residual cost. Recall the cost function from Section 4.1.1 defining the trade-off in Figure 10:

```
total cost = rollback cost +
             memory management cost +
             lost opportunity cost
```

Memory management schemes are required in Time Warp because LP's consume excessive amounts of memory due to uncontrolled optimism. As demonstrated by results presented here, controlling optimism significantly reduces memory consumption so that the need for memory management schemes may be eliminated (except in the special case of source LP's). Consequently, we focus on rollback and lost opportunity costs — our protocols do not incorporate memory management currently. Memory management in NPSI adaptive protocols is a topic for future research. The cost function reduces to:

```
total cost = rollback cost +
             lost opportunity cost
```

Thus, the second performance measure is the effectiveness of a protocol in eliminating rollback cost without adding significant lost opportunity cost.

6.3.2 Testing Environment

The hardware consists of a cluster of four Sparc 2 workstations connected by Ethernet and a *parallel reduction network* (PRN) that we have designed and built [RePS92]. Each workstation communicates with its own auxiliary processor (AP) through dual-ported memory (DPRAM). The four AP's are connected to the PRN which computes and disseminates globally reduced values at very high speeds. The AP's are responsible for sending values into the PRN and reading its global output. Dedicated processors have been used to off-load this task from the workstations so as to keep interference with the simulation to a

minimum. The workstations communicate relevant state changes to the AP's through the DPRAM. The AP's process this information and send it to the PRN. Also, the AP's read the globally reduced output of the PRN and propagate relevant parts of it to the workstation.

The two primary software components are the event execution loop for ETA (Figure 24) and the algorithms to compute NPSI. The event loop executes on each of the four workstations. The three T-values (σ_i, υ_i and η_i) are maintained by *npGVT2P*, modified as described earlier (Section 6.2). The portion of *npGVT2P* to be executed by the workstations is integrated into the event execution loop for ETA.

6.3.3 Test Cases

As noted in Section 6.3.1, a measure of success for ETA is to consistently outperform Time Warp over a wide range of workloads that include those for which Time Warp performs well and those for which it does not. Unfortunately, there is no standard set of benchmarks to test the performance of PDES protocols. While some protocols have been tested over a range of applications over time by several researchers, the typical approach to performance analysis in the PDES community has been to study the performance of a proposed protocol on one or two applications (most often, queueing networks). We believe a protocol must be tested on several kinds of workloads to demonstrate its effectiveness.

In terms of performance of Time Warp, the characteristics of an application (or workload) that are important are event execution times, message destination probabilities, load distribution, message latencies, etc., since they determine rollback behavior and memory consumption. In general, Time Warp performs well when the event grain is large compared with overheads such as message latencies (which determine the cost of a single rollback) and when load is balanced. When the cost of a rollback is comparable with the time to execute events, Time Warp can become unstable [LuWS89, Fuji90]. Similarly, imbalanced load can lead to excessive rollbacks and/or memory consumption, both of which can degrade performance severely. Thus, a performance study should focus on these characteristics of applications, rather than their semantics.

6.3.3.1 Synthetic Workload Generator

We employ a parameterized synthetic workload generator to realize our test cases. Synthetic workloads are used instead of actual applications because they allow us to mimic applications by capturing their essential characteristics (as described above) without the excessive time and effort required to implement each of them. Synthetic workloads have been used in previous performance studies in PDES. Our workload generator supports a wider range of workloads than the PHOLD workload model of [DaFu94].

The important parameters that can be changed to realize different workloads are event execution time, average timestamp increment, state saving cost, communication topology and distributions, number of local events per message, and state saving and fossil collection frequencies. Time consuming application-specific actions such as event executions and state saving are simulated by busy loops; all other mechanisms such as rollback, restoring state, sending antimessages and fossil collection and data structures such as saved state list and antimessage list are implemented in detail.

There are two basic modes of operation: *message initiating* and *self initiating.* It is possible to combine these two modes in a single workload. In the message initiating mode, each LP contains some number (possibly different for each LP) of JOB events initially. Execution of a JOB event schedules some LOCAL events, followed by exactly one OUTPUT event. A LOCAL event represents local activity in the LP — it does not affect other LP's directly. It is simulated by a busy-loop. The number of LOCAL events scheduled per JOB event is distributed normally about a mean parameter, which can be zero. Execution of an OUTPUT event sends a message to another LP. The selection of the destination of the message can be random (a distribution is specified over a set of possible receivers) or deterministic (the trajectory of the message is exactly specified). Several classes of deterministic messages can be specified, each with a different trajectory. Arrival of a message schedules a JOB event. Thus, the resulting system is a closed one. It is possible to simulate an open system by defining *sources* and *sinks*. A source is a generator of messages — each OUTPUT event at a source schedules a JOB event locally. A sink consumes messages — an OUTPUT event at a sink does not result in a message. In the message initiating mode, our workload generator is similar to the PHOLD model [DaFu94]. This mode is used to simu-

late applications such as logic simulations, communication networks, battlefield simulations and queueing networks.

A self-initiating system is inherently an open one in which an LP schedules its own next event. Each LP contains a single SELF event initially. Execution of a SELF event involves three activities: (i) execution of a busy-loop to simulate local activity, (ii) sending a message with some probability, and (iii) scheduling the next SELF event. If a message is sent out, the destination is selected randomly from a set of receivers based on a specified distribution. A message does *not* schedule an event at the receiving LP — however, it may cause a rollback. This mode is used to simulate applications such as the Ising spin model [Luba89].

Other parameters that determine the nature of the workload are:

- *event delay* — the execution time of an event — normally distributed about a mean. The mean can be different for each LP;
- *state saving delay* — the time to save state — normally distributed about a mean;
- *state saving frequency* — the number of events between successive state saving operations;
- *fossil collection frequency* — the number of events between successive fossil collection operations;
- *timestamp increment* — the mean increment in timestamp between a pair of successive events — uniformly or exponentially distributed;
- *maximum simulation time* — the maximum logical time to be simulated.

6.3.3.2 Workloads

We have tested ETA on many workloads using the workload generator described above. We describe a suite of workloads for which performance results are presented in Section 6.3.5. Figure 25 shows the different topologies used in the workloads. A number associated with an arc is the probability that a generated message is sent along that arc. A short mnemonic is associated with each workload in this suite for ease of reference.

a) **LGRAIN** is a message initiating workload with the T_1 topology (a torus) and 25 messages per LP initially. The event delay is 1 millisecond and the mean number of

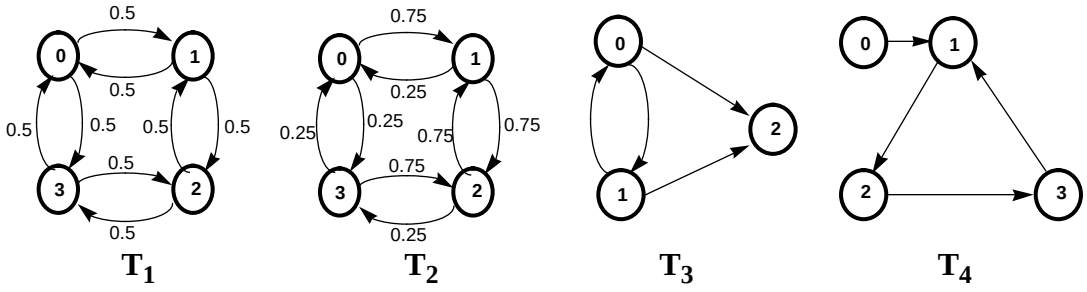


Figure 25 - Topologies for workloads

local events per message sent is 5. Due to the large event grain and high local activity, this workload represents a typical simulation where Time Warp performs well (such as battlefield, traffic and computer network simulations).

b) **SELF-SYM** is self initiating with topology T_1. The probability of sending a message after each event is 0.2. The event delay is 100 microseconds. It simulates the Ising Spin model [Luba89].

c) **SELF-BIAS** is similar to SELF-SYM but uses topology T_2 rather than T_1. Since this topology is biased, it is expected to increase rollback costs.

d) **MSG-SYM** is a message initiating workload using topology T_1. No local events are simulated — each JOB event schedules only the corresponding OUTPUT event. The event delay is 100 microseconds. This workload is similar to a uniform queueing network. There are 25 events at each LP initially.

e) **MSG-BIAS** is similar to MSG-SYM but uses topology T_2 rather than T_1. The bias in the topology mimics hot-spot activity in interconnection networks.

f) **ASYM** is similar to MSG-SYM (message initiating, topology T_1 and no local messages). The mean event delay for LP_0 and LP_3 is 2 milliseconds while that for LP_1 and LP_2 is 100 microseconds. Thus, this workload simulates load imbalance where some LP's execute events at a slower rate than others.

g) **ECHO** is an implementation of the *echoing* example described in [LuWS89]. It uses three LP's with T_3 topology. LP_0 and LP_1 execute in self-initiating mode, sending messages to LP_2 after each event. The mean event delay is 100 microseconds. In addition, a single message-initiating event is at LP_0 initially. After processing this event, LP_0 sends a message to LP_1. This causes LP_1 to schedule an event, possibly after rolling back. After processing this event, LP_1 sends a message back to LP_0 and

the cycle repeats. Thus, this *causal* message rolls back its receiver and while the receiver is rolling back, the sender simulates ahead, sending erroneous messages to LP_2. Thus, LP_0 and LP_1 spend increasing amounts of time rolling back and the progress of the simulation diminishes rapidly. Clearly, this is a stress case for Time Warp. This workload is inherently unstable.

h) **CHASE** represents a second kind of adverse rollback behavior experienced by Time Warp, in which erroneous messages outpace the corresponding antimessages. The workload is message initiating with topology T_4. LP_0 is a slow source of messages — it sends a message every 10 milliseconds.These messages serve only to roll LP_1 back and are consumed by LP_1. LP_1, LP_2 and LP_3 execute events at a fast rate (event delay = 100 microseconds). Initially, each of LP_1, LP_2 and LP_3 has 5 events. The erroneous computation proceeds in the form of these 5 events circulating among the three LP's. A message from LP_0 causes LP_1 to roll back and start a chain of antimessages that chases the erroneous computation around the cycle.

i) **PHASES-MSG** is a message initiating workload in which the LP's undergo alternating phases of slow and fast execution. The workload uses topology T_1. LP_0 and LP_3 begin executing in the slow phase while LP_1 and LP_2 begin executing in the fast phase. A slow phase is followed by a fast phase and vice-versa. A slow phase consists of 10 events, each with a mean delay of 1 millisecond, while a fast phase consists of 100 events, each with a mean delay of 100 microseconds. This type of phase behavior is commonly observed in practice — for instance, in logic simulations — where the load is represented by a set of events that moves over the set of LP's.

j) **PHASES-SELF** is similar to PHASES-MSG but uses the self-initiating mode instead of the message initiating mode. The probability of sending a message after an event is 0.25.

A torus was used because it is widely used in the PDES literature and because it provides sufficient connectivity to introduce rollback costs. For all workloads, the mean state saving time is 25 μs and state saving and fossil collection are performed after each event.

6.3.4 Metrics

Recall the cost function of Section 6.3.1: total cost=rollback cost+lost opportunity cost. Rollback cost can be measured directly by the rollback time, which is the time an LP spends rolling back, including state restoration and sending of antimessages. Since limiting of aggressiveness is expected to reduce rollback time, ideally to zero, this metric is a good indicator of how close the actual performance is to the goal. Reduction in rollback time has the effect of reducing the completion time of the simulation, which reflects the reduction in total cost. On the other hand, lost opportunity costs introduced by limiting optimism tend to increase completion time. Thus, the completion time is another important metric of the effectiveness of a protocol. Other metrics of rollback behavior are: number of rollbacks, and average depth of rollback (number of events rolled back on average).

Memory in aggressive systems can be of three types:

- processed and committed — this is managed by fossil collection;
- processed and uncommitted — this is created due to uncontrolled aggressive processing and therefore can be managed by limiting optimism;
- unprocessed — this is created by *runaway LP's* (unthrottled LP's sending messages to others and consuming their memory to store these messages) and can be managed by a memory-based flow control mechanism.

Processed and uncommitted memory is measured by the maximum and average sizes of the saved-state list (in terms of the number of entries in the state list). The maximum size is important because the workstation must have sufficient memory to store that much state even if it is a rare occurrence. While it is possible to incorporate memory-based flow control into NPSI protocols (mapping (ix) in Section 4.3.1), ETA does not do so currently. Therefore, we do not measure unprocessed memory.

The slope *s*, in M_2 translates the value of EP_i from logical time units to a delay in real-time. The range of EP_i is dependent on the logical time increments exhibited by the LP's and the rate at which they execute events, send messages, etc. Since these factors differ considerably across applications, it is expected that the value of slope that maximizes performance will be different for each application. Thus, slope is our choice for the independent variable in the performance tests.

6.3.5 Performance Results

For each of the workloads, we present performance results using the metrics described above. In addition, we plot the average time for which an LP waits after each event. Excluding completion time, all metrics are for a single LP. Recall ETA is essentially identical to Time Warp when slope = 0. In graphs that use a logarithmic scale for the slope, the slope-co-ordinate of the origin is artificially set to a non-zero value. This is necessary since a logarithmic scale reaches zero only at $-\infty$.

6.3.5.1 LGRAIN

Figure 26 shows the performance of the LGRAIN workload. Since the event grains are fairly large and the mean number of local events between messages is five, we observe that rollback costs are small. The completion time exhibits the expected parabolic shape due to the trade-off shown in Figure 10 (Section 4.1.1). Two important points must be noted: (i) the rollback time is nearly zero at the point where completion time is minimized (slope = 5), and (ii) the maximum reduction in completion time (~10 seconds at slope = 5) is greater than the reduction in rollback time (at slope = 5). This implies that ETA is able to eliminate most of the rollback cost before any appreciable lost opportunity cost is introduced. Further, since rollback cost is nearly zero, the performance of ETA is close to the best possible performance for protocols with controlled optimism, as defined in Section 6.3.1. The reduction in maximum state list size is significant. As slope increases beyond the point at which completion time is minimized (slope=5), we see that the maximum state list size tends to increase. This is because the rate of progress of TSVT is decreased in this region due to high lost opportunity cost. This implies that state list entries are converted into fossils at a slower rate, resulting in longer state lists.

Parts (b) and (d) show the effectiveness of ETA in reducing rollback costs. Note, at slope=5 the number of rollbacks is reduced only by about 33% while the average rollback depth is close to 1, the minimum depth. Thus, most of the benefit of ETA appears to be from reducing the rollback depth rather than the number of rollbacks. This is as expected since completely eliminating rollbacks requires either conservative processing (ETA approaches this at slope=1000) or perfect guessing (which is not in ETA's design). Further, these results

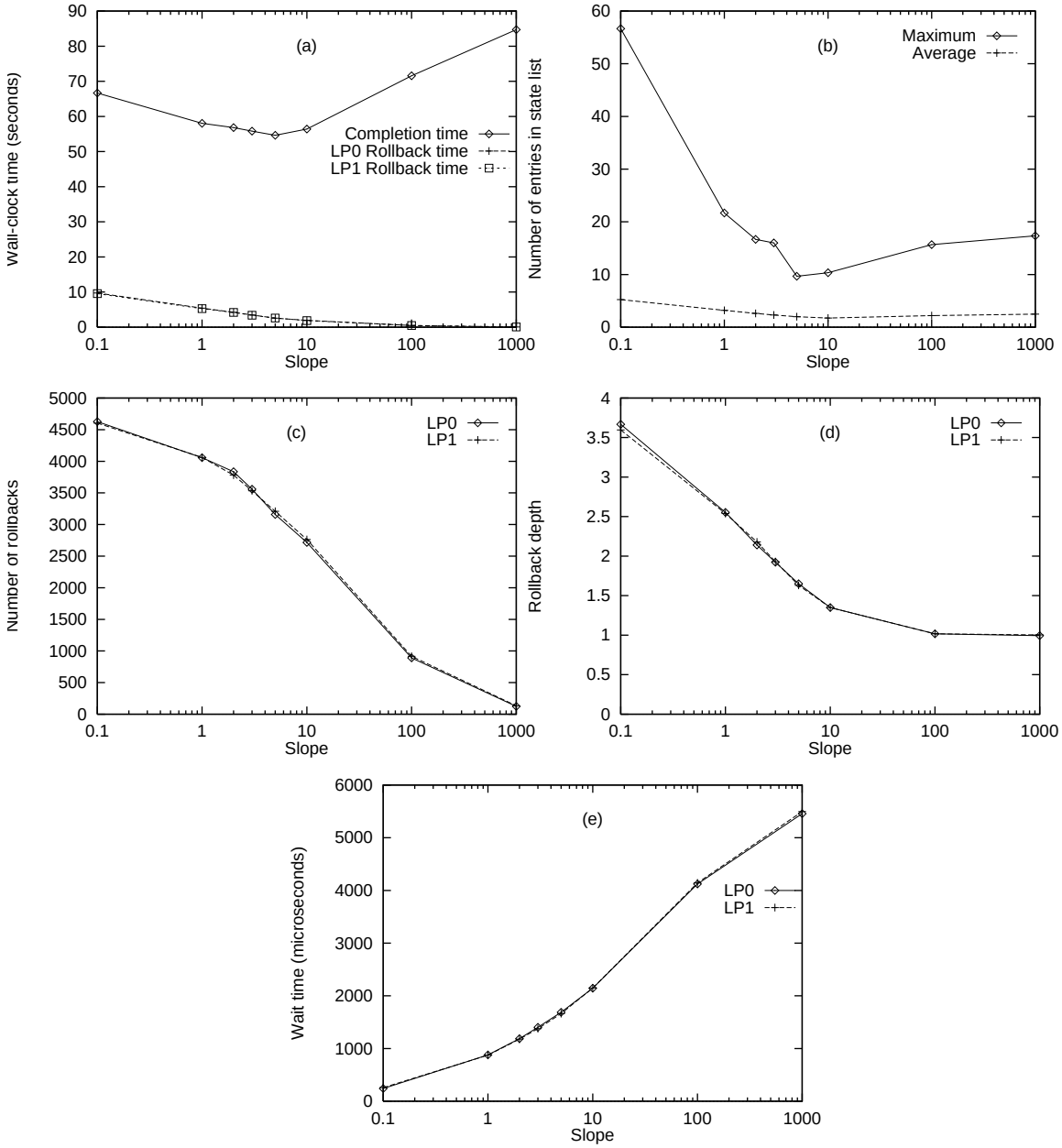


Figure 26 - Performance for LGRAIN

support the intuition that the benefits from eliminating rollbacks completely (as in a conservative approach) are far outweighed by the resulting lost opportunity cost.

Part (e) shows the average wait time for two LP's. Note, in parts (c) through (e), the two curves in each graph (for LP_0 and LP_1) are nearly identical — this is because the LGRAIN workload is symmetric.

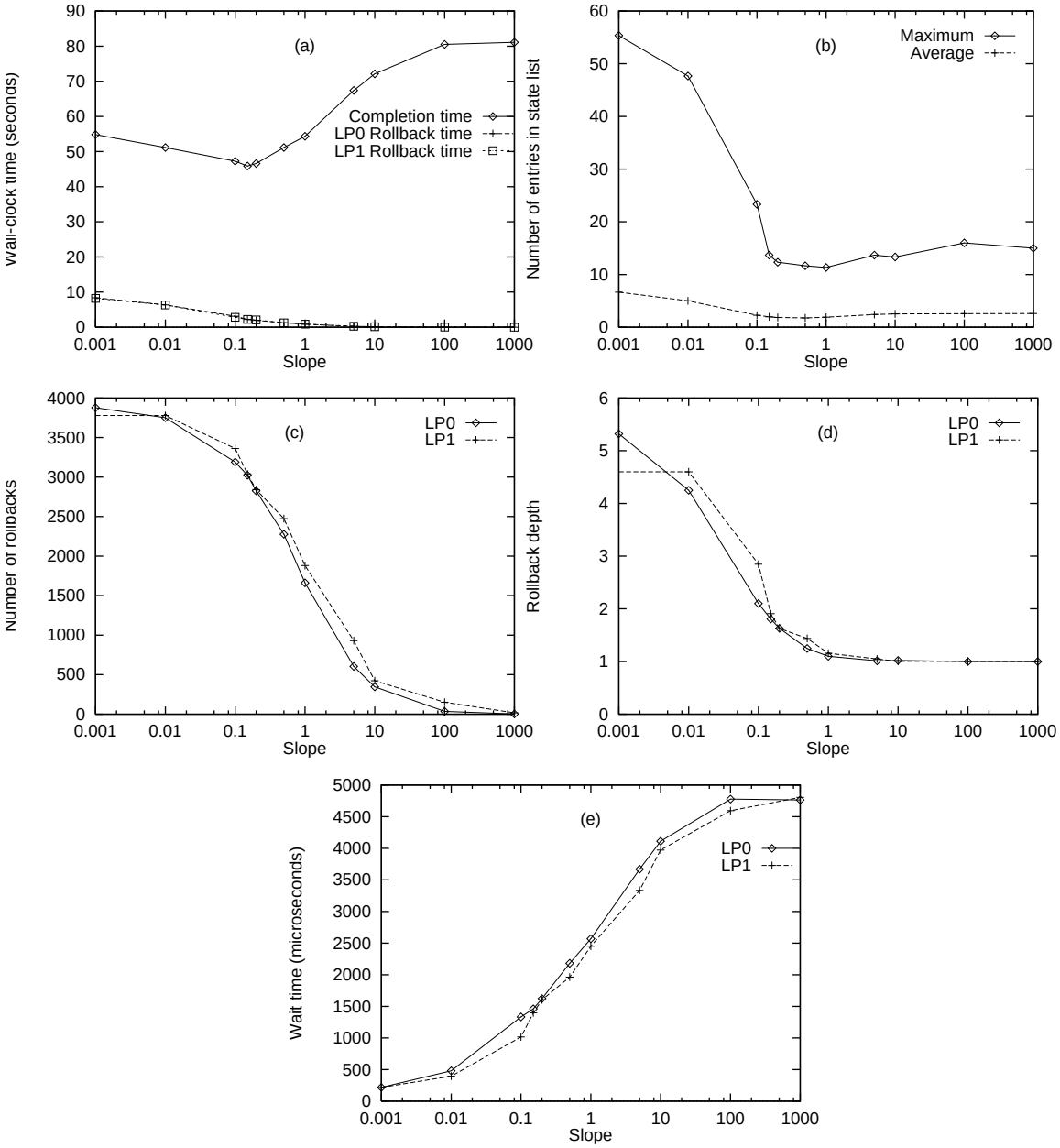


Figure 27 - Performance for SELF-SYM

6.3.5.2 SELF-SYM

Figure 27 shows the performance of ETA on the SELF-SYM workload. Although the event grain in SELF-SYM is smaller than that in LGRAIN, the probability of sending a message after executing an event in SELF-SYM is much smaller — 0.2. This results in significant concurrency in the workload and therefore, low rollback costs. In this respect, SELF-SYM is similar to LGRAIN. Accordingly, we observe that the performance of ETA

with SELF-SYM is similar to that with LGRAIN. It is interesting to note that ETA reaches *saturation* (i.e. the curves begin to flatten) around slope=100 with SELF-SYM while there is no indication of saturation with LGRAIN even at slope=1000. As explained earlier, the range of values over which the slope affects performance depends on the characteristics of the application, in particular on the timestamp increments and event grain. The observed difference in operating range is attributed to the different event grain sizes in SELF-SYM and LGRAIN.

6.3.5.3 SELF-BIAS

Figure 28 shows the performance of ETA with SELF-BIAS. Since the workload is biased with messages being concentrated at LP_1 and LP_2, these LP's tend to stay synchronized while LP_0 and LP_3 do not. Accordingly, the rollback time for Time Warp is greater for LP_0 than for LP_1. From part (c) we observe that LP_0 has fewer rollbacks than LP_1, but these rollbacks are significantly deeper. Since, as mentioned before, ETA is most effective in minimizing the depth of rollbacks, the reduction in completion time in part (a) appears to be due to the significant reduction in the rollback depth at LP_0 (and LP_3). Again, at the point of minimum completion time (slope=0.1) we observe that the reduction in completion time (~16 seconds) is greater than the reduction in rollback costs (~9 seconds), implying very low lost opportunity cost. The memory performance with SELF-BIAS is similar to that with previous workloads. Finally, note the difference in average wait times for LP_0 and LP_1 (part (e)) is greatest at slope=0.1, the point at which completion time is minimized.

6.3.5.4 MSG-SYM

Figure 29 shows the performance of ETA with MSG-SYM. The absence of the characteristic parabolic shape in the completion time curve is attributed to the lack of sufficient concurrency in this workload. This is due primarily to two reasons: (i) a message is sent after each event — since the event grain is small (100 μs) while the communication latency is high (1-2 ms), the computation-to-communication ratio is too small to support concurrency, and (ii) there are only four processors. Recall for large values of slope, ETA behaves similar to a conservative protocol. Since the communication graph is strongly connected and there is no lookahead, a conservative protocol will effectively execute this simulation sequentially (albeit with messaging latencies and other overheads). Therefore, we observe

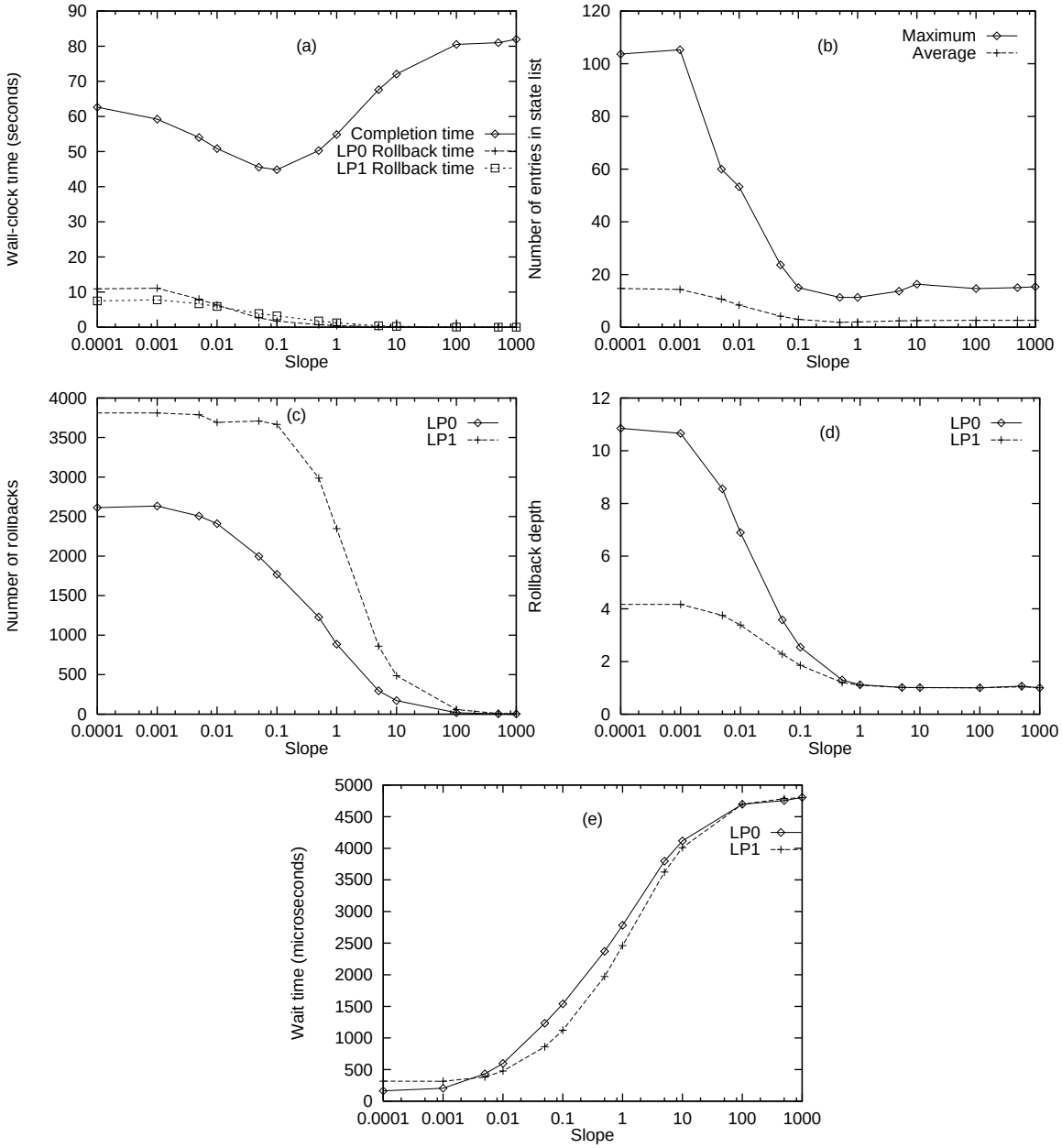


Figure 28 - Performance for SELF-BIAS

---

that the completion time does not increase beyond the minimum. The memory performance with MSG-SYM is similar to that with previous workloads. Since the workload is symmetric, we observe in parts (c) and (d) that the curves for LP_0 and LP_1 are nearly identical. Once again, it is clear that the rollback depth is minimized (slope=10) before the number of rollbacks is reduced significantly (slope=~50).

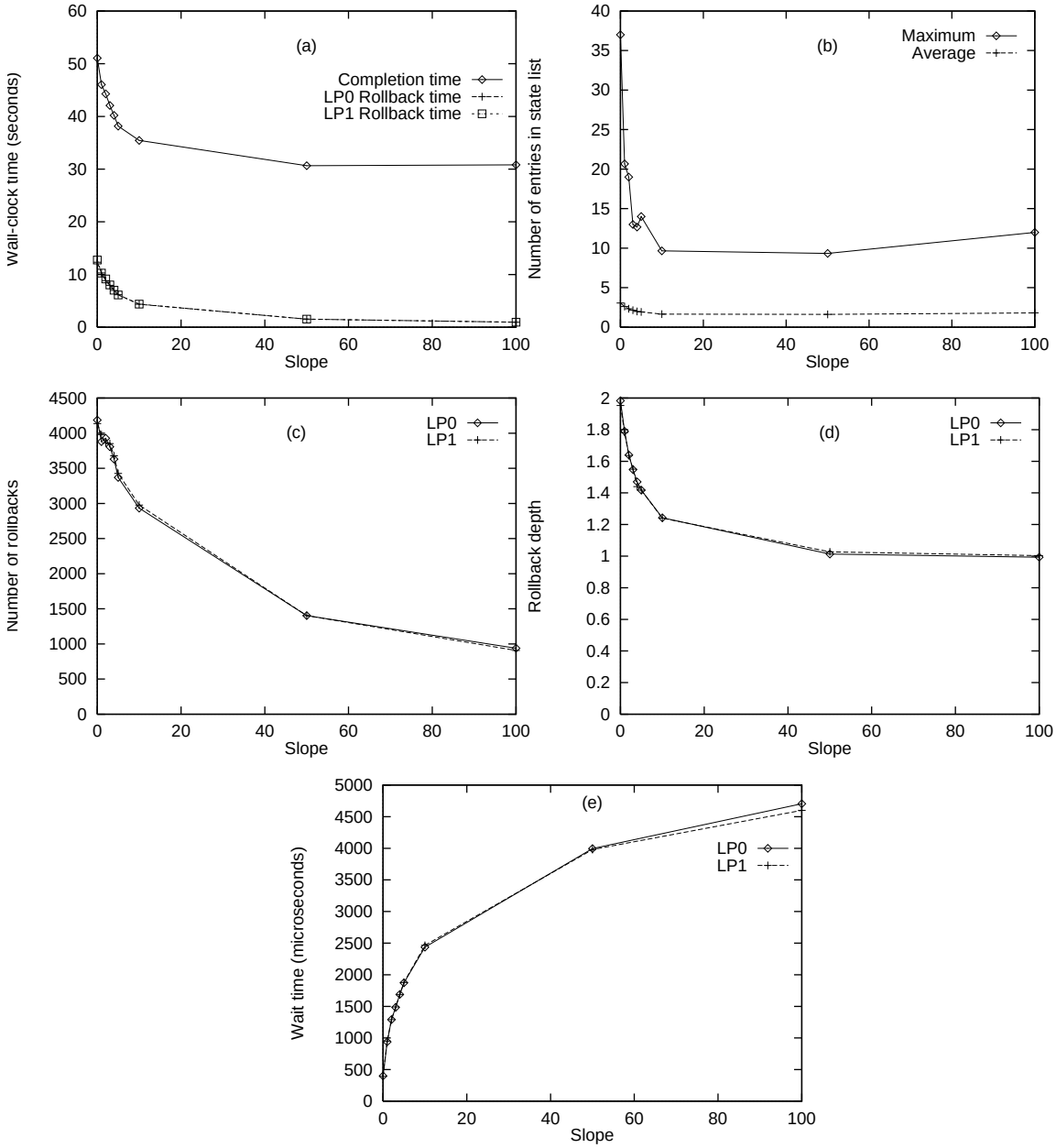


Figure 29 - Performance for MSG-SYM

6.3.5.5 MSG-BIAS

Figure 30 shows the performance of ETA with MSG-BIAS. The rollback cost for Time Warp is significantly higher with MSG-BIAS than with MSG-SYM, due to the bias in the workload. ETA is able to eliminate nearly all of this rollback cost. As with MSG-SYM, the lack of concurrency in the workload explains the absence of a parabolic shape to the completion time curve. Again, at the point of minimization of completion time (slope=50), the

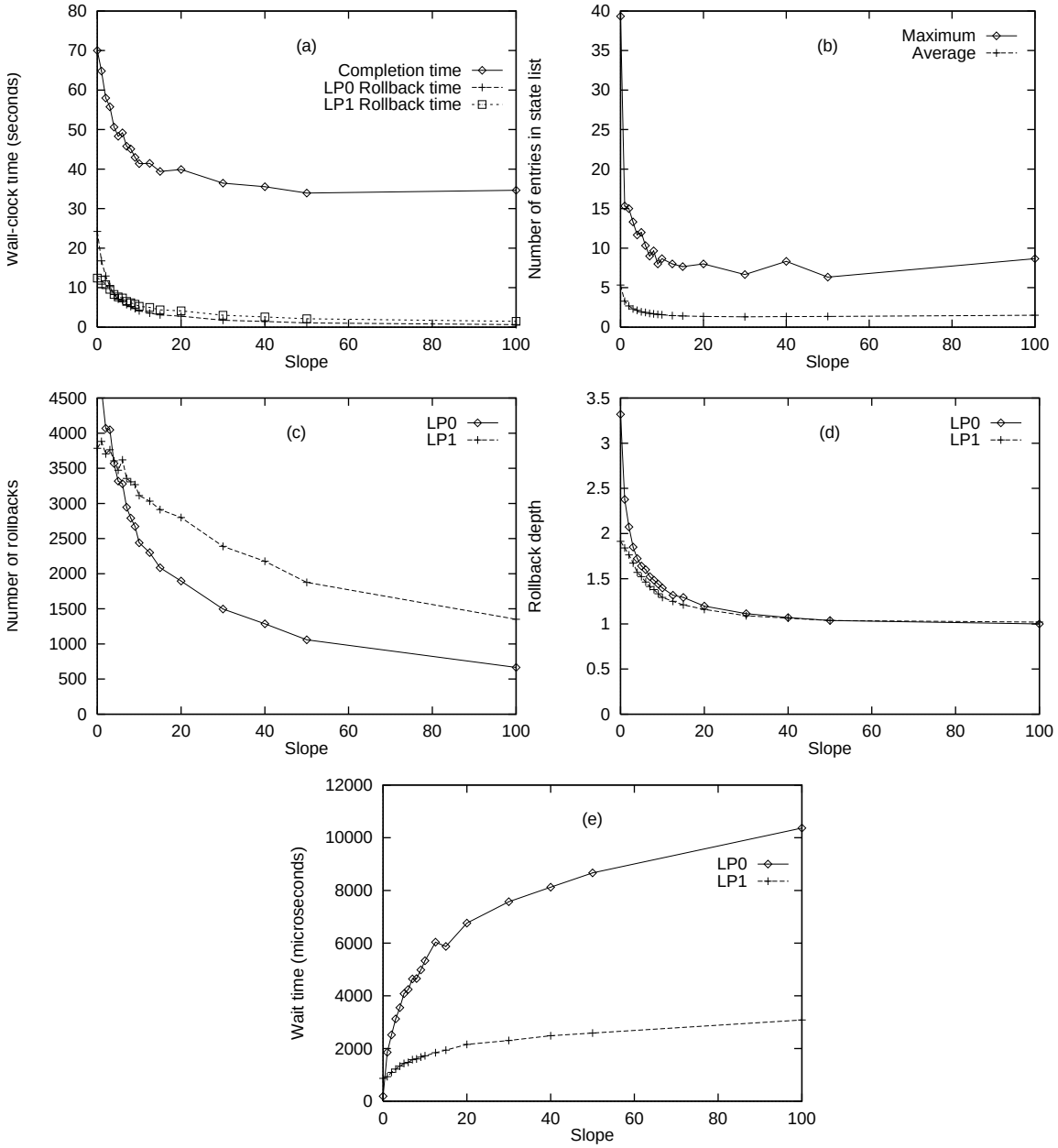


Figure 30 - Performance for MSG-BIAS

rollback time is nearly zero and the decrease in completion time (~34 seconds) is greater than the decrease in rollback time (~25 seconds). The effect of the bias is apparent in the rollback behaviors of LP_0 and LP_1 as seen in parts (c) and (d). It is interesting to note that even though the rollback depth is reduced dramatically around a slope of 20, the completion time is minimized only after the number of rollbacks decreases significantly. As noted earlier, this is because there is insufficient concurrency in the workload and a conservative

approach works best. Part (e) clearly demonstrates the adaptive nature of ETA — since the bias in the workload was chosen arbitrarily and ETA was not tailored in any way to compensate for this bias, this graph shows that ETA adapts to the load imbalance by throttling LP_0 (and LP_3) more than LP_1 (and LP_2). The performance of ETA with the PHASES workloads further illustrates this point.

6.3.5.6 ASYM

Figure 31 shows the performance of ETA with ASYM. The performance is similar to that with MSG-SYM, since the two workloads are similar. The effect of the asymmetry in the workload is apparent as the separation in the curves for LP_0 and LP_1 in parts (c) through (e). While this separation is not as significant as with MSG-BIAS, it is noticeably greater than with MSG-SYM.

6.3.5.7 ECHO

Figure 32 shows the performance of ETA with ECHO. ETA produces substantial reductions in completion time and rollback time. Recall that ECHO is inherently unstable in that the amplitude of rollbacks increases without bound, reducing the efficiency of the simulation to zero asymptotically. By limiting optimism, ETA eliminates this instability (see Chapter 7 for a proof). Again, the absence of the characteristic parabolic shape to the completion time curve is due to the inherently sequential nature of the workload — the only significant activity in the simulation is the exchanging of the single message between LP_0 and LP_1. ETA produces correspondingly significant reductions in memory consumption as well. Memory consumption is high in ECHO because while one of LP_0 and LP_1 rolls back, the other simulates aggressively far into the logical future. Limiting optimism reduces this memory consumption. In part (d) we observe that the average rollback depth is exceedingly high. As mentioned, this is because of the phenomenon of *echoing*, where the depth of rollbacks increases without bound.

6.3.5.8 CHASE

Figure 33 shows the performance of ETA with CHASE. Recall, CHASE is designed to create a situation where erroneous messages outpace their corresponding antimessages.

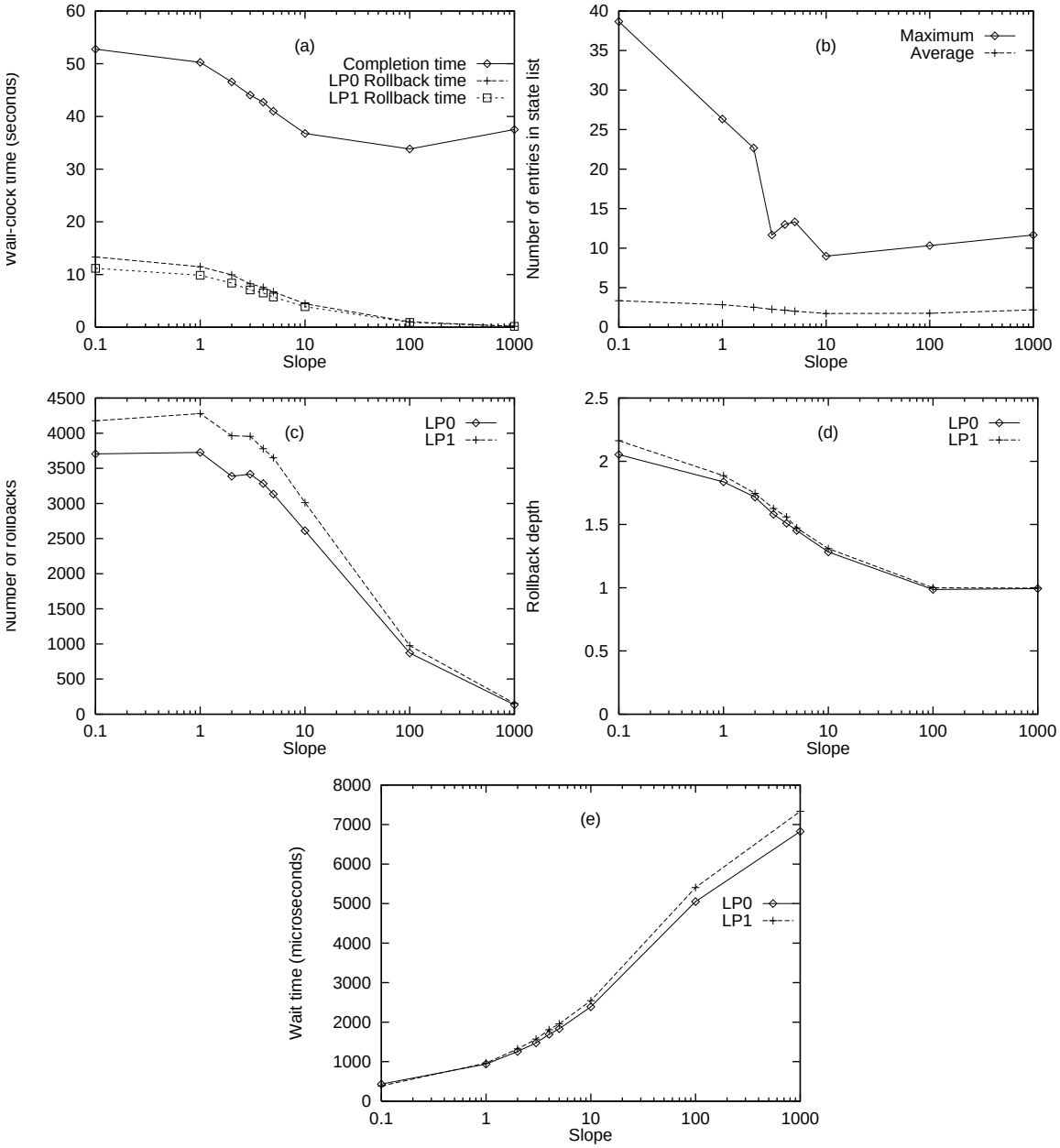


Figure 31 - Performance for ASYM

This *chasing phenomenon* will persist only if the logical distance between an erroneous message, *M* and the corresponding antimessage, -*M* (i.e. the number of LP's separating *M* and -*M)* is sufficiently large. The short length of the cycle in CHASE (3 LP's) limits this distance to at most two and thus suppresses the chasing phenomenon. This is evident in the fact that the average rollback depth with Time Warp (slope = 0) is relatively small (1.8) and the number of rollbacks is high (> 8000). The chasing phenomenon is expected to create

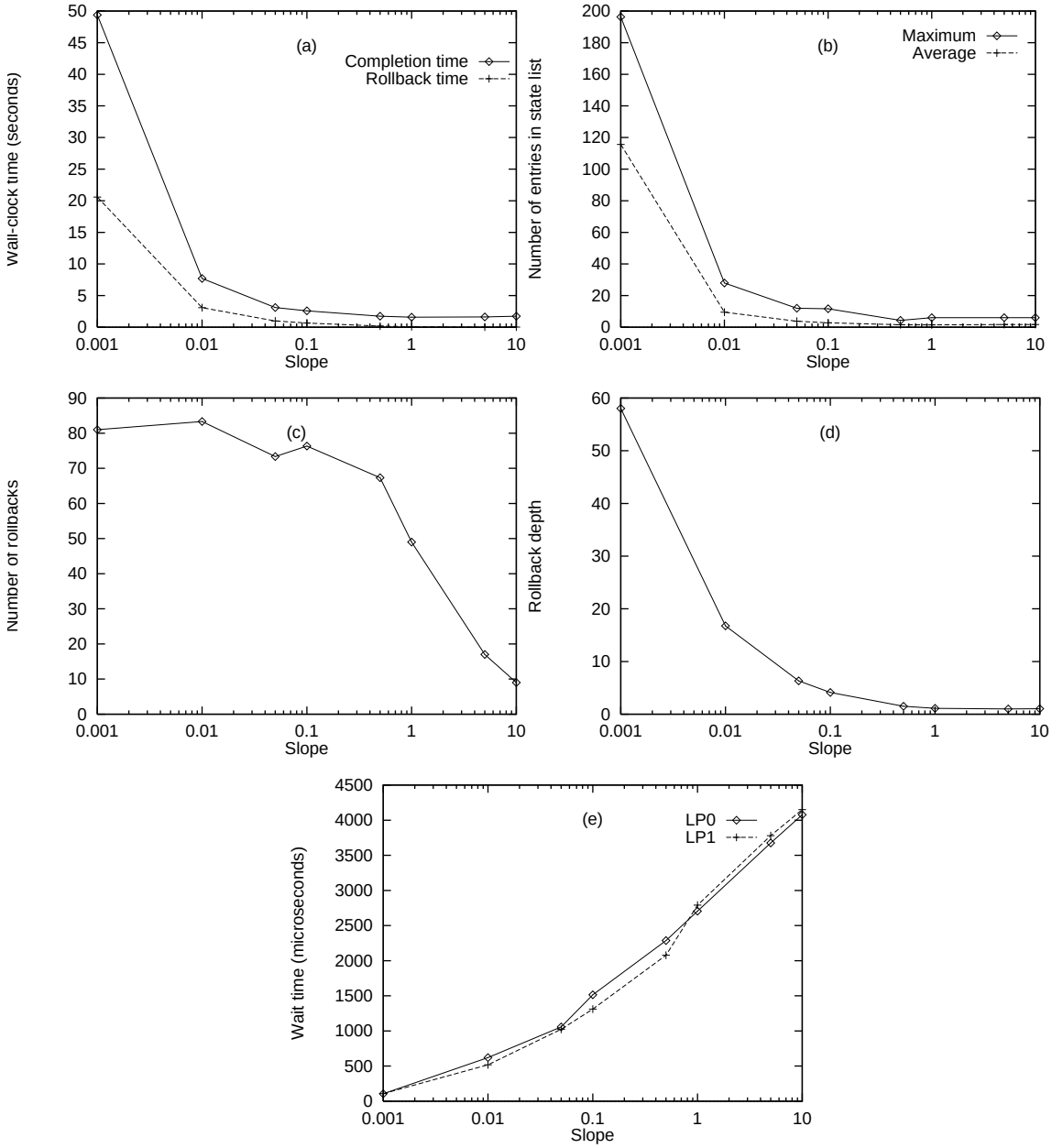


Figure 32 - Performance for ECHO

deeper and fewer rollbacks since LP's process several erroneous messages before the first antimessage arrives. The simulated performance of ETA with CHASE for larger systems (8 and 16 LP's - described in Section 6.3.6.4) supports the above reasoning.

Another effect of the short cycle length is that there is very little concurrency in the workload. The large numbers of rollbacks at the LP's in the cycle reduces the rate at which they progress through logical time. On the other hand, ETA does not restrain LP_0 since LP_0

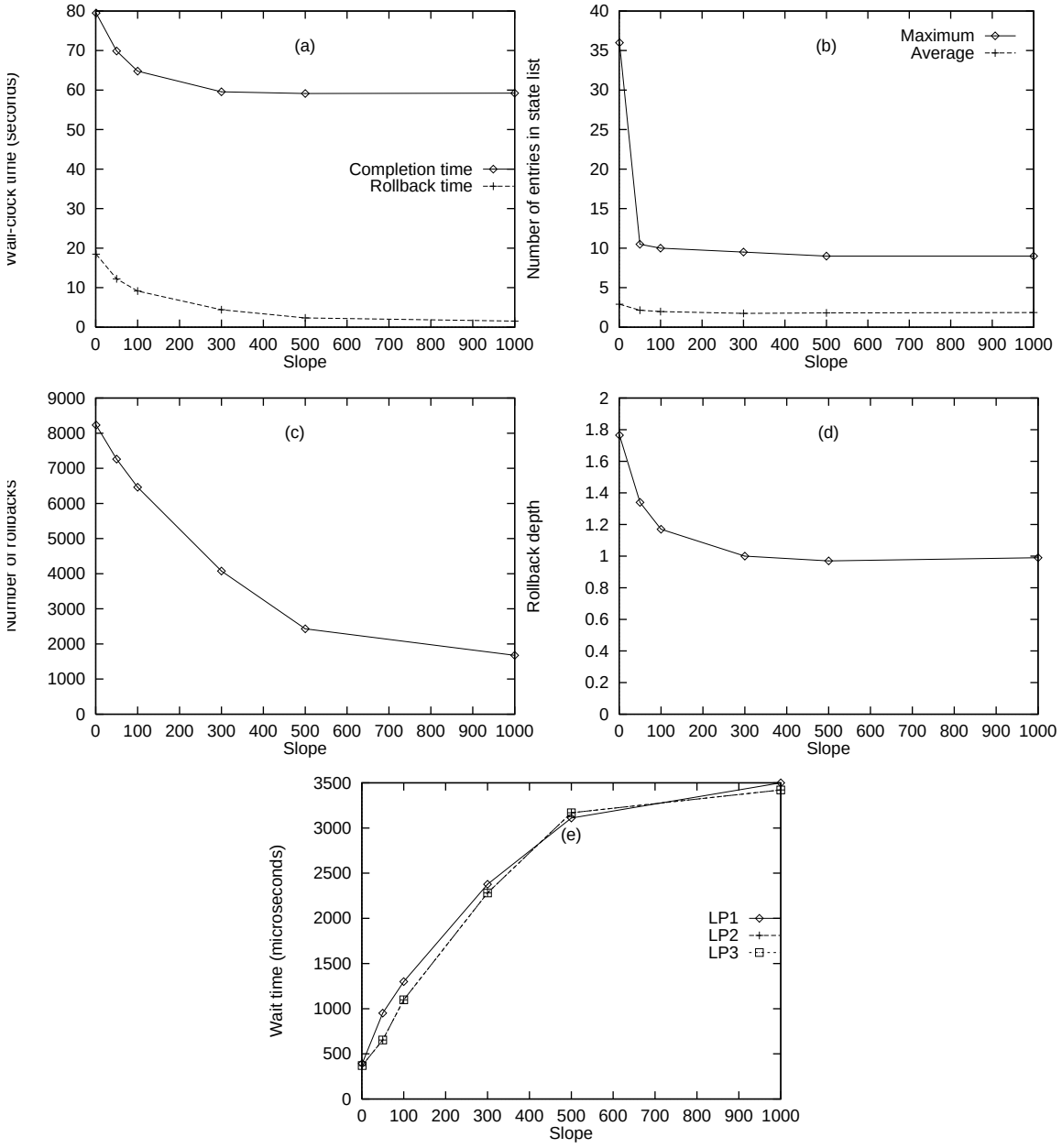


Figure 33 - Performance for CHASE

has no predecessors. As a result, LP_0 effectively becomes a *fast* source of messages for LP_1 resulting in an unstable situation where LP_1's memory consumption increases slowly but unboundedly. As noted earlier, this problem can be solved by introducing memory-based flow control into the computation of the error potential. Since such a scheme is not included in ETA currently, ETA is unable to eliminate this instability. Investigation of memory based flow control in NPSI protocols is the subject of future work (Section 9.2). Since LP_0 has no

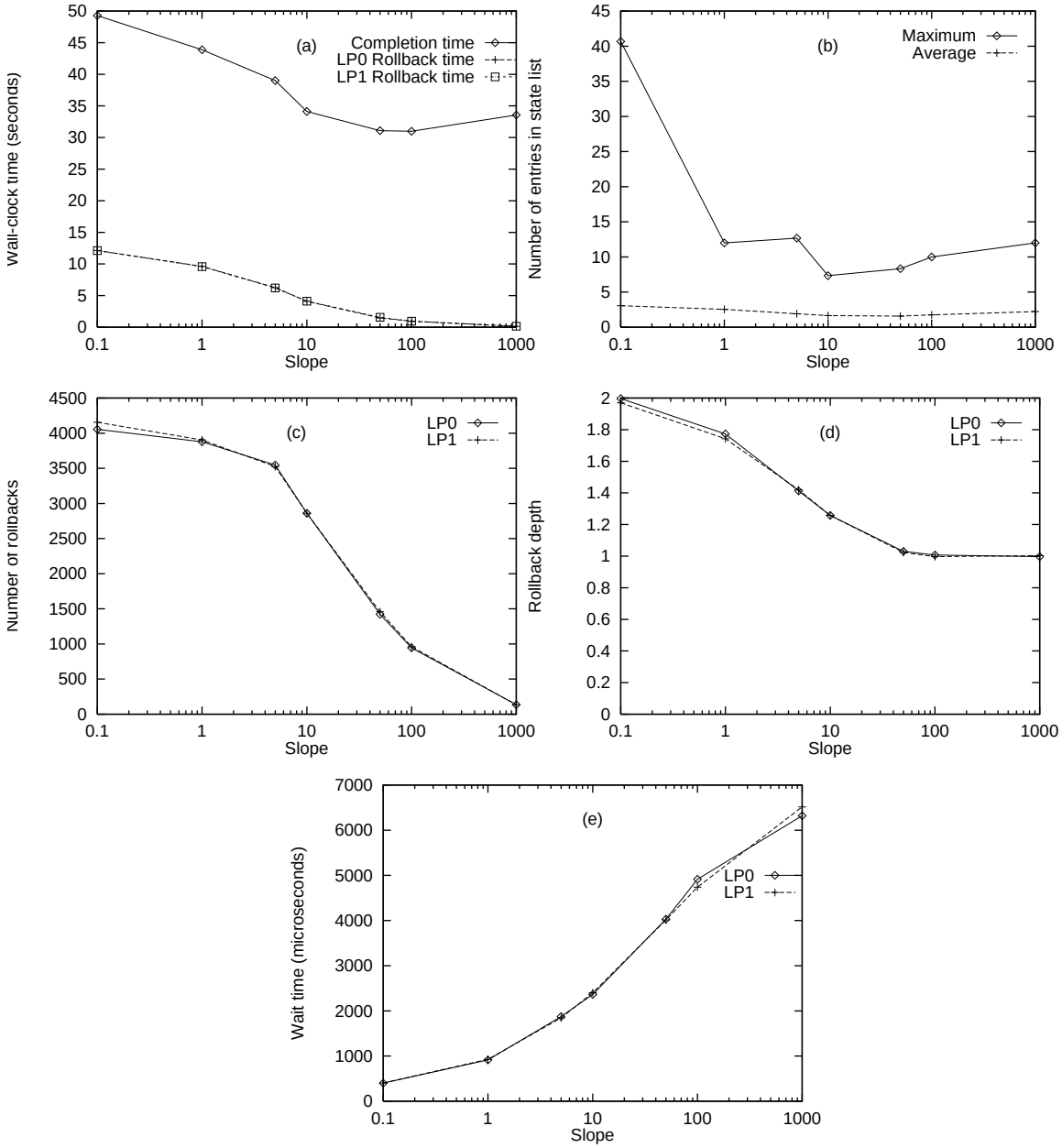


Figure 34 - Performance for PHASES-MSG

---

appreciable effect on the behavior of the other LP's besides gradually consuming LP_1's memory, it can be ignored for the purpose of analyzing the effect of ETA on the rest of the LP's. The resulting workload is very similar to MSG-SYM and correspondingly, the performance is also similar to that with MSG-SYM (Figure 29).

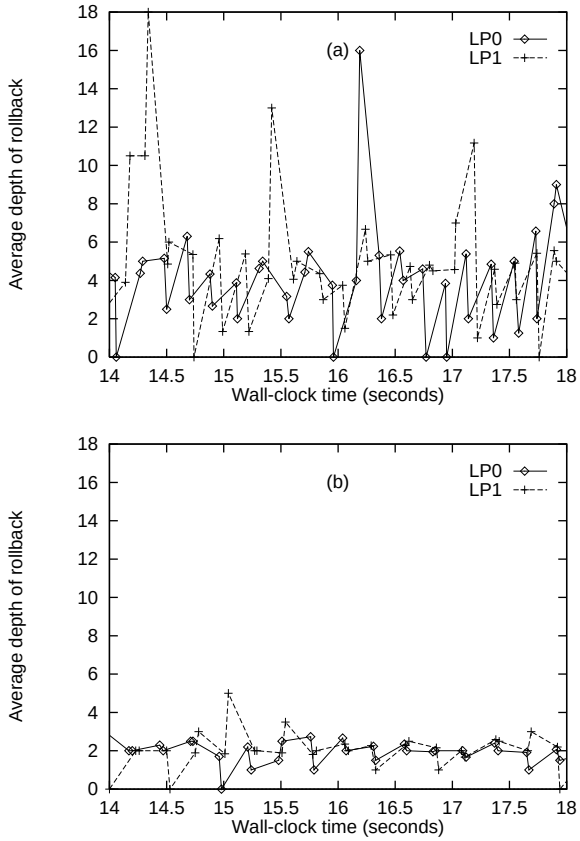


Figure 35 - Adaptive response of ETA to phase behavior

6.3.5.9 PHASES-MSG

Figure 34 shows the performance of ETA with PHASES-MSG. Due to the large event grains in the slow phases of the computation, there is some concurrency in this workload, as seen by the slightly parabolic shape of the completion time curve. As with previous workloads, the rollback time is nearly zero at the point where completion time is minimized, and the reduction in completion time (~15 seconds) exceeds the reduction in rollback time (~12 seconds). Note, in parts (c) through (e) the values of the different metrics are averaged over the entire simulation run. Since each LP executes both phases alternatingly, the average behaviors of LP_0 and LP_1 are nearly identical and do not evidence the adaptive nature of ETA.

Figure 35 shows the average rollback depth as the simulation proceeds, for both Time Warp and ETA. From this figure, the adaptive nature of ETA is clear in that it smooths out the phase behavior by throttling the appropriate LP's in each phase.

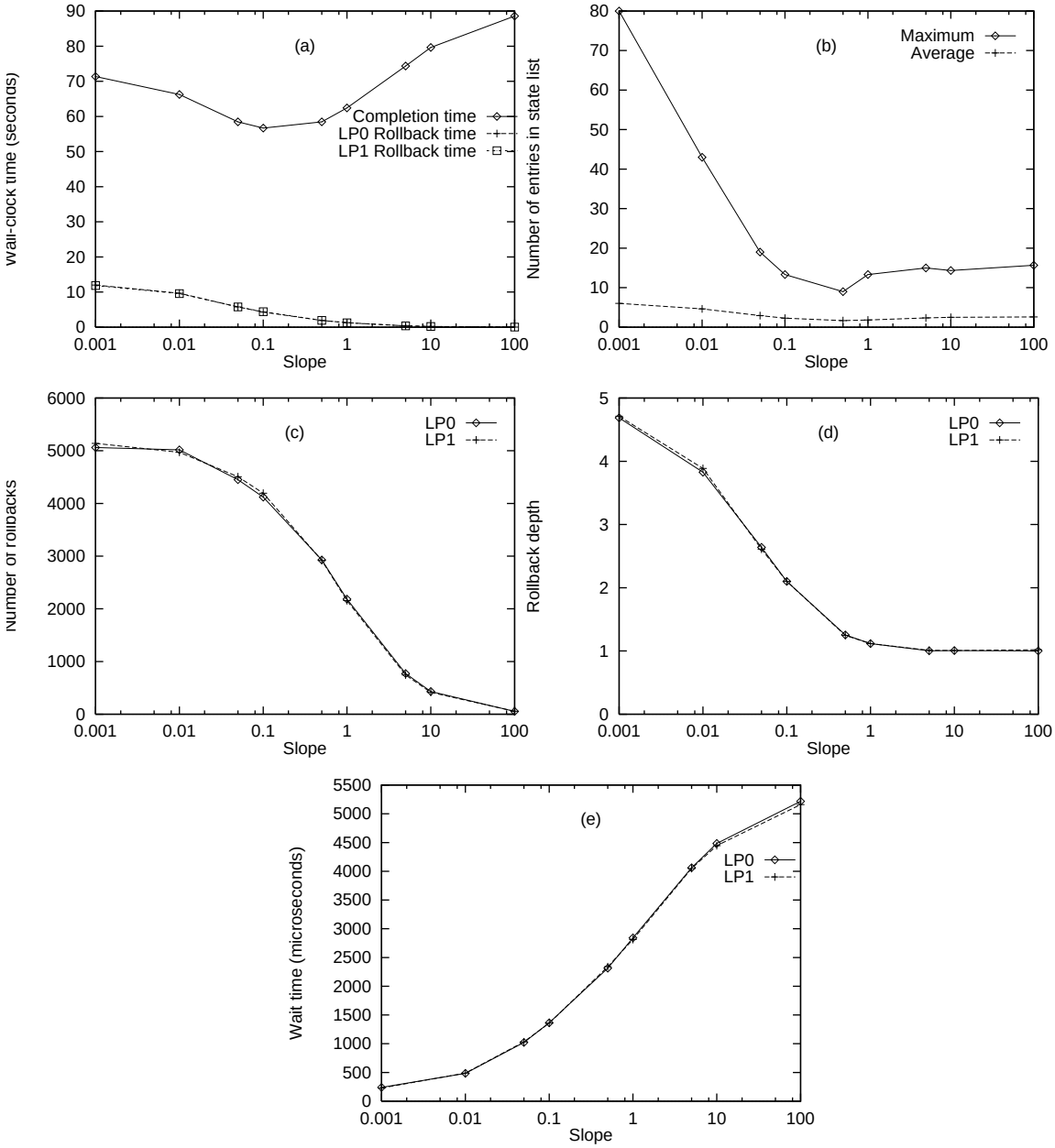


Figure 36 - Performance for PHASES-SELF

6.3.5.10 PHASES-SELF

Figure 36 shows the performance of ETA for PHASES-SELF. Since the probability of sending a message after an event execution is only 0.25, there is considerable concurrency in this workload — hence the parabolic shape for completion time. Note, the maximum state list size is high without adaptiveness — this is because of uncontrolled optimistic exe-

cution during the fast phase. Adaptive throttling of optimism by ETA reduces this memory consumption significantly.

6.3.6 Scalability

We describe a simulation study to demonstrate the scalability of ETA (and the NPSI approach). A simulation was necessary since the testing environment described previously includes a reduction network designed for four processors only.

The goal of the simulation study was to obtain some indication regarding the scalability of ETA (in general, the NPSI approach). Consequently, the study was focused on determining whether the trends observed in the earlier study continued to exist in larger systems, rather than on making specific predictions about the expected benefits of ETA. The specific trends of interest are the following:

- The completion time curve has a parabolic shape as a function of the slope. Recall, the absence of this shape in some earlier experiments was attributed to the lack of concurrency in the corresponding workloads. Due to the inherently greater concurrency in larger systems, the completion time curve is expected to have a parabolic shape.
- At the point of minimum completion time, the reduction in completion time (relative to the Time Warp execution) equals or exceeds the reduction in rollback time (relative to the Time Warp execution).
- The rollback time is very low (relative to the completion time) at the point of minimum completion time.

6.3.6.1 Simulation Model

The simulation model, developed using the SES/Workbench simulation tool [SES], consists of detailed models of the different components of the testing environment, as shown in Figure 37. The LP's are modeled by associating execution times with various segments of the code implementing ETA. Similarly, the AP model consists of the code to implement the synchronization algorithms of *npGVT2P,* with execution times associated with various segments. The execution time delay for a segment of code is approximated as the product of the number of instructions to execute that segment (estimated from a com-

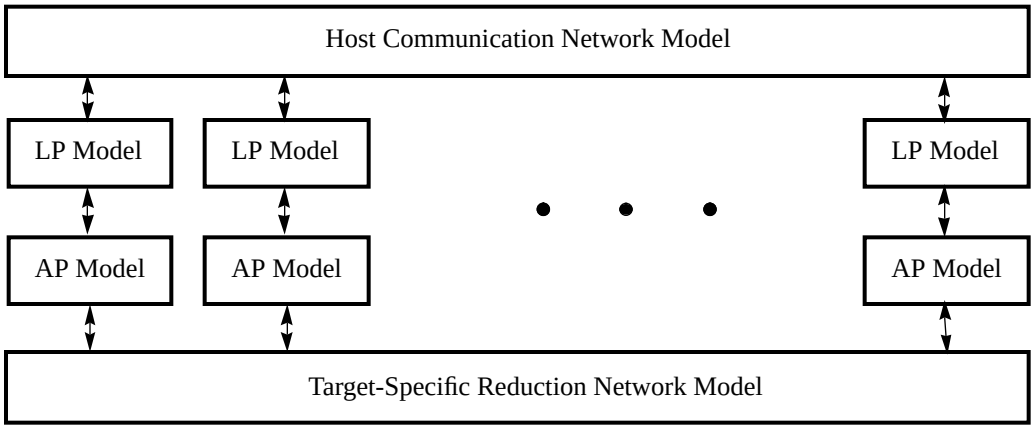


Figure 37 - Structure of simulation model

piler's assembly output) and the average instruction cycle time of the workstation. The host communication network is modeled as a simple delay with mutually exclusive access (thus, the transmission delay is proportional to the number of LP's attempting to transmit concurrently). Since the target-specific reduction network operates at very high speeds relative to the other components in the system, the model for this network was designed to execute *on demand* — i.e., it is activated only when an AP model interacts with it.

6.3.6.2 Validation

The simulation model was validated using two standard techniques [LaKe91]: face validity and comparison with an existing system. Several factors contributed to a high level of confidence in the face validity of the model: (i) the LP and AP models were constructed using code from the implementation study described earlier, (ii) a previously developed simulation model of a global reduction network [Srin92] was the basis for the target-specific reduction network model, (iii) parameters for this model were estimated by profiling the existing network using a logic analyzer, and (iv) detailed simulation traces were analyzed to confirm that the model behaves as expected. The model was also validated by comparing its output with that of the implementation on the testing environment described earlier. In each case, the output consisted of a number of statistics about the LP's, such as number of events simulated, number of messages sent and percentage of events executed. The statistics were compared for trends as slope was varied, rather than for absolute values. The validation was performed for three workloads (MSG-BIAS, SELF-BIAS and ECHO).

6.3.6.3 Testing Strategy

As noted, several model parameters used during the validation of the simulation model were based on measurements of corresponding delays on the existing system. Since this system is a prototype, these delays are much larger than those projected for a production version of the system. Accordingly, model parameters were set to the corresponding projected values for the main simulation runs. Due to the excessive lengths of the simulation runs (up to six hours to simulate one second), the runs were terminated after simulating approximately twice the amount of time to reach steady state (thus, if steady state was reached in 4 seconds, typically, the system was simulated up to 10 seconds). For this purpose, steady state was defined as a variation of less than 5% over three consecutive samples of the rate of advance of GVT.

6.3.6.4 Results

We present scalability results for the following representative set of workloads:

SELF-BIAS

The torus topology of the SELF-BIAS workload described in Section 6.3.3.2 is modified for larger systems as shown in Figure 38. Figure 39 shows the performance of ETA with SELF-BIAS for 9, 16, 36 and 64 LP's. These graphs demonstrate that the performance of ETA scales with the system size for SELF-BIAS. The increase in maximum memory consumption for larger values of slope is due to the fact that as slope increases beyond the minimum point, the rate of advance of GVT decreases due to higher lost opportunity cost. This results in a lower rate of detection of fossils and hence, longer state lists.

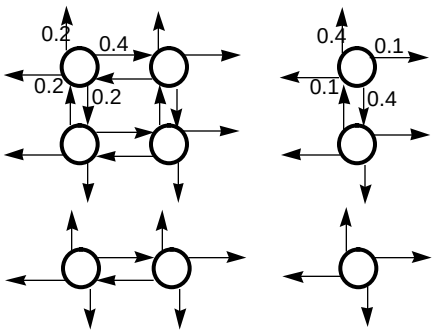


Figure 38 - Modified torus topology for large systems

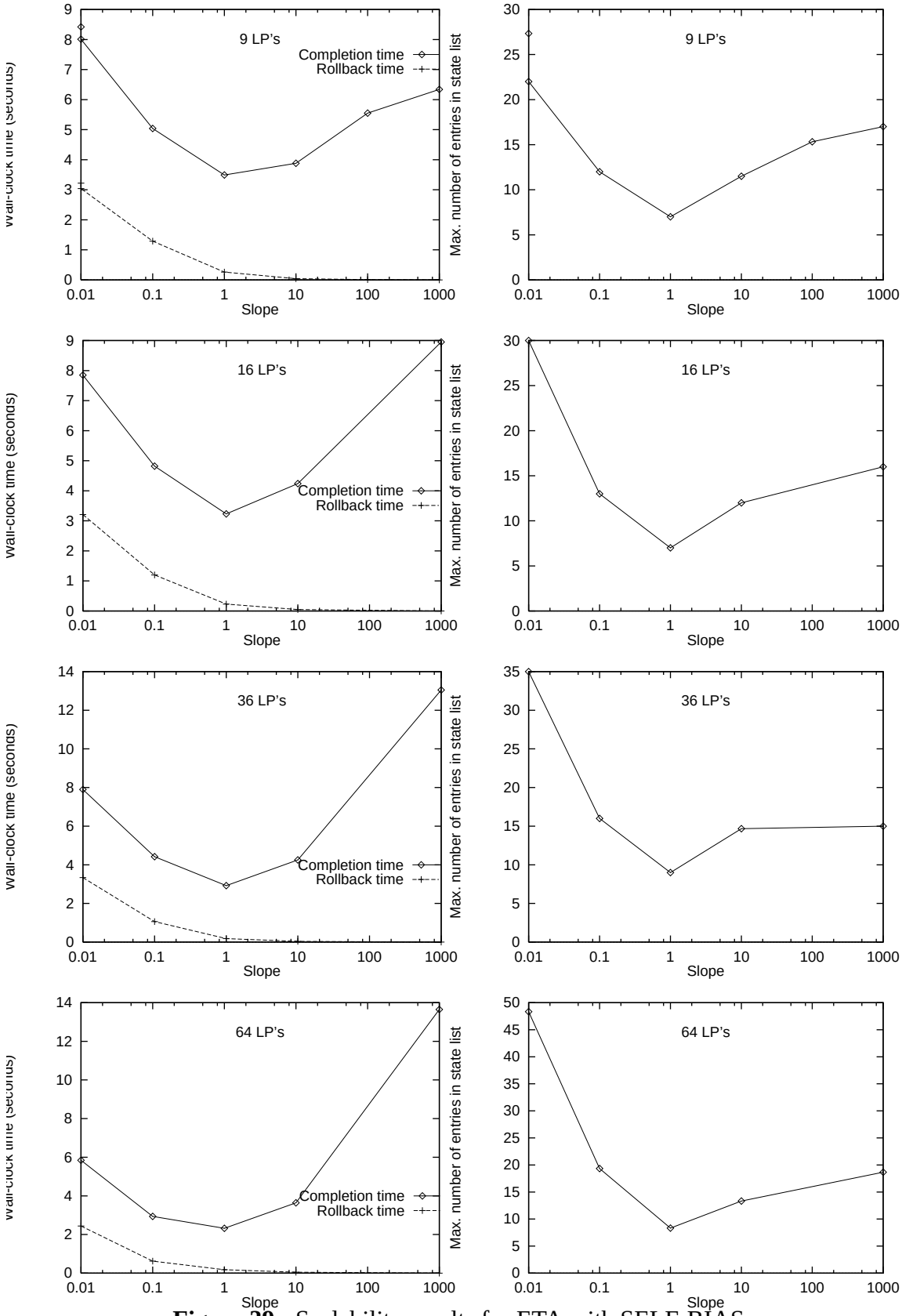


Figure 39 - Scalability results for ETA with SELF-BIAS

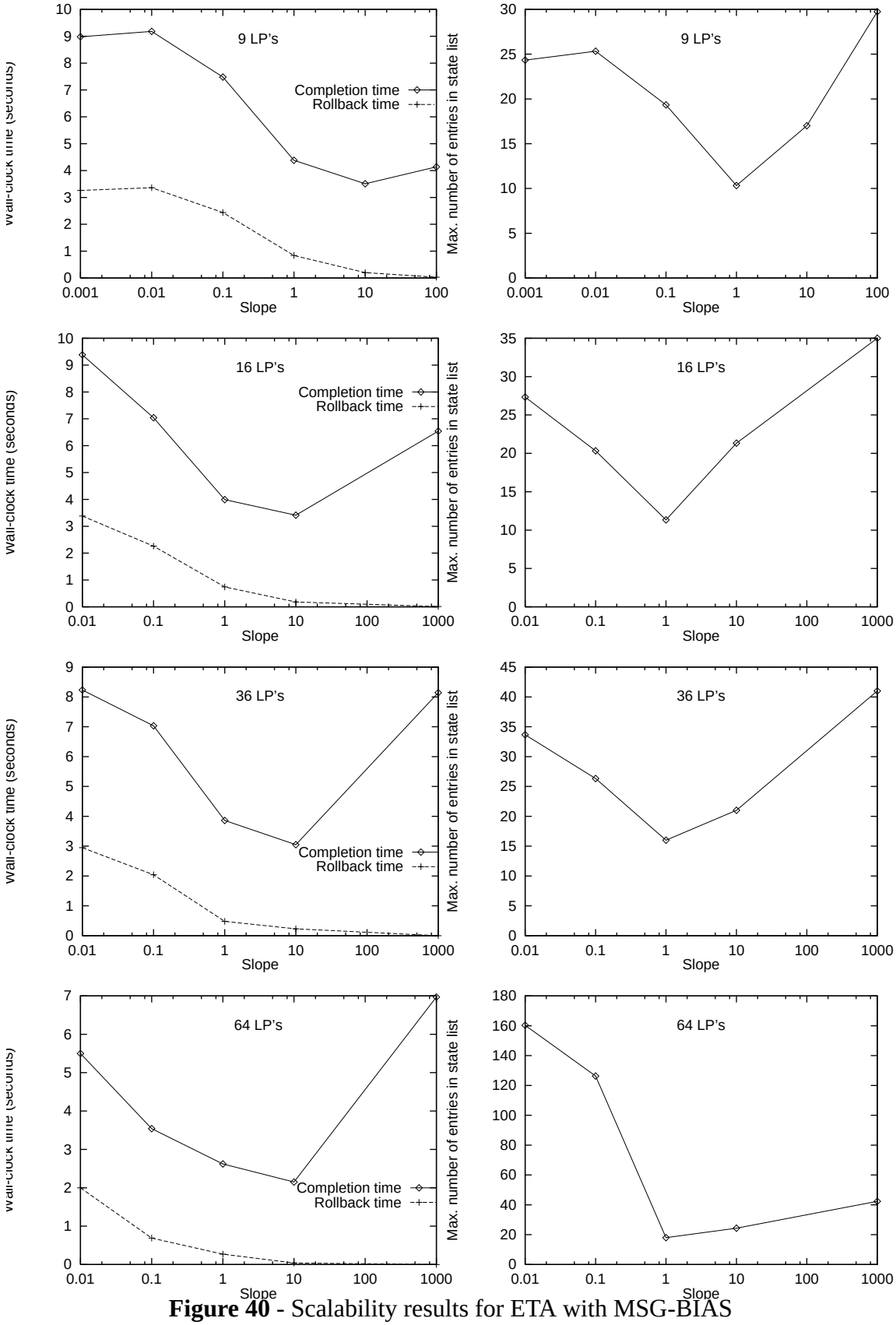


Figure 40 - Scalability results for ETA with MSG-BIAS

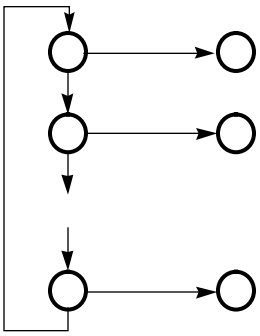


Figure 41 - Modified topology for ECHO

MSG-BIAS

This workload also uses the modified torus topology of Figure 38. Figure 40 shows the performance of ETA with MSG-BIAS for 9, 16, 35 and 64 LP's. These graphs demonstrate the scalability of ETA for the MSG-BIAS workload. The increase in maximum memory consumption for high slope values is more dramatic with MSG-BIAS than with SELF-BIAS due the fact that the former is a closed system while the latter is an open one. Since the number of events circulating in the system in MSG-BIAS is constant, they represent a fixed global memory requirement that tends to concentrate in the direction of the bias (in this case, towards the right), resulting in higher memory consumption at these LP's.

ECHO

The topology of the ECHO workload is scaled to larger systems as shown in Figure 41. A single message-initiating event is circulated among the LP's on the left. While this message rolls back one of these LP's, the other LP's process forward, incorrectly sending messages to their respective *sink* LP's. The scalability of ETA for ECHO is demonstrated in Figure 42.

CHASE

The topology for CHASE (T_4 in Figure 25) is scaled to a system of *n* LP's by increasing the length of the cycle to *n*-1 LP's. As explained in Section 6.3.5.8, increasing the length of the cycle is expected to have two effects: (i) increase the potential for sustaining the chasing phenomenon, and (ii) increase the concurrency in the workload. The increase in concurrency is evident from the parabolic shape of the completion time curves in Figure 43. The chasing phenomenon is expected to create fewer and deeper rollbacks. Since the number of rollbacks depends on the length of the simulation run, we used a normalized metric — the

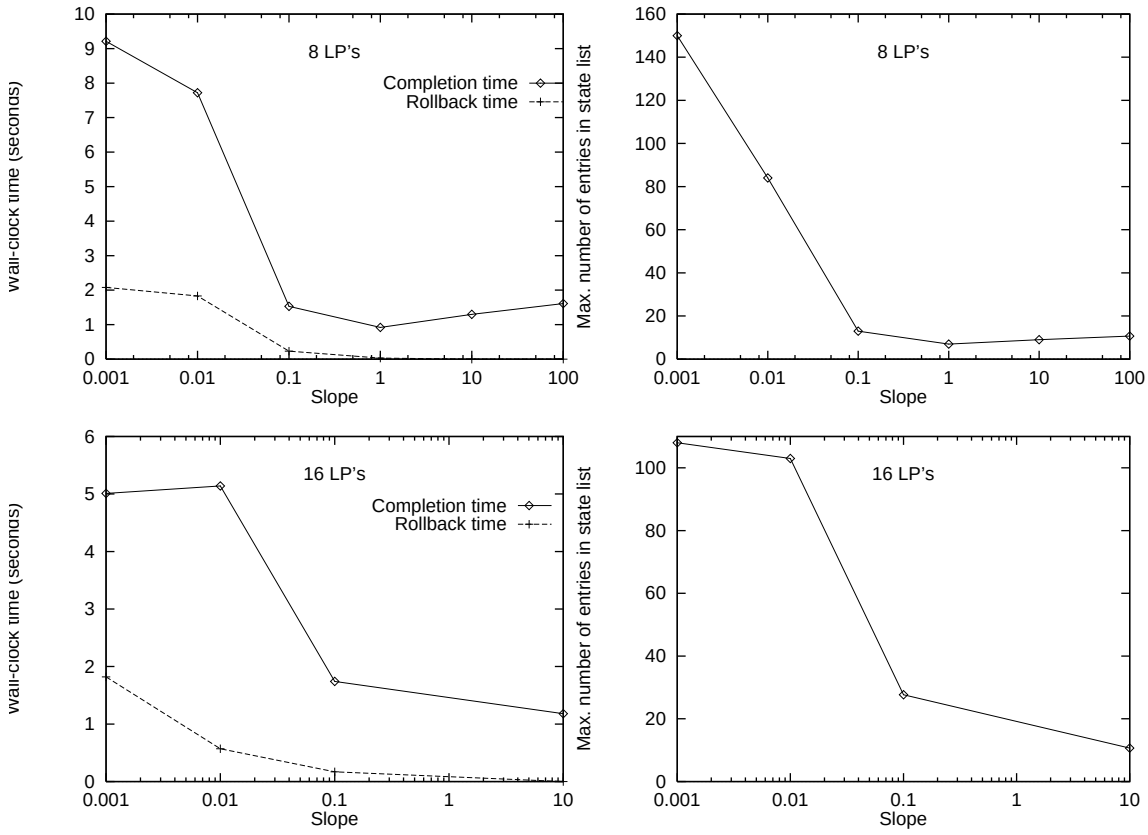


Figure 42 - Scalability results for ETA with ECHO

ratio of the number of rollbacks to the total events executed.The trends for these two metrics are shown in Table 1. It is clear from this table that rollbacks are fewer and deeper for

Table 1 - Trends demonstrating the occurrence of the chasing phenomenon

	$\frac{\text{Number of rollbacks}}{\text{Number of events}}$	Avg. depth of rollback
4 LP's	0.79	1.8
8 LP's	0.5	3.2
16 LP's	0.18	11

larger systems. This supports our reasoning that the chasing phenomenon occurs in the CHASE workload for larger systems. Figure 43 shows that ETA significantly reduces the effects of the chasing phenomenon.

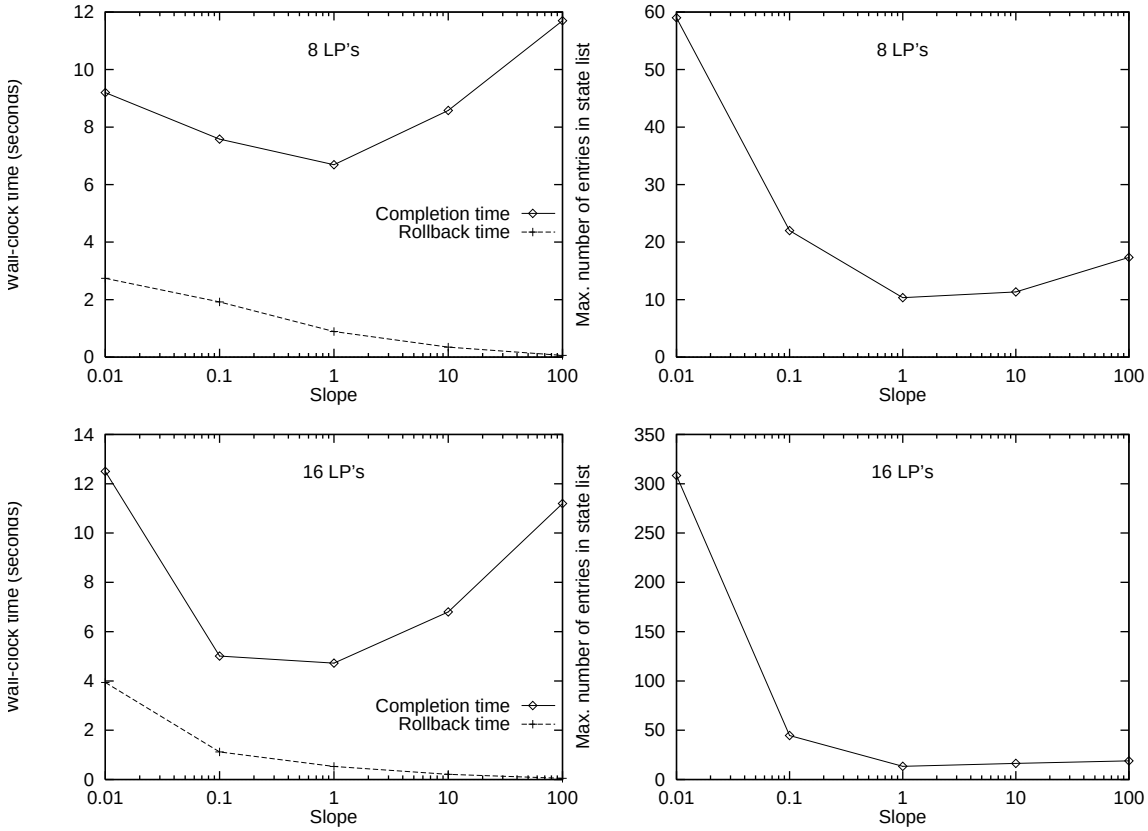
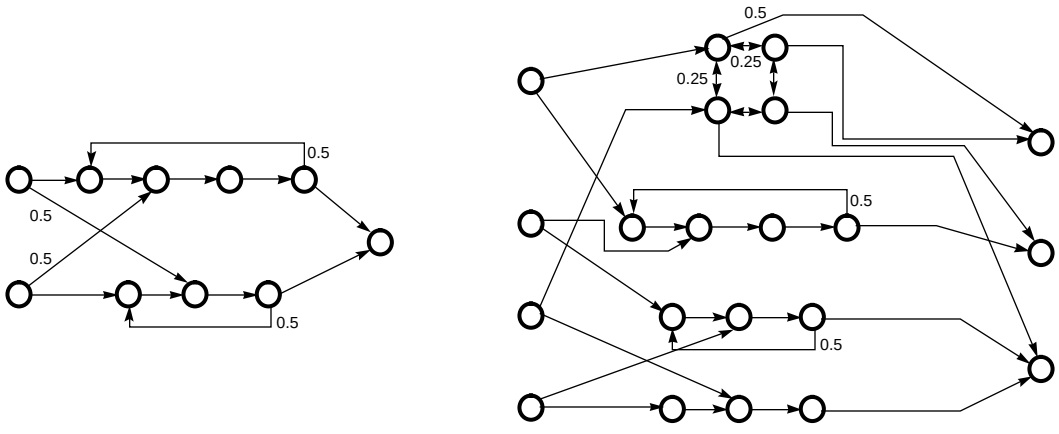


Figure 43 - Scalability results for ETA with CHASE

LOGIC-NW

The topologies for LOGIC-NW are shown in Figure 44. Note, these topologies are not



(a) Ten LP's **(b)** Twenty-one LP's

Figure 44 - Topologies for LOGIC-NW

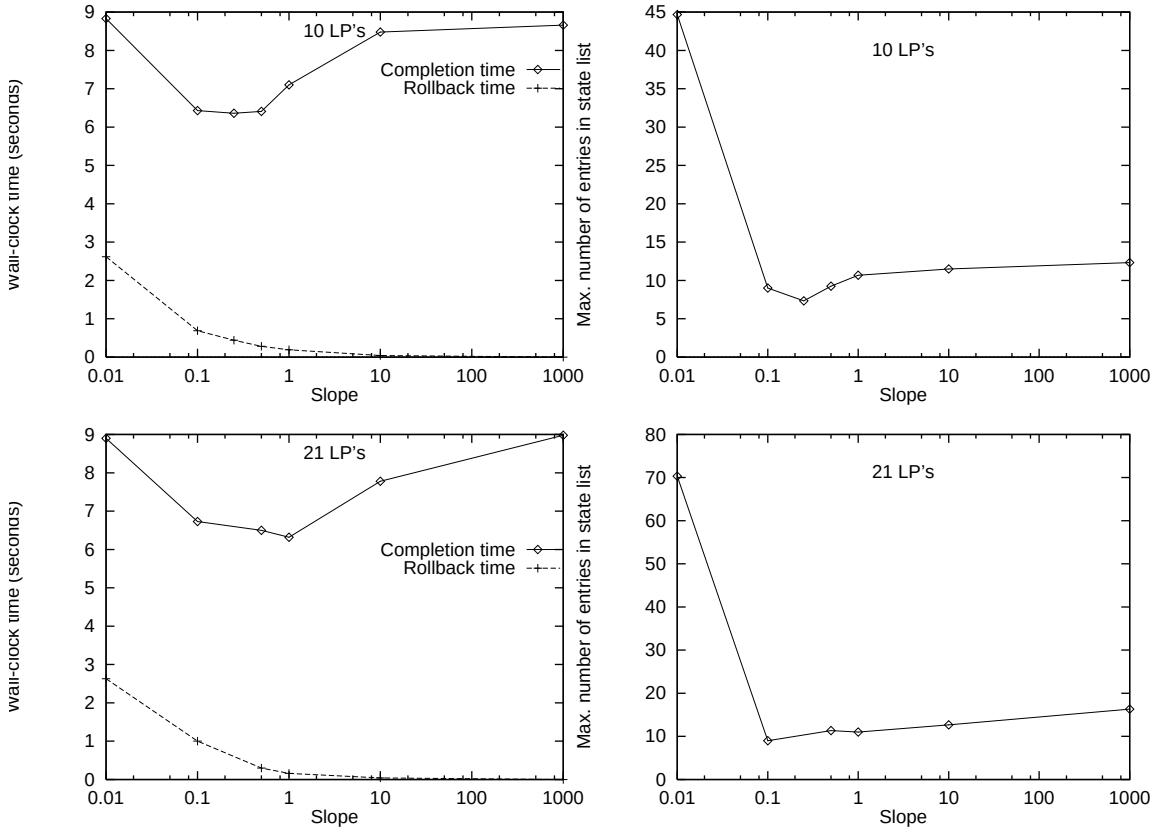


Figure 45 - Scalability results for ETA with LOGIC-NW

strongly connected (as opposed to a torus). Such topologies are typical in logic networks, which consist of a set of sources of signals and *nets* that define the flow of signals. Sequential elements in the simulated circuit introduce feedback in the topology. Alternately, this workload can represent a shop-floor with multiple assembly line systems. The performance of ETA with LOGIC-NW (Figure 45) conforms to the general trends observed with other workloads.

6.3.7 Summary of Results

In summary, we draw the following conclusions about ETA's performance:

- ETA consistently outperforms Time Warp in terms of both completion time and memory consumption.
- At the point where completion time is minimized, rollback time is very low (relative to the completion time). Further, at this point, the reduction in completion time exceeds the reduction in rollback time. This implies ETA eliminates nearly all

of the rollback costs in the application without introducing significant lost opportunity cost. Thus, the performance of ETA is close to the best possible by controlling optimism (Section 6.3.1).

- ETA produces significant reductions in maximum memory consumption. This decreases the dependence of ETA on memory management schemes such as Cancelback or Artificial Rollback [LinY92a]. By incorporating memory-based flow control into ETA's M_1 (see Chapter 4), we believe the need for such costly memory management schemes can be eliminated altogether.
- ETA is adaptive, *even for a fixed value of slope*. This is demonstrated clearly by the performance of ETA on the biased workloads as well as the PHASES workloads. Since ETA was not tailored in any way to compensate for the bias or the phase behavior (i.e. the *same* algorithm was used in *all* experiments), we have shown that ETA adapts to changes in load locality.
- The performance of ETA scales well over moderately parallel systems. We have studied the performance of ETA for systems containing 8, 9, 10, 16, 21, 36 and 64 LP's. The lack of any declining trends in performance is a clear indication that ETA's performance will continue to scale. A comprehensive scalability study over larger systems and topologies other than those considered here is a topic for future work. This will require extensive streamlining of the simulation to reduce execution times below the MTBF of the computing platforms used to execute the simulations.

6.3.8 Sensitivity Analysis

An informal sensitivity analysis of ETA was also conducted as part of the performance study, primarily to confirm theoretical reasoning about the nature of ETA and to gain insights into its operation. Sensitivity of ETA's performance to the following parameters was measured: (i) the mean timestamp increment, (ii) the frequency of state saving, and (iii) concurrency in the workload, which is controlled by two factors — the mean number of local events simulated for each message sent and the mean event execution time.

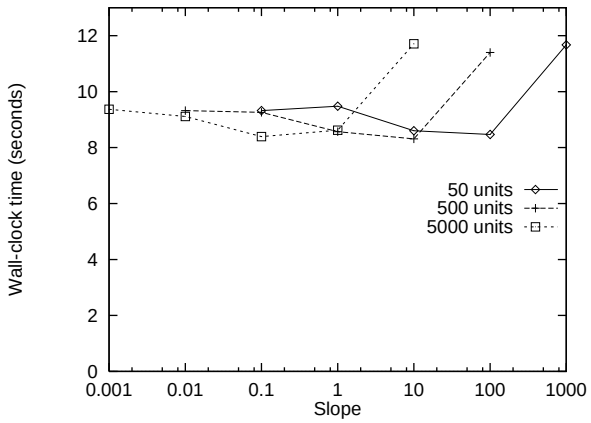


Figure 46 - Sensitivity of ETA to timestamp increments

Timestamp increment

Recall the error potential in ETA is a function of the next event time and the quantity α_i', which is also measured in units of logical time. Thus, EP is essentially a difference in logical time. The magnitude of this difference is proportional to the mean difference between timestamps of successive events — i.e., the smaller the mean timestamp increment, the smaller the range of values for EP. Since the delay introduced by ETA after each event is proportional to the EP by the slope, it is expected that smaller values of EP will require larger values of slope (and vice-versa) to provide sufficient delay. This is clearly demonstrated in Figure 46, which shows the performance of ETA for three different settings of the mean timestamp increment (50, 500 and 5000 logical time units).

State saving frequency

The frequency with which state is saved introduces a trade-off in optimistic systems. On the one hand, reducing the frequency decreases state saving costs (both time and space).

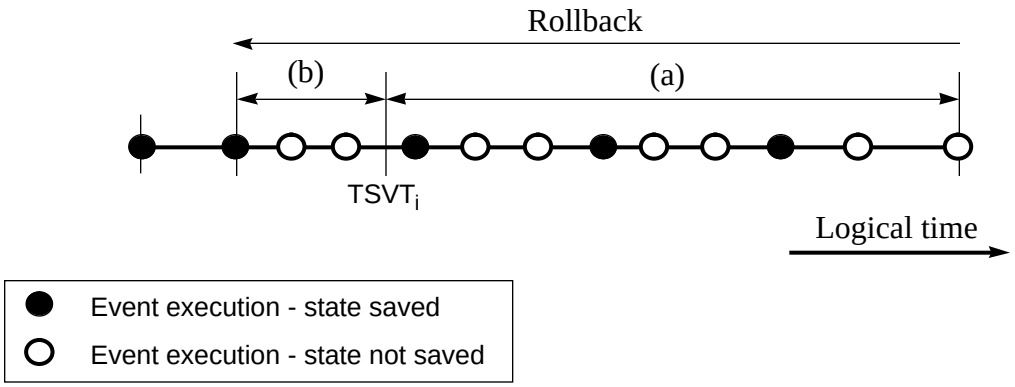


Figure 47 - Effect of infrequent state saving on rollback depth

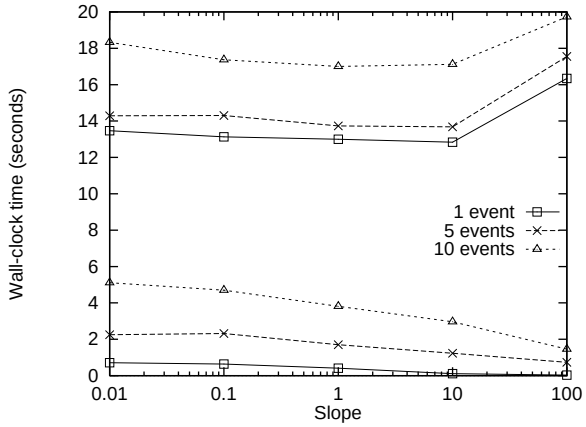


Figure 48 - Sensitivity of ETA to state saving frequency

On the other hand, if state is saved less often, LP's have to roll back farther to rebuild state, resulting in higher rollback costs. Thus, a rollback has two parts, as shown in Figure 47. Part (a) is due to optimism while part (b) is due to infrequent state saving. Since controlling optimism can reduce only part (a), it is expected that the residual rollback costs at the point of minimum completion time for ETA will be higher with less frequent state saving. Figure 48 shows the performance of ETA with three state saving frequencies: after each event, once every five events and once every ten events. The observed performance supports the reasoning above.

Concurrency in the workload

Concurrency can be increased in the workload by increasing the computation to communication ratio. In our workload generator, this is controlled by two factors: the mean number of local events per message set and the mean event execution time. As the number

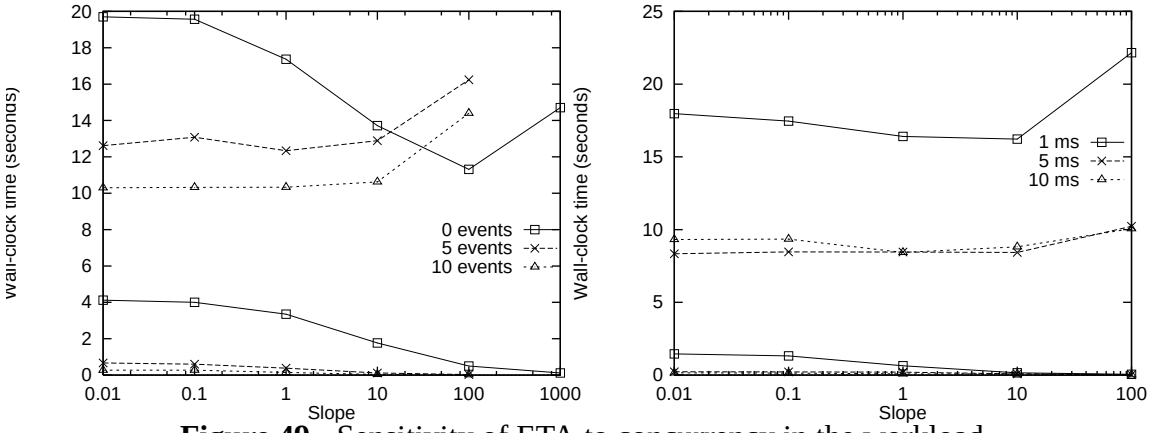


Figure 49 - Sensitivity of ETA to concurrency in the workload

of local events is increased, rollback costs decrease since fewer antimessages need to be sent out for each rollback. With large grained events, the cost of sending antimessages becomes insignificant compared to the granularity of the events. In both cases, therefore, rollback costs decrease with increasing concurrency, a situation where Time Warp is expected to perform well. Figure 49 shows the performance of ETA as concurrency is increased in the workload. With no internal events, the rollback costs are relatively high and, as expected, ETA produces significant reductions in rollback and completion times. For larger numbers of internal events, rollback costs for Time Warp (slope=0.01) are insignificant and correspondingly, ETA does not provide significant benefit. The performance for varying event execution times is similar.

6.4 ZETA

We describe ZETA, another NPSI adaptive protocol. ZETA is the forerunner of ETA and differs from it only in the mapping M_1. For ZETA, M_1 is the following function:

$$M_1 : EP_i = MAX \{\sigma_i - TSVT_i, 0\}$$

ZETA is described here for two reasons: first, the criterion for limiting optimism in ZETA (logical time distance from TSVT or GVT) is an intuitive one — other protocols have used similar criteria [Stei93, McAf90]. It is therefore important to note that in general, ETA outperforms ZETA. Second, an analysis of the drawbacks of ZETA motivated the design of ETA.

6.4.1 Performance Analysis

We describe the performance of ZETA on three representative workloads from the suite described earlier (Section 6.3.3.2). LGRAIN represents an average workload for which Time Warp performs well. MSG-SYM represents a more challenging workload for Time Warp, with high rollback costs. Finally, ECHO is a stress case due to its inherent instability. Performance with other workloads follows the same trends as with these three.

6.4.1.1 LGRAIN

Figure 50 shows the performance of ZETA with LGRAIN. The reduction in completion and rollback times is similar to that with ETA (Figure 26). Note, however, that the number

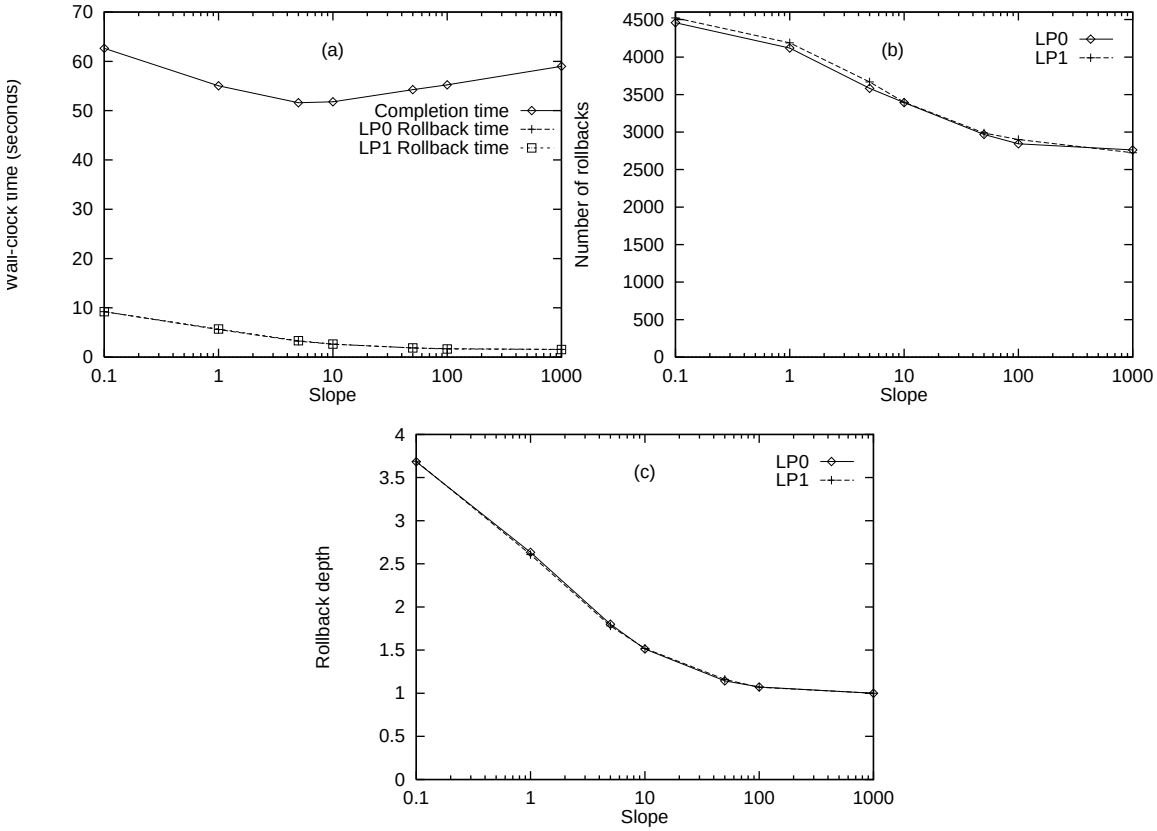


Figure 50 - Performance of ZETA for LGRAIN

---

of rollbacks decreases initially with ZETA but saturates at a fairly high value (~2750). As with ETA, the average depth of rollback decreases rapidly to the minimum value (1).

6.4.1.2 MSG-SYM

Figure 51 shows the performance of ZETA with MSG-SYM. Note, the rollback time does not decrease below 5 seconds, which is about 15% of the minimum completion time (compared with under 5% for ETA). Recall there is very little concurrency in this workload. Therefore, the best completion time results from eliminating rollbacks for the most part. However, we observe that ZETA is unable to reduce the number of rollbacks below 2500. Hence the residual rollback cost.

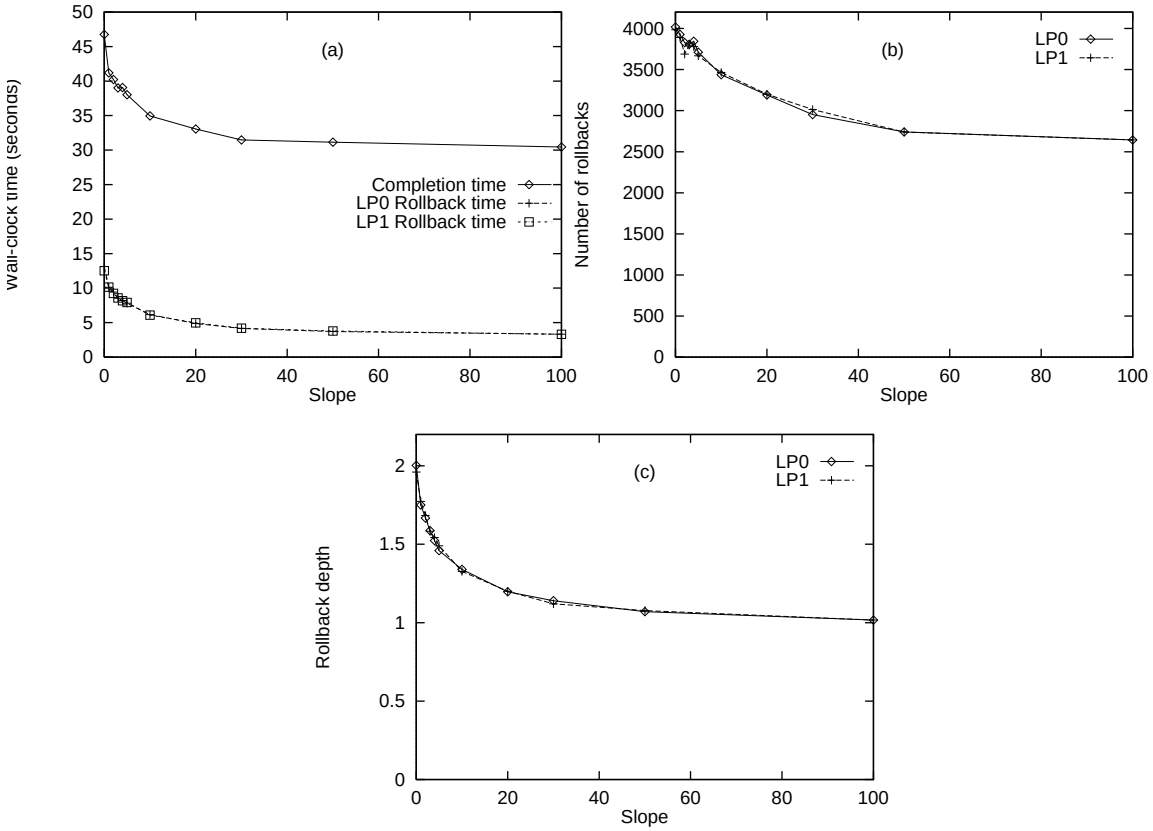


Figure 51 - Performance of ZETA for MSG-SYM

6.4.1.3 ECHO

Figure 52 shows the performance of ZETA with ECHO. Clearly, ZETA eliminates the echoing (and the instability) in the workload as effectively as ETA (Figure 32). Note, however, that ZETA does *not* reduce the number of rollbacks significantly, while ETA does.

6.4.2 Discussion

The performance study above shows that ZETA can significantly enhance the performance of Time Warp. However, in comparison with ETA, the following two conclusions can be drawn:

- In general, the percentage of rollback time eliminated by ZETA is less than or equal to the percentage eliminated by ETA.
- ZETA is as effective as ETA in reducing the average rollback depth; it is much less effective than ETA in reducing the number of rollbacks.

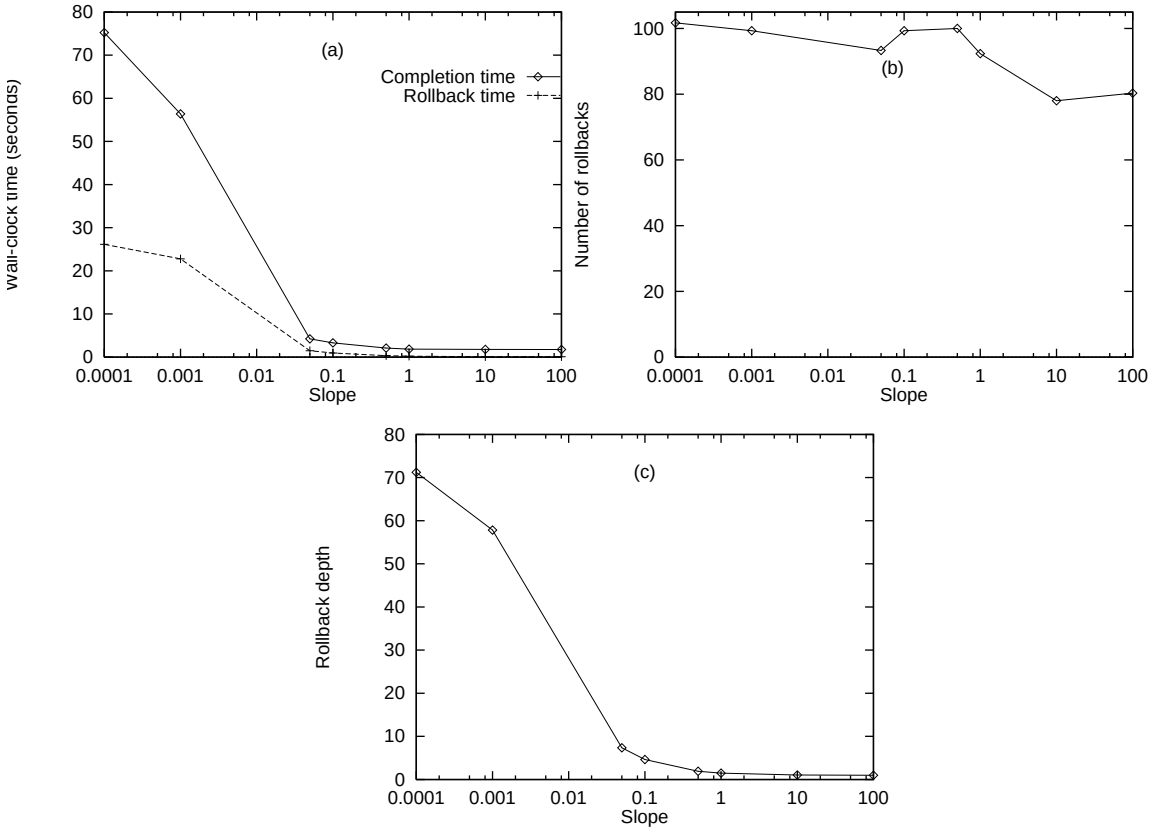


Figure 52 - Performance of ZETA for ECHO

The first of these conclusions can in fact be attributed to the second. ZETA's inefficacy in reducing the number of rollbacks is explained as follows: Consider large values of slope (this is the region in which the number of rollbacks is reduced). Since ZETA's M_2 is the same as that for ETA, a large value of slope (*s*) implies large values of δ even for small values of EP_i. In such a case, an LP will generally block until its EP reduces to zero. Recall in ETA this happens when the LP's next event is the next *safe* event (i.e. guaranteed not to be rolled back). Unfortunately, this is not the case with ZETA. EP equals zero for an LP_i using ZETA when $\sigma_i = TSVT_i$. This happens when all of the predecessors of LP_i have processed all events with timestamp < σ_i and any messages with timestamps < σ_i have been received. At this point, LP_i will terminate its waiting and execute its next event. However, there is no guarantee that its next event is safe. For example, consider an immediate predecessor of LP_i — LP_j — that has just completed executing an event so that $\sigma_j = 100$. Assume $\sigma_i = 80$, $TSVT_i = 100$, $\eta_j = 110$ and $\eta_i = 120$. In this case, ZETA will compute $EP_i = 0$ and allow LP_i to proceed even though LP_j's next event could result in a message that rolls LP_i back.

Note that in the same situation, α_i' cannot exceed 110 (η_j) and therefore $EP_i \geq 10$ using ETA. Thus, by not taking the next event times of LP's into account, ZETA's M_1 has the drawback that it tends to terminate waiting prematurely, resulting in more rollbacks. This analysis motivated the definition of ETA.

6.5 Summary

We have introduced an NPSI protocol called the Elastic Time Algorithm (ETA) and analyzed its performance. Our goal is that ETA perform well over a broad range of applications. To test for this, we have employed a synthetic workload generator to realize a suite of test workloads ranging from those for which Time Warp performs well to those that are stress cases for Time Warp. The results show that ETA *consistently* outperforms Time Warp, both in time and space (memory consumption). The two key observations are that (i) ETA consistently eliminates nearly all of the rollback cost, and (ii) the reduction in completion time exceeds the reduction in rollback time, which implies very low lost opportunity cost. These two observations suggest that ETA's performance is close to the best performance possible by controlling optimism. No other synchronization algorithm ever proposed has been shown to perform so well under such a wide range of conditions. We consider our results a *significant* advancement in PDES research.

We have presented another NPSI protocol — ZETA, the forerunner of ETA. While ZETA is an intuitive choice for controlling optimism (other protocols have used similar techniques), a performance study shows it to be not as effective as ETA.

Chapter 7

Adaptive Algorithms vs. Time Warp: An Analysis

In previous chapters, we have discussed adaptive synchronization algorithms designed to improve upon conservative and optimistic algorithms. Indeed, experimental studies show that controlled optimism (adaptive in some cases) can improve the performance of Time Warp. However, there have been no analytical studies of adaptive protocols. We present the *first known* analytical comparison of adaptively optimistic algorithms with Time Warp.

Our analysis is analogous to a previous comparison of Time Warp with the Chandy-Misra protocol [LiMi90]. We define a class of adaptive protocols, the asynchronous adaptive waiting protocols (AAWP's) and identify several practical protocols that belong to this class. We show that: (i) Time Warp can outperform an AAWP arbitrarily, and (ii) an AAWP we call ZETA, previously described in Section 6.4 can outperform Time Warp arbitrarily. Thus, while intuition suggests that adaptive protocols should enhance performance in general, our analysis indicates that they must be designed carefully since incorrect adaptive decisions can lead to arbitrarily worse performance than Time Warp. This work appears in [SrRe95c].

7.1 Assumptions

We assume the following for all protocols in this chapter:

- Each LP is located on its own processor.
- The protocols employ aggressive cancellation and aggressive rollback.

The following assumptions define *asynchronous adaptive waiting protocols* (AAWP's), the class of adaptive protocols to which our analysis applies. These protocols are based on Time Warp and control optimism by introducing delays between event executions:

```
while not done
        Process event
        Wait for time δ >= 0
        Rollback if necessary
        Process received messages
        Save state
        Collect fossils
endwhile
```

Figure 53 - Adaptive event execution loop

---

i) The simulation loop of an LP is as shown in Figure 53. Note, we do not make any assumptions about the particulars of the AAWP, such as criteria for deciding if an LP should wait on a given iteration or how long it should wait. We only assume that such waiting occurs between event executions, if at all. An LP aborts its waiting if it receives a message that will cause it to roll back. A separate delay scheme may be used to control risk.

ii) The waiting at each LP is asynchronous with respect to the waiting at other LP's.

iii) The actions of an LP relevant to the analysis are: event execution, rollback and adaptive waiting. We ignore overheads such as state saving, receiving messages, GVT computation and fossil collection for simplicity; the analysis can be extended to include these as well.

iv) The AAWP does not increase the capability of the LP's to guess computation. An increase in the guessing power of an LP could compensate for any detrimental effects of the adaptive control. Since an LP's guessing capability depends entirely on the application, it is reasonable to assume the AAWP cannot increase this capability.

v) Since adaptive waiting is expected to reduce rollback costs, the depth (and therefore cost) of each rollback is assumed to be bounded by a constant for the AAWP. If the rollback costs are not bounded for the AAWP, our claim that an AAWP can perform arbitrarily worse than Time Warp can be shown trivially.

vi) Since the delay time δ can be controlled directly by the AAWP, it is assumed to be bounded by a constant. Once again, if this were not true, our claim that an AAWP can perform arbitrarily worse than Time Warp can be shown trivially.

Several protocols that have been proposed and implemented belong to the class of AAWP's. In the penalty based throttling scheme of [ReJe89], an LP that has been rolling back excessively is made to block for some period of time. This decision is made independently of other LP's. Similarly, in Adaptive Time Warp [BaHo90], an LP may decide to block after executing an event based on local history and statistical estimation. In [Madi93], LP's estimate each others' logical clock values and block if their clock value differs largely from that of another LP. In [HaTr94], a real-time blocking window is computed each time an LP executes an event and the LP blocks for an amount of time equal to this window (which may be zero). Similarly, in [FeTr94], an LP blocks probabilistically for some amount of time after each event execution. NPSI adaptive protocols, also satisfy the AAWP assumptions. Windowing algorithms in which the windows are computed individually for different LP's (such as Unified Distributed Simulation system [McAf90] and Breathing Time Warp [Stei93]) are AAWP's as well since the LP's wait when they reach the ceilings of their independent windows. Note, global windowing algorithms do not fit the AAWP model since the global window forces all LP's to synchronize before any of them can proceed.

7.2 Time Warp Outperforms AAWP's

We show by example that an AAWP can take arbitrarily longer than Time Warp to complete a simulation. The intuition behind the example is this: on the one hand, it is possible for Time Warp simulations to execute very efficiently, with few rollbacks; on the other hand, it is also possible for a Time Warp simulation to generate many false events and consequent rollbacks which can degrade its performance severely [LuWS91]. Errors in adaptive decisions regarding when and how long to wait can cause a Time Warp execution to move from the former category to the latter. In the example here, we show a situation where the AAWP induces a false rollback chain that delays the committing of an event (relative to the Time Warp execution) by at least an amount of time proportional to the length of the

rollback chain. By arguing that this rollback chain can be arbitrarily long, we show that the committing of an event can be delayed arbitrarily.

Consider the Time Warp execution shown in Figure 54. The x-axis denotes advance of wall-clock time while the y-axis denotes the different LP's. The numbers below the events are their respective timestamps. A dashed arrow indicates the causal dependence of an event on a message (i.e. the event at the head of the arrow was scheduled by the arrival of the message at the tail of the arrow). The two important events to note in this execution are: (i) the event with timestamp 140 at LP_1, which is causally dependent on a message from LP_0 that arrives just in time to be executed by LP_1, and (ii) the event with timestamp 165 at LP_5, which is the one whose committing execution will be delayed due to an erroneous waiting decision.

Figure 55 shows an execution of the same simulation as in Figure 54 using an AAWP. For simplicity, we assume the initial conditions for the two executions are the same except for one difference: both LP_0 and LP_1 wait for some time at the beginning of the portion of the execution depicted. Due to what will turn out to be an error in the decision process, LP_0 delays longer than LP_1. As a result, LP_1 is ready to execute an event before the message from LP_0 arrives and causes the event with timestamp 140 to be scheduled. LP_1 executes its next scheduled event (with timestamp 150) and sends a message to LP_2. Since we assume aggressive rollback, this event is a *false* one as it is executed out of order. When the message from LP_0 arrives later, this false event is rolled back and an antimessage is sent to LP_2. However, the message sent to LP_2 by the false event has already initiated a chain of false events. The antimessage starts a chain of rollbacks and antimessages that follows close behind the chain of false events. Regarding these chains, we observe the following:

i) It is possible that the rollback chain catches up with the false event chain immediately and thus terminates both chains. However, the case that is relevant to this discussion is the one shown in Figure 55 where the rollback chain does not catch up with the event chain until the latter reaches LP_5 and initiates an unnecessary rollback. This scenario is feasible [LuWS91].

ii) The chains by themselves may be harmless - the problem arises from the fact that the final false message that arrives at LP_5 (marked M_F in Figure

55) has timestamp 160, which is smaller than 165, the timestamp of the second event at LP_5 (marked E_D in Figure 55). Thus, M_F rolls back the first execution of E_D, *even though that execution was correct and could have been committed.* E_D is re-executed after the false rollback completes. Therefore, the committing execution E_D is delayed by at least an amount of time proportional to the length of the false rollback chain.

iii) Each of the two chains has a cycle in it, involving LP_2, LP_3 and LP_4. While we have shown only one iteration of this cycle, it is possible to have an

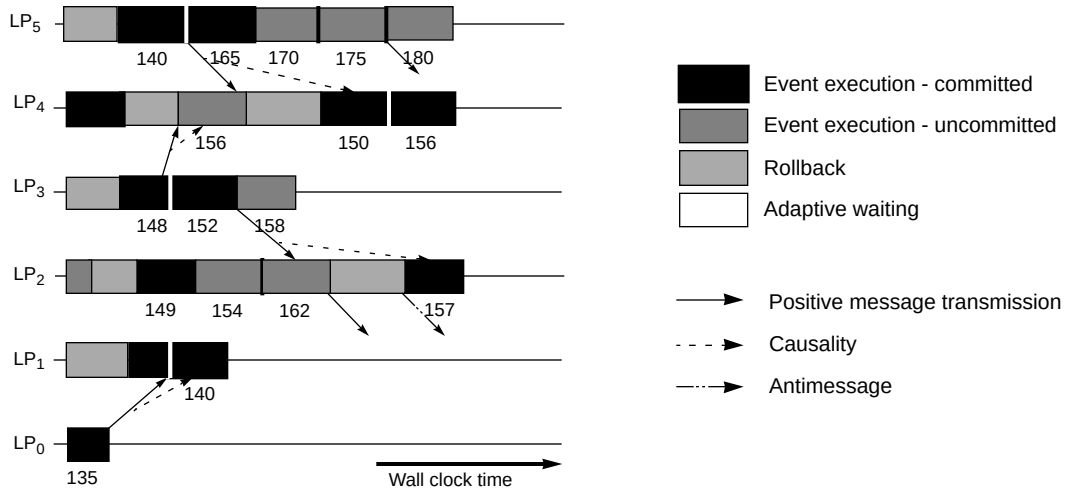


Figure 54 - Time Warp execution

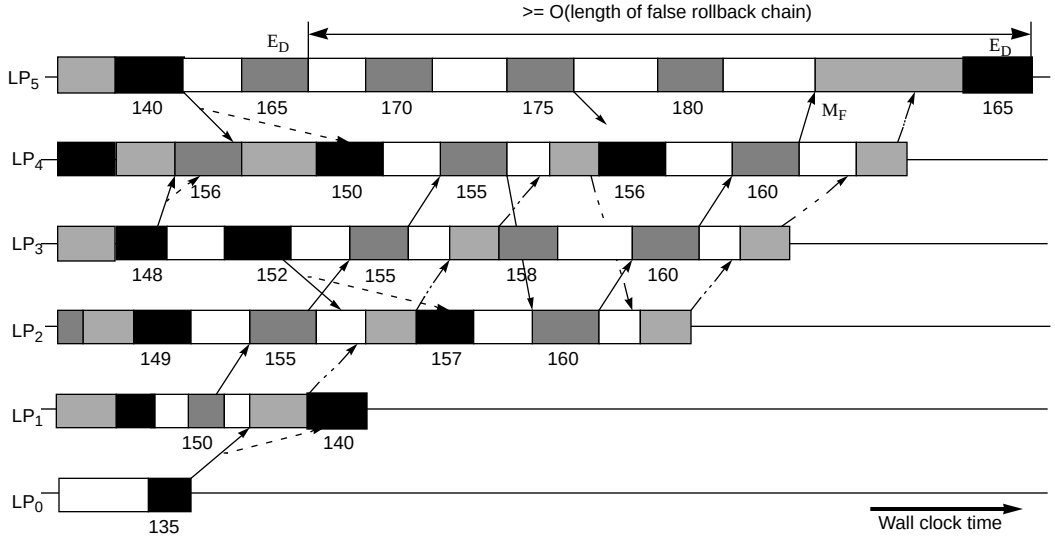


Figure 55 - False rollback chain with cycle due to incorrect waiting

arbitrary number of iterations, i.e. the chains could revisit these LP's an arbitrary number of times before reaching LP_5.

iv) For the false event chain to delay the committing execution of E_D, the timestamp of M_F must be smaller than that of E_D. Since we assume a correct implementation of the underlying Time Warp protocol and the application, it follows that logical time must advance eventually as we traverse the false event chain. Therefore, an arbitrarily long chain would require that the timestamp of E_D also be arbitrarily large (we disregard pathological situations such as when the false chain makes diminishing advances in logical time such that the timestamp of M_F is bounded; such situations reflect a flaw in the implementation - the physical system being simulated cannot make diminishing progress in real time!). The larger the timestamp of E_D, the less likely that its first execution is on the critical path of the simulation, since other LP's are farther behind in logical time. Thus, while it is possible for the false chains to be arbitrarily long, the probability that these chains are damaging to the simulation decreases with the length of the chain. However, the probability is not zero - we can imagine a simulation where E_D marks the transition to a new phase of simulated time, i.e., the entire simulation makes a "jump" in logical time to the next phase of activity. If so, the delaying of E_D could cause the entire phase transition to be delayed.

v) It is possible to have very long false chains which do not span much logical time, as shown in Figure 55. Here, logical time does not increase in an iteration of the cycle in the false chain; it increases only across iterations. Thus, the timestamp of M_F is small but the chains are long.

vi) Finally, even if the lengths of the chains are bounded (by some means), it is possible to have an arbitrary number of instances of such chains in a simulation (i.e., the number of instances is bounded only by the logical time span of a simulation run).

In summary, we have shown that the committing execution of an event can be delayed (relative to the Time Warp execution) by an arbitrary amount of time due to false events and rollbacks created by errors in the waiting decision.

7.2.1 Lazy Cancellation

Since the events in the rollback chain are false events (i.e. they should not have been generated), they will have to be rolled back even under lazy cancellation. Thus, the example holds under lazy cancellation as well. In fact, the problem is aggravated to a certain extent with lazy cancellation because the initial antimessage generated by the rollback of the event with timestamp 150 at LP_1 will be delayed until LP_1 crosses time 150 after executing the event with timestamp 140. Thus, the physical time lag between the two chains may be longer, allowing the false event chain to propagate farther.

7.2.2 Lazy Rollback

The commitment of the first execution of E_D in the Time Warp execution implies that the event with timestamp 150 at LP_1 *must* be a false one even with lazy rollback. Thus, the two chains will be generated in this case also. However, E_D may not be rolled back when M_F arrives. This can happen due to one of the following: (i) M_F has no impact on the state in which E_D was executed, or (ii) the rollback chain arrives at LP_5 while LP_5 executes the event scheduled by M_F so that when the execution is complete (or is preempted, assuming event preemption), it is cancelled immediately and the operating state for E_D is not affected. However, even if the final execution of E_D is not delayed, the advance of GVT[1] beyond 165 will be delayed by at least the amount of time it takes for the chains to disappear. Since GVT is the commitment horizon for events, the committing of E_D can be delayed arbitrarily even with lazy rollback.

7.2.3 Pragmatic Issues

We have identified a general class of adaptive protocols, the AAWP's, and shown that it is possible for Time Warp to outperform them by an arbitrary amount. Since several practical protocols belong to the AAWP class, this counter-intuitive result suggests that AAWP's must be designed with care. Correspondingly, while we have developed several

1. GVT is the minimum of all logical clocks and timestamps of any messages in transit.

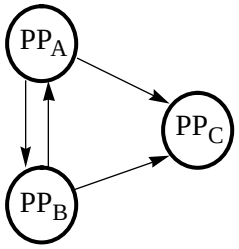


Figure 56 - EchoSystem: Physical system for echoing

very effective NPSI adaptive protocols, we have also discovered some that perform no better than Time Warp and some that perform worse.

It is possible to design AAWP's so as to avoid the scenario described earlier. However, it is not clear whether doing so will be beneficial in the general case. Moreover, since there may be other scenarios similar to the one we have described, establishing a property of AAWP's that avoids this scenario would not by itself guarantee that Time Warp will not outperform AAWP's by more than a constant factor.

7.3 AAWP's Outperform Time Warp

We show by example that Time Warp can take arbitrarily longer than an AAWP to complete a simulation. Our approach is to describe the execution of a simulation using both Time Warp and a specific AAWP. We show that Time Warp takes an amount of time that is quadratic in the amount of logical time simulated while the AAWP takes linear time. Since the difference in completion times is not bounded by a constant for a given simulation, the AAWP outperforms Time Warp arbitrarily.

7.3.1 Physical System

The system we consider for simulation was described previously in [LuWS89] as an example of "echoing" in Time Warp. We refer to it as *EchoSystem*. It consists of three physical processes (PP's), A, B and C with the communication topology shown in Figure 56. Upon receiving a message from PP_A, PP_B processes it and sends a message to PP_A and vice-versa. If no message is received from the other, both of them prepare a message to send to PP_C. If a message arrives when one is being created or sent to PP_C, that sending is aborted and the new message is processed. Sending and receiving of messages takes no time. Processing a message from PP_A (PP_B) takes u real time units on PP_B (PP_A). Preparation of a

message to PP_C takes $2u$ real time units. Suppose at time 0 PP_A receives the first message from PP_B. Then it may be verified that the only message traffic that occurs in this system is between PP_A and PP_B at intervals of u real time units. The idle periods between intervals are insufficient to build a message to PP_C. Note, real time in the physical system corresponds to logical time in the simulator.

For the simulator, we assume the following: processing a message between PP_A and PP_B takes one unit of wall-clock time (including sending the follow-on message), preparing a message to PP_C takes one time unit and sending of an antimessage also takes one time unit. LP's advance their simulation clocks to the timestamp of the next event *after* executing that event. We assume this to simplify the proofs - the theorems can be proven even if it is assumed that logical clocks are advanced before commencing event execution. LP_C is assumed to perform its work fast enough so that its actions are irrelevant to the discussions and proofs. σ_X denotes the logical clock value of LP_X.

7.3.2 Time Warp Execution

The Time Warp execution of this simulation is shown in Figure 57. The x-axis represents wall-clock time. The numbers (in multiples of u) at the junctions of the LP time lines and the unit time intervals indicate the logical time to which the LP's have simulated (i.e. the logical clock value of the LP). A solid arrow indicates a message transmission while a dashed arrow indicates an antimessage transmission. The bold lines at various points on the time lines of LP_A and LP_B indicate rollback. From the picture, the echoing is evident immediately in the fact that the two LP's roll back alternately, with increasing amplitudes.

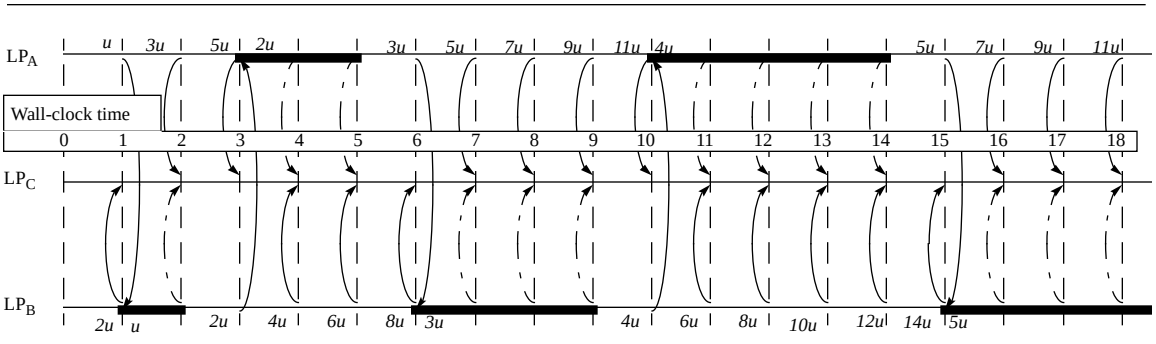


Figure 57 - Echoing in Time Warp

THEOREM 1: A Time Warp execution of EchoSystem takes $n(n+1)/2$ wall-clock time units to simulate nu units of logical time.

PROOF: The theorem is proved by induction on the amount of wall-clock time required for GVT to advance from logical time *(n-1)u* to *nu*.

Induction hypothesis: It takes *n* units of wall-clock time for GVT to advance from logical time *(n-1)u* to *nu*.

Base case: *n* = 1: From Figure 57 we see that in the first wall-clock time interval (i.e. wall-clock time [0:1)) LP_A advances σ_A to *u* and LP_B advances σ_B to *2u*. Thus, GVT has advanced from 0 to *u* in one wall-clock time unit and the hypothesis holds.

Induction step: Assume the hypothesis holds for the logical time window [*(n-1)u*:*nu*). Recall that the message traffic in the physical system consists only of the single message being exchanged by LP_A and LP_B. Thus, in any correct simulation of this system, GVT advances by *u* logical time units each time an LP receives this message, processes it and sends it back. During this process, the actions of other LP's cannot affect GVT. Without loss of generality, assume LP_A takes *n* wall-clock time units to advance GVT from *(n-1)u* to *nu* (by induction hypothesis). In these *n* wall-clock time units, LP_B will send *n* messages to LP_C. Thus, at the end of [*(n-1)u*:*nu*), LP_A has just sent a message to LP_B with timestamp *nu* and LP_B has sent *n* false messages to LP_C. Therefore, LP_B takes *n* wall-clock time units to send the *n* antimessages to LP_C and one wall-clock time unit to process the message, advance σ_B to *(n+1)u* and send the message back to LP_A with timestamp *(n+1)u*. At this point, GVT will have advanced to logical time *(n+1)u*, requiring *n+1* wall-clock time units to do so. Thus the total wall-clock time required to simulate up to logical time *nu* is $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$. ■

We have thus shown that Time Warp takes $O(n^2)$ wall-clock time to simulate the specified physical system, where *n* is a measure of the logical time span of the simulation run.

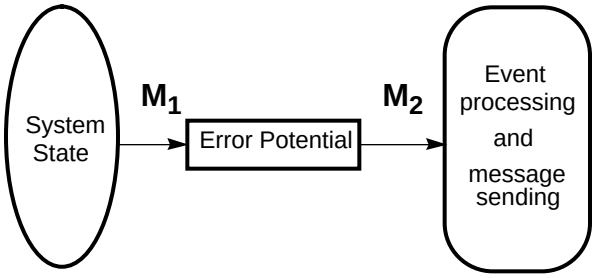


Figure 58 - Framework for NPSI adaptive protocols

7.3.3 AAWP Execution

Recall the general design framework of Section 4.3, repeated in Figure 58 for convenience. Based on this framework, different NPSI adaptive protocols may be constructed by designing the mappings M_1 and M_2. Note, these mappings only specify if an LP should wait and how long it should wait; the general structure of NPSI protocols conforms to the AAWP model in Section 7.1, i.e., NPSI adaptive protocols are AAWP's. Consider the following instance of ZETA, the NPSI adaptive protocol described in Section 6.4:

$$M_1 : EP_i = \sigma_i - GVT_i$$

$$M_2 : \delta_i = \begin{cases} \dfrac{EP_i}{MaxEP_i} & EP_i > 0 \\ 0 & EP_i = 0 \end{cases}$$

where $MaxEP_i$ is the maximum value of EP_i observed thus far. After executing an event, LP_i re-computes δ_i and waits for δ_i units of wall-clock time before proceeding to the next event. If a message is received during a wait period that will cause a rollback, the LP aborts the waiting and proceeds to roll back.

Figure 59 shows the execution of the simulation using the protocol described above. The shaded lines represent waiting due to the adaptive protocol. It is evident from the diagram that the echoing observed under Time Warp (Figure 57) has been avoided since each rollback is of only unit length.

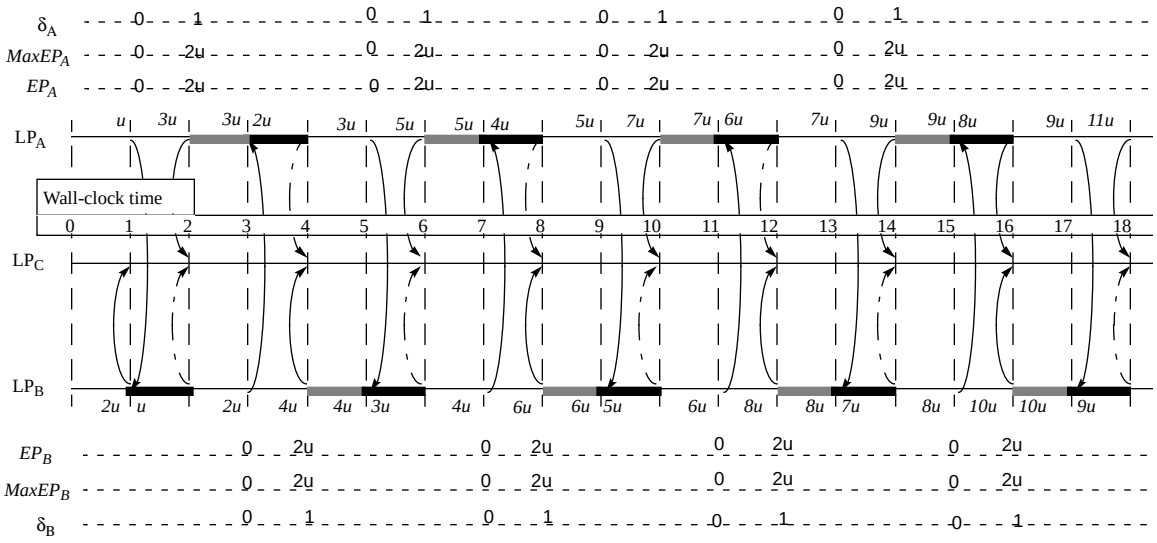


Figure 59 - Avoiding echoing with adaptive aggressiveness

The discussion in Section 4.2 justifies the following assumption:

NPSI Assumption: A change in an LP's logical clock value is reflected in the values of GVT visible to the different LP's in a fraction of the time it takes for an LP to execute an event.

For example, we may assume this latency is equal to 0.1 wall-clock time units since an LP takes one wall-clock time unit to execute an event.

THEOREM 2: An execution of EchoSystem using ZETA takes 2*n*-1 wall-clock time units to simulate *nu* units of logical time.

PROOF: The proof consists of induction on the amount of wall-clock time required for GVT to advance from logical time *(n-1)u* to *nu*.

Induction hypothesis: For *n* = 2, 3, ... (i) it takes 2 units of wall-clock time for GVT to advance from logical time *(n-1)u* to *nu*, (ii) $\text{MaxEP}_A <= 2u$, and (iii) $\text{MaxEP}_B <= 2u$.

Base case: *n* = 2: Referring to Figure 59, at wall-clock time 1 LP_A is at logical time *u* and has sent a message to LP_B while LP_B is at logical time *2u* and has sent a false message to LP_C. Thus GVT = *u* at wall-clock time 1. The message from LP_A causes LP_B to roll back to logical time *u* and send an antimessage to LP_C, requiring one wall-clock time unit. In

the wall-clock time interval [2:3), LP_B processes the message, advances σ_B to *2u* and sends a message with timestamp *2u* back to LP_A. In the wall-clock time period [1:2), LP_A advances σ_A to 3*u*, builds a message and sends it to LP_C. Since LP_B rolls σ_B back to *u* just after wall-clock time 1, by the NPSI assumption, at wall-clock time 2 we have $\sigma_A = 3u$, $GVT_A = u$ and therefore $EP_A = 2u$. Since $MaxEP_A$ was 0 initially, $MaxEP_A = 2u$, giving $\delta_A = 1$. Thus LP_A waits during the wall-clock time period [2:3). Consequently, at wall-clock time 3, GVT = *2u*, $MaxEP_A = 2u$ and $MaxEP_B = 0$.

Induction step: Assume the hypothesis holds for the logical time window [*(n-1)u*:*nu*). As in the proof of Theorem 1, the argument is made that in any correct simulation of the specified physical system, (i) GVT advances by *u* logical time units each time an LP receives a message, processes it and sends it back and (ii) the actions of other LP's during this period cannot influence this GVT advance. Without loss of generality, assume LP_A takes 2 wall-clock time units (by hypothesis), say [*i-1:i*) and [*i:i+1*), to advance GVT from *(n-1)u* to *nu* and send a message with timestamp *nu* to LP_B. In [*i-1:i*), LP_B advances σ_B from *(n-1)u* to *(n+1)u* and sends a message to LP_C. Since LP_A sent a message with timestamp *nu* to LP_B at the end of [*i:i+1*), σ_A must be *(n-1)u* at the end of [*i-1:i*). Further, LP_A received a message from LP_B at the beginning of [*i-1:i*) (because LP_A advances GVT). This implies LP_A could not have built and sent a message to LP_C during [*i-1:i*). Therefore, LP_A must have rolled back during [*i-1:i*). This means σ_A must have been rolled back to *(n-1)u* at the beginning of [*i-1:i*). By the NPSI assumption, at the end of [*i-1:i*) $GVT_B = (n-1)u$. Thus, $\sigma_B = (n+1)u$, $EP_B = 2u$, $MaxEP_B = 2u$ (by hypothesis) and $\delta_B = 1$. Consequently, LP_B waits during [*i:i+1*). At the end of [*i:i+1*), GVT = *nu*, LP_A has sent a message with timestamp *nu* to LP_B and LP_B has sent one false message to LP_C. The message from LP_A at the end of [*i:i+1*) causes LP_B to roll back and send one antimessage to LP_C (requiring one wall-clock time unit). LP_B uses another wall-clock time unit to process the message, advance σ_B to *(n+1)u* and send a message with timestamp *(n+1)u* to LP_A. Thus, after two wall-clock time units, GVT is advanced from *nu* to *(n+1)u*. Since LP_B controls the advance of GVT in these two wall-clock time units, $EP_B = 0$ and consequently, $MaxEP_B = 2u$. The actions of LP_A in this period are exactly the same as

those of LP_B in the interval [*(i-1)u:(i+1)u*) described earlier. It follows that $MaxEP_A$ also equals 2*u*.

The total wall-clock time required to simulate *nu* units of logical time is therefore given by the sum of the wall-clock times required to simulate the windows [0:*u*) and [*u*:*nu*):

$$1 + \sum_{i=2}^{n} 2 = 2 \cdot n - 1 \quad \blacksquare$$

We have thus shown that ZETA takes O(*n*) wall-clock time to simulate EchoSystem, where *n* is a measure of the logical time span of the simulation run. The corresponding completion time with Time Warp is $O(n^2)$. Since the ratio of completion times is not bounded by a constant for a given simulation, the adaptive protocol outperforms Time Warp arbitrarily.

7.3.4 Pragmatic Issues

We have described ZETA, a specific AAWP and shown that it can outperform Time Warp arbitrarily. While the simulation for which this phenomenon is observed is somewhat contrived, ZETA is not. In Section 6.4.1.3 we have presented the empirical results of simulating EchoSystem (the ECHO workload) using an implementation of ZETA. In conformance with our analysis here, we observed that the speedup of ZETA over Time Warp increased with the logical time span of the simulation, i.e., the larger the maximum simulated time, the larger the speedup. Although this analysis has not been extended to ETA, the main NPSI algorithm in Chapter 6, we believe it is possible to do so. The fact that the performance of ETA (Section 6.3.5.7) with EchoSystem closely matches that of ZETA supports this belief.

7.4 Summary

The lack of consistently good performance with the two traditional approaches to synchronization in parallel discrete event simulations (conservative and optimistic) has led to the exploration of adaptive approaches. Experimental studies indicate adaptive protocols can improve PDES performance. However, there have been no analytical studies comparing the performance of adaptive protocols with that of traditional protocols. We have pre-

sented the first known analytical comparison of adaptive protocols with Time Warp. We have demonstrated that it is possible for Time Warp to arbitrarily outperform a class of adaptive protocols we call asynchronous adaptive waiting protocols (AAWP's). Protocols in this class control aggressiveness and risk by introducing independently controlled delays at the LP's. This class is general enough to include many practical protocols. Conversely, we have presented an example in which an NPSI protocol (which is also an AAWP) outperforms Time Warp arbitrarily. Thus, while adaptive limiting of optimism appears to enhance performance in practice, our study shows that care must be taken in the design of AAWP's since incorrect adaptive decisions can lead to arbitrarily worse performance than Time Warp.

Chapter 8

Tunable Parameters

By definition, an adaptive protocol that dynamically controls optimism based on the (observed or predicted) state of a system includes tunable parameters — those whose values must be determined in an application-specific manner or adjusted dynamically to ensure good performance. This chapter concerns automatic methods to tune such parameters by monitoring the values of relevant metrics. We identify two such metrics based on a characterization of tunable parameters. One of these metrics is used in an algorithm designed to automatically tune the slope parameter in ETA (Chapter 6). Performance results show this algorithm to be very efficient.

8.1 Why Tunable Parameters?

One of the conclusions drawn from the results presented in Section 6.3 is that the scaling factor *s* must be chosen properly for ETA to perform well. Too small a value will not reduce rollback costs sufficiently while too large a value will introduce lost opportunity cost. The value of *s* that minimizes the completion time depends on the application and hence cannot be determined *a priori* in general. The user may be able to provide some hints about this value; on the other hand, there is no theory to aid the user. We refer to parameters such as *s* as *tunable parameters.* Tunable parameters are those whose values must be determined in an application-specific manner or adjusted dynamically to ensure good performance.

While *s* is an artifact of the particular M_2 chosen for ETA, such tunable parameters will exist in any adaptive protocol. By definition, adaptive protocols are expected to react to changes in the system. If the protocol is a general one (i.e. one that will be used with a wide range of applications), it must interpret the state information it obtains or estimates in the context of the particular application. This process entails tunable parameters such as scaling factors, size of history and filter constants. A problem similar to adaptive synchronization of PDES's is adaptive load balancing/sharing in parallel and distributed systems. It is well known that good load balancing requires good estimation of tunable parameters

such as thresholds and maximum number of transfer-hops. Accordingly, we observe that all existing adaptive protocols admit tunable parameters of some sort (the time-window bound in [TuXu92], N_1 and N_2 in [Stei93], cluster size in [RaAT93], blocking window size in [BaHo90], the scaling factor c_i in [HaTr94] and M_f and M_p in [DaFu94]).

Clearly, there is a need to invent a scheme by which adaptive protocols may tune the values of such parameters dynamically. The Adaptive Time Warp algorithm [BaHo90] computes its blocking period based on a particular logistic response function. Thus its efficacy depends on how accurately this function reflects reality. Moreover, it is an entirely local optimization. The Local Adaptive Protocol [HaTr94] also uses a local optimization strategy in which the parameter is tuned based on observed rate of progress of virtual time per channel. The scheme of [DaFu94] tunes two parameters, M_f and M_p dynamically based on an analysis of the flow of memory buffers in a Time Warp simulation. This scheme is specific to the nature of the parameters being tuned.

8.2 Types of Parameters

We categorize tunable parameters along two dimensions — static vs. dynamic and global vs. local:

static The desired value (i.e. the value that maximizes global performance) of a static parameter depends on the application but is constant throughout a simulation. Thus, once tuned, the value of a static parameter will not require re-tuning during a simulation.

dynamic Not only is the desired value of a dynamic parameter application dependent, but the value may change during the simulation. Thus a dynamic parameter requires re-tuning as the simulation proceeds.

global the value of a global parameter is the same for all LP's.

local the value of a local parameter is determined independently for each LP.

The four types of parameters are shown in Table 2:

Q I Global Static	Q II Global Dynamic
Q III Local Static	Q IV Local Dynamic

Table 2 - Types of tunable parameters

8.3 Metrics

The general approach to tuning parameters is to monitor the progress of the simulation in order to determine how well the current values for tunable parameters are performing. Metrics must be identified that can be measured during the simulation and that reflect the progress of the computation accurately. Once again, an analogy may be drawn with load balancing where a good load index is essential [FeZh87]. We propose two metrics, based on whether the tunable parameters are global or local.

8.3.1 Global Tunable Parameters

A global tunable parameter implies that all LP's use the same value of the parameter at any point in the simulation. In other words, there is only one value to be tuned (for each parameter). Thus, we can employ a global metric to tune a global parameter. Since GVT denotes the logical time up to which the simulation has advanced, the rate of advance of GVT is a good indicator of the progress of the simulation. We propose a different global metric:

- R_C: the *total* rate of commitment of events over all LP's

We believe R_C to be a better indicator of global progress than the rate of advance of GVT because the base of reference for R_C (events/second) is the same for all applications unlike that for GVT (units of logical time/second). By taking the total commitment rate over all LP's we take into account the case where events are distributed unevenly over the logical time line at different LP's.

Given the reduction model described in Chapter 4, R_C may be computed easily as follows: each LP maintains an element in its input state vector (ISV) that tracks the number

of events committed by that LP. Each time an LP commits events, the value of this element is incremented by the number of events just committed. The NPSI calculator applies the ADD function to this element of *all* ISV's to produce the total number of committed events in the OSV's. R_C may then be computed by each LP by sampling this element in the OSV periodically and using the following formula:

$$R_C = \frac{N_i - N_{i-1}}{T_i - T_{i-1}}$$

where N_i is the i'th sample and T_i is the time at which it was taken.

8.3.2 Local Tunable Parameters

With local tunable parameters, different LP's can use different values for a parameter at any point in the simulation. In parallel simulation, as in most parallel computations, local optimization does not imply global optimization (in fact it may be detrimental). Consequently, a purely local metric may not be sufficient. On the other hand, if a single global metric is used, an LP may not be able to determine if a change in the measured value of the metric occurred due to some action that it performed (or that it should have performed). Thus a combination of a global and a local metric seems desirable. We propose the following local metric to be used in conjunction with R_C:

- P_{RB} - the percentage of events rolled back at an LP

Note, P_{RB} combines the number and depth of rollbacks. R_C indicates the overall progress of the simulation while P_{RB} indicates to an LP whether it is a potential cause for any observed poor global performance. Table 3 summarizes the actions an LP takes under various situations. In [PaWi93], the authors suggest the use of similar metrics. Similar to R_C, they define W_i as the rate of commitment of events at LP_i in one GVT cycle. Thus, W_i is a local metric. While [PaWi93] argue for an indicator of rollback behavior, they do not suggest one — the algorithm they propose uses W_i only.

R_C	P_{RB}	Action
Low	Low	Increase aggressiveness/risk rapidly
Low	High	Decrease aggressiveness/risk rapidly
High	Low	Increase aggressiveness/risk gradually
High	High	Decreaseaggressiveness/riskgradually

Table 3 - Actions for dynamic tuning of parameters

Maximizing global performance by tuning *n* independent local parameters simultaneously is very difficult — in all likelihood, it is NP-hard. However, there may exist heuristics that approach the maximum performance. These heuristics could be based on the metrics proposed above (R_C and P_{RB}).

8.4 Tunable Parameter for ETA

The tunable parameter in ETA is the slope, *s*. Recall that the waiting period in ETA, δ, is a function of the error potential, EP — the slope is merely a scaling factor in this function. *Even with a fixed value of slope, ETA is adaptive because (i) the waiting period at each LP is proportional to its own EP, and (ii) this waiting period changes with EP.* In effect, we can think of *s* as translating the EP from the logical time domain of the application to the appropriate range of real-time delays, while the EP itself adaptively controls the variation in the delay. Thus it is reasonable to treat *s* as a global tunable parameter. The experimental results in Section 6.3 support this reasoning in that in each of those experiments, we have used a single, global value of slope. By the same reasoning, we believe the value of *s* that most reduces the completion time will be constant for most applications, i.e., once this value is determined, there is no need to change it. Once again, this is corroborated by the experimental results of Section 6.3 where the value of global slope remains constant throughout the simulation. It is important to note that the constant nature of the slope value holds even when the workload exhibits phases (as demonstrated in Figure 35 in Section 6.3. When such phase behavior occurs in the workload, it is the EP that reacts to the phases and controls the waiting periods appropriately to mitigate the effects of the phases. This re-iterates the point that *s* is merely a scaling factor — the proper relative delay is achieved by computing EP appropriately. Thus, for a large number of applications, we may treat *s* as a global, static parameter.

It is possible there are applications which undergo *shifts* in the logical time domain during the simulation (for instance if the timestamp distributions are changed for the entire simulation). In such cases, conceivably, *s* will have to be re-tuned during the simulation, making it a global, dynamic tunable parameter. We believe such applications are rare. The performance analyses in the PDES literature have not considered such applications.

Finally, it is conceivable there are applications where the EP is inaccurate for a subset of the LP's (recall EP is only an *estimate* of the likelihood of rollback). The inaccuracy in EP can be compensated for by having different values of slope at different LP's (i.e. slope becomes a local parameter). However, since it is difficult to tune local parameters efficiently, a preferable solution is to improve the mapping M_1 so as to reduce the inaccuracies in EP. Chapter 9 discusses some enhancements to M_1 such as incorporating rollback and lookahead information and choosing a good predecessor set.

In summary, we have argued that for ETA, the applications of general interest lie in quadrant QI of Table 2. We believe that this is generally not the case with other adaptive algorithms described in Chapter 2. For example, the two window sizes in [Stei93] (N_1 and N_2) control the optimism of an LP directly. Consequently, they must be independent for each LP for the algorithm to perform well with a workload that exhibits alternating phases (such as the one we have used to test ETA in Section 6.3.3.2). Similarly, the blocking window size in [BaHo90] and the scaling factor c_i in [HaTr94] are defined locally for each LP.

8.5 Algorithm for Tuning Slope

We use R_C, the global metric described earlier, to automatically tune the value of slope. Based on the trade-off shown in Figure 10 and the experimental results described in Chapter 6, we hypothesize that the behavior or R_C as a function of slope is characterized by one of the three graphs shown in Figure 60. Part (a) shows the typical trade-off situation where R_C is maximized (i.e. the completion time is minimized) for some non-zero, finite value of slope. Part (b) shows the situation where R_C is maximized as slope tends to ∞ (i.e. when the simulation is conservative) and part (c) shows the situation where R_C is maximized when slope is zero (i.e. with Time Warp). In each case, the function is unimodal. The

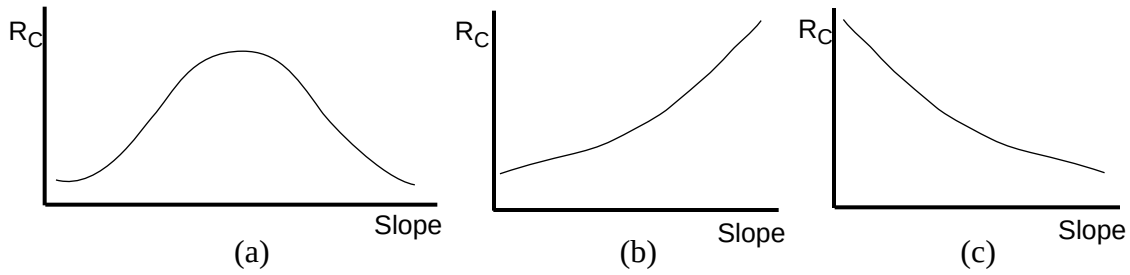


Figure 60 - Hypothesized behavior of R_C

```
AutoSlope (Rc, current_slope)
{
  delta_rc = Rc - last_rc
  abs_delta_rc = |delta_rc|
  delta_slope = current_slope - last_slope
  max_delta_rc = MAX (max_delta_rc, abs_delta_rc)

  If performing initial scan
     If (Rc > max_rc)                {remember the maximum observed}
        max_rc = Rc                  {R_C and}
        max_slope = current_slope    {corresponding slope}
     Endif
     new_slope = current_slope * 10
  Else if first sample after initial scan
                                      {get another data point to start}
     new_slope = max_slope * 1.1     {small perturbation}
     last_rc = max_rc                {first data point is max_rc}
     last_slope = max_slope          {at max_slope}
  Else
     If variation in observed Rc < ε * Rc
        continue in previous direction
     Else if (delta_slope is 0) OR (delta_rc/delta_slope > 0)
                                      {climbing}
        new_slope = current_slope * (1 + abs_delta_rc/max_delta_rc)
     Else                             {falling}
        new_slope = current_slope / (1 + abs_delta_rc/max_delta_rc)
     Endif
  Endif
  If (new_slope > MAX_SLOPE)          {bound the delay}
     new_slope = MAX_SLOPE
  Return new_slope
}
```

Figure 61 - `AutoSlope` - algorithm for automatically tuning the slope

aim of the algorithm described here is to automatically locate the value of slope at which R_C is maximized.

The algorithm, `AutoSlope`, is listed in Figure 61. Since slope is a global parameter, the same value must be used by all LP's. Thus, this algorithm is executed by only one

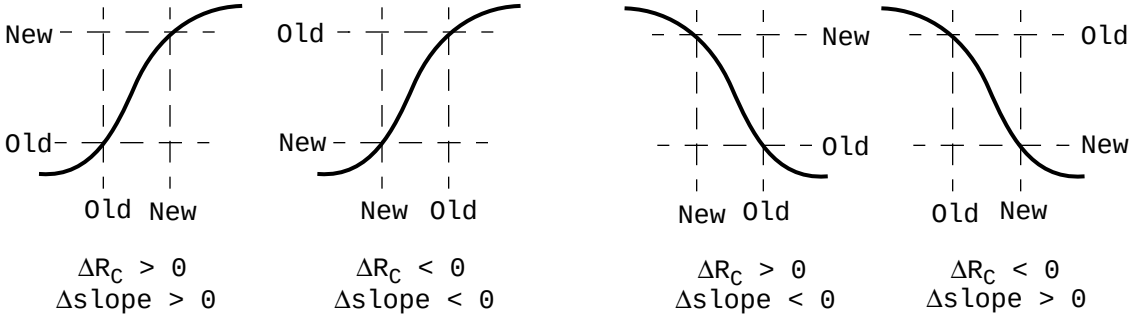


Figure 62 - Determining the direction of slope change

LP and the computed value of slope is distributed to the others. After an initial warm-up period to allow transients to subside, `AutoSlope` is invoked by the main event execution loop periodically. Prior to invoking the procedure, the calling function computes the latest value of R_C as described in Section 8.3.1 and passes this value in the call. `AutoSlope` works by making an initial scan of the range of slope values. For the first few invocations, it multiplies the slope by a constant factor each time, until the slope reaches a predefined maximum value. During this scan, `AutoSlope` retains the largest value of R_C observed and the corresponding slope value. After the scan is complete, `AutoSlope` uses this slope value as the starting point of the tuning process. Since at least two points are required to determine the direction in which to proceed next, the algorithm begins by applying a small perturbation to the starting point.

Given two sample points, the algorithm must determine: (i) the direction in which slope must be varied (i.e. increase or decrease), and (ii) the magnitude of change. The direction of change depends on which side of the maximum the procedure is currently operating in, i.e. if the current slope is on the left side of the maximum, the slope must increase and correspondingly, if the current slope is on the right side of the maximum, the slope must decrease. Since it is generally not possible to know the true maximum of the function, the desired direction is determined by computing the first derivative of the response function, as shown in Figure 62. The slope must increase if the derivative is positive, and it must decrease if the derivative is negative. An increase is effected by multiplying the current slope by a factor that controls the magnitude of change while a decrease is effected by a division.

While the magnitude of the change can be arbitrary, it is desirable to have the change be small when the current slope is near the maximum and large when it is farther away. A possible factor (used in `AutoSlope`) is given by the formula:

$$1 + \frac{\Delta R_C}{MAX\{\Delta R_C\}}$$

Since changes in R_C are smaller near the maximum, this factor should have the desired property.

Finally, while the response function of R_C is theoretically as shown in Figure 60, in practice we may expect the measured value of R_C to exhibit some variance, giving rise to local maxima. To account for this, `AutoSlope` tolerates some amount of variance in R_C by ignoring any changes in the measured value within a small percentage of the absolute value of R_C.

8.6 Performance

Table 4 captures the performance of `AutoSlope` on the various workloads described in Section 6.3.3.2. `AutoSlope` was invoked every 2500 events during the initial scan and every 5000 events thereafter. The tolerance to variance in R_C was set to 3%. For each workload, the table lists the time (in seconds) to complete the simulation using Time Warp, the completion time using ETA with `AutoSlope`, the smallest completion time using ETA by manually tuning the slope and the relative efficiency of ETA with `AutoSlope` compared to the manually tuned version. The efficiency (η) is computed as follows:

$$\eta = \frac{TW - Automatic}{TW - Handtuned} \times 100$$

In the case of EchoSystem, we were unable to obtain data for pure Time Warp because the inherently unstable nature of the workload caused system buffers to overflow for long runs. Nonetheless, it is clear that the automatic scheme was able to eliminate the instability and performed nearly as well as the manually tuned version..

Note, in the design of the automatic tuning scheme it was not assumed that slope is a static parameter (only that it is a global parameter). `AutoSlope` does not settle at a particular value of slope; rather, it tends to keep the slope in a region about the maximum. Therefore, in the event of a major shift in the logical time domain of the simulation, we

Workload	Time Warp	Automatic	Hand-tuned	η
LGRAIN	1047	901.5	882	88.2%
SELF-SYM	870.67	744.25	742.67	98.8%
SELF-BIAS	794	639.67	615.33	86.4%
MSG-SYM	561.67	396.33	373.33	87.8%
MSG-BIAS	843.67	419.75	394.67	94.4%
ASYM	614.33	424.67	397	87.3%
ECHO	—	120.33	102.67	—
CHASE	810	671.67	663.33	94.3%
PHASES-MSG	557	397.33	373.33	86.9%
PHASES-SELF	711.33	593.33	582	91.2%

Table 4 - Efficiency of automatic tuning scheme

believe this algorithm will react by changing the slope accordingly, ultimately reaching the new maximum. If it is known that the simulation will not undergo such major shifts in the logical time domain (as is most often the case), then slope is a global, *static* parameter for this simulation. A pilot run may then be used to determine the region of slope values in which the automatic tuning algorithm settles down so that the slope value may be fixed at some point in this region for the main runs.

8.7 Summary

By definition, an adaptive protocol entails tunable parameters — parameters whose values must be determined in an application-specific manner or adjusted dynamically to ensure good performance. Tunable parameters can be global or local and static or dynamic. They may be tuned dynamically based on the measured values of metrics that reflect the relevant state of the computation. We have identified R_C, a global metric and P_{RB}, a local metric. We have described an algorithm that automatically tunes the value of slope (the tunable parameter of ETA) based on R_C. Performance results show that this algorithm is very efficient (86%—99%) relative to a version in which the slope is tuned manually to minimize completion time.

Chapter 9

Conclusions

In this final chapter, we summarize the results and contributions of our research, identify areas of future research and comment on the significance, strengths and weaknesses of our work.

9.1 Summary of Results

We summarize the results and contributions of this thesis by chapter.

In **chapter 2**, we surveyed previous work related to this thesis. We discussed the merits and demerits of traditional PDES protocols — conservative, optimistic, window-based and error-tolerant. This study leads to the observation that a protocol in which optimism is controlled is desirable. We surveyed protocols that limit optimism, categorizing them based on the criterion for limiting optimism. Since comparative performance results are unavailable, we have reasoned about the pros and cons of each category.

In **chapter 3**, we presented new results in the area of critical path analysis of PDES's. Since simulations must execute events so as not to appear to violate dependence constraints, critical path analysis has been used to study lower bounds on the completion time of PDES's. Our investigation into the optimality of NPSI protocols revealed two interesting points: (i) due to the phenomenon of super-critical speed, there is no consensus on optimality, and (ii) some previously published results concerning super-critical speed are flawed. We presented a correct sufficient condition to determine whether a protocol is super-critical. Also, we showed that a similar condition used previously to argue about the super-criticality of four protocols is necessary but not sufficient. Secondly, we disproved a previous claim that super-criticality requires independence among certain events in the simulation. This result is significant because it suggests that a non-trivial lower bound for all simulations cannot exist, except for special classes of simulations. The results of this chapter have been published in [SrRe95a].

In **chapter 4**, we developed the theory of NPSI protocols. We motivated an approach in which optimism is controlled adaptively using state information. We described

the reduction model of computation underlying NPSI protocols and discussed how this model may be realized in practice. We proposed a framework for the design of NPSI protocols. The main advantage of this framework is that, using two mappings M_1 and M_2, it isolates the two phases in the design of NPSI protocols — identifying relevant state information and controlling optimism using this information. We discussed the general requirements for these mappings and described several possibilities for each. This work has been published in [SrRe95b]

In **chapter 5**, we described how the reduction model of chapter 4 may be used to produce state information required by adaptive protocols. Specifically, we presented *npGVT*, an algorithm to compute the global virtual time (GVT) in an optimistic PDES. Being based on the reduction model, *npGVT* produces accurate values of GVT at low-cost and is thus suitable for use in adaptive protocols. Since computing GVT is an instance of obtaining a distributed snapshot, it is inherently a non-trivial problem (except in special cases). We identified two correctness criteria for GVT algorithms in general and proved that *npGVT* satisfies these. Also, we described an implementation of *npGVT* called *npGVT2P*, in which the reduction model is realized using an asynchronous reduction network. The asynchronous nature of the network introduces the potential for race conditions which are accounted for in *npGVT2P*. We extended the proof of correctness to *npGVT2P*. Finally, we described three properties of *npGVT* (and *npGVT2P*) that are useful in the context of adaptive protocols. *npGVT2P* has been published in [SrRe93a].

In **chapter 6**, we described the Elastic Time Algorithm (ETA), an NPSI protocol based on the framework of chapter 4, by specifying the two mappings M_1 and M_2. ETA is implemented using *npGVT*, the algorithm described in chapter 5. We described a detailed performance study of ETA. The study was conducted over a suite of test workloads with widely different characteristics, which were realized using a synthetic workload generator designed for that purpose. The most important conclusion from this performance study is that ETA consistently outperforms Time Warp, in both completion time and memory consumption. Further, these results suggest that the reduction in completion time produced by ETA is close to the best possible by controlling optimism. For comparison, we presented another NPSI algorithm, ZETA. Although ZETA is an intuitive choice for controlling optimism (other protocols have used similar techniques), we presented a performance study

that shows it to be not as effective as ETA. Early work in this chapter has been published in [SrRe95b].

In **chapter 7**, we presented the first known analytical comparison between adaptive protocols and Time Warp. We described a class of adaptive protocols, which we call the Asynchronous Adaptive Waiting Protocols (AAWP's). This class includes many practical protocols, among them NPSI protocols. We showed that: (i) Time Warp can outperform an AAWP arbitrarily, and (ii) AAWP's (specifically, ZETA, the algorithm described in chapter 6) can outperform Time Warp arbitrarily. Thus, while experimental studies suggest that adaptive protocols enhance performance in general, our analysis establishes a theoretical limitation of a general class of adaptive protocols. This result has been published in [SrRe95c].

In **chapter 8**, we addressed the issue of tunable parameters in adaptive protocols. Tunable parameters are those whose values must be determined in an application-specific manner or adjusted dynamically to ensure good performance. The general approach to tuning parameters is to monitor the progress of the system using relevant metrics. We identified four types of tunable parameters. Based on whether a parameter is global or local, we proposed two metrics, one global and one local. We described an algorithm to automatically tune the single tunable parameter in ETA — slope — using the global metric. Performance study of this algorithm on the workload suite described in chapter 6 shows it to be very efficient.

9.2 Future Research

In this thesis, we have laid the foundation for a new class of PDES protocols — the NPSI adaptive protocols. We have proposed an NPSI adaptive protocol — the Elastic Time Algorithm (ETA). A thorough performance study of this algorithm shows it holds significant potential to be consistently efficient. ETA is the first step in research into NPSI protocols. We describe several issues that will define the direction of future research in this area.

9.2.1 Enhancing M_1

In the context of the framework for NPSI adaptive protocols we proposed in Chapter 4, we note that the higher the accuracy of M_1 in identifying potentially incorrect computations, the less the reliance of M_2 on tunable parameters like *s*. For example, in the ideal

case, if M_1 is sophisticated enough to always identify any computation exactly as either correct or not, then M_2 is simply a binary function that stops event execution (and the sending of messages) when M_1 flags an error. Research is required into identifying other kinds of state information that may be used in evaluating M_1 and that would increase the accuracy of the error potential. We suggest several such enhancements.

Rollback information

In the context of near-perfect information, it is possible to utilize effectively the knowledge that one or more LP's are in the process of rolling back. The fact that an LP is rolling back is important to its successors in the communication topology because rollbacks tend to propagate. Consider the following analogy: visualize an LP as a person driving a car on a highway. When this person observes that the car in front of her's has applied its brake, that is (typically) interpreted as a warning of an impending collision and suggests that she should apply her brakes as well[1]. Similarly, if we are able to provide an LP with the information that its predecessor is rolling back, that LP should interpret that as a warning of impending rollback and hence slow down. Extending the analogy further, if the driver of the car observes activated brake lights on cars farther ahead (than the one immediately ahead of her's), that may be interpreted as a warning of impending braking and accordingly, the driver may cease to accelerate and let the car coast. Correspondingly, if an LP is informed that some of its farther predecessors are rolling back, it may continue to process aggressively, but reduce its risk to zero (i.e. temporarily suspend the sending of messages). There is a strong intuitive appeal in this scheme. However, along with providing information about impending rollbacks, this scheme has a tendency to increase lost opportunity cost as well. This is because a predecessor of LP_i may be rolling back but may never send an antimessage to LP_i. Similarly, rollbacks occurring at farther predecessors may subside before they can reach LP_i.

Memory Information

Excessive memory consumption usually occurs in aggressive systems due to so-called *runaway processes* - LP's that execute events faster than other LP's so that they

1. We disregard the possibility of changing lanes!

schedule a large number of events at these other LP's. By limiting optimism, ETA reduces this memory consumption in general. However, ETA is ineffective in the specific case of runaway processes that are purely sources of messages. Since these LP's have no predecessors, their optimism will not be controlled by ETA. In such cases, an adaptive, memory-based flow control scheme can be integrated naturally with NPSI protocols by including information about the memory availability of successor LP's in the mapping M_1. In this way, a runaway source could slow down when any of its successors is at the risk of running out of memory.

Lookahead

Lost opportunity cost is a direct result of *artificial blocking* [Reyn82], the situation where an LP blocks when it could safely proceed. In non-aggressive protocols, artificial blocking occurs due to incomplete information (i.e. when LP's don't have enough information to determine lack of dependence) whereas in NPSI adaptive protocols, artificial blocking occurs due to inaccurate information. The most effective way of reducing artificial blocking in non-aggressive protocols is to utilize any lookahead [Fuji90] in the application to accelerate the dissemination of information. The same technique may be applied to adaptive protocols. In our framework, lookahead would be incorporated into M_1 to reduce the error potential when possible.

Causality

While timestamps of events and messages are the typical basis for defining causal relationships, as noted in Chapter 3, this is usually pessimistic — it assumes causal dependences when semantically, none exist in the application. Since ETA is based on timestamps also, it suffers the same drawback, i.e. it may overestimate the error potential. Thus, a mapping M_1 that captures the causality of the application directly (not restricted to timestamps) may be desirable.

EP for messages

In the framework of Section 4.3, we associate an EP (error potential) with each LP. Another option is to associate an EP with each message. Since messages generally form *causal chains*, scheduling events as they move over the LP's, this scheme may support finer

control over optimism by allowing a safe causal chain to proceed through an LP while at the same time restricting unsafe ones.

Heuristic target-specific sets

While target-specific information is usually computed over an LP's entire predecessor set, this is not a requirement. The general definition of target-specific information is that for each LP, the information is computed over a possibly unique subset of all LP's. Heuristics may be used to define these subsets, depending on the type of information being computed. For instance, rollback information (see above) may be computed over the set of immediate predecessors only.

9.2.2 Other Areas

The following are some other areas that warrant research:

i) Independent control of risk — as noted in Section 6.1, risk is controlled in ETA only indirectly. An independent control over risk may reduce lost opportunity by allowing LP's to execute events locally rather than block.

ii) Analytical performance study of ETA — the goal would be to prove that ETA always performs as well as or better than Time Warp. Existing models of Time Warp behavior [Gupt95, GuAF91] could be extended to model ETA (and other NPSI protocols).

iii) Identify properties of ETA (and NPSI protocols in general) that avoid the adverse situation described in Chapter 7 where an erroneous adaptive decision leads to an arbitrarily long rollback chain.

iv) Comparative analysis of adaptive protocols with conservative protocols, similar to the comparison presented in Chapter 7.

v) Experimental comparison with other adaptive protocols — very few such comparisons exist in the PDES literature.

vi) Design and implementation of a scalable target-specific reduction system in shared memory as well as on distributed memory machines.

9.3 Significance, Strengths and Weaknesses

The most conclusive result of several years of PDES research is that the synchronization problem in PDES is hard [Fuji90]. A large number of protocols have been proposed to solve this problem. However, each has its pitfalls, which restrict its applicability. There has been no single protocol that consistently performs well. In fact, several efforts have focused on designing protocols for specific applications [NiRe84, FuNi92]. Among other reasons, we believe the lack of a consistently efficient protocol is a significant contributor to the fact that fifteen years of research in PDES have made almost no impact on the general simulation industry [Fuji93a, NiHe95]. Consequently, we concur with [Reyn93] that a consistently efficient protocol is still one of the most important goals for PDES researchers. The NPSI adaptive approach is a significant step in achieving this goal.

The two main strengths of our approach are (i) it is adaptive, and (ii) it uses near-perfect state information. It is well known that the synchronization requirements of parallel simulations are irregular, highly data-dependent and dynamic [Fuji90, NiRe90]. Being adaptive, our approach can satisfy the dynamic nature of these requirements. To satisfy the irregular, data-dependent nature of the requirements, approaches have been proposed that either (i) estimate the non-local state of the system using locally available information, or (ii) predict the non-local state in the immediate future based on complex statistical analysis of the recent history of the simulation. Both of these are approximations of the desired state information. We believe the direct approach, which assumes the availability of such state information, will be able to make the correct adaptive decisions and thus satisfy the synchronization requirements of PDES. In this thesis, we have demonstrated that the NPSI adaptive approach does indeed hold significant potential to be consistently efficient.

It may be argued that one of the strengths of our approach, namely the assumption of availability of low-cost near-perfect state information, is also its weakness. As a counter-argument, we justify the NPSI assumption as follows:

i) We believe protocols that use state information (state-based protocols) have significant potential to be consistently efficient. As discussed before, many kinds of state information can be utilized effectively to improve simulation performance. Thus, state-based protocols warrant a thorough study. Recent activity in this area of research is encouraging.

ii) We do not assume the availability (at each LP) of large amounts of detailed, non-local state information — clearly, such a model would be extremely difficult to realize in practice. As in adaptive load sharing in distributed systems [EaLZ86], we believe simple (easily computed and disseminated) information can produce significant benefits for PDES. Therefore, we assume a simple reduction model that captures the relevant state information at each LP in a small set of values (floating point or fixed point numbers) and provides each LP with a consistent snapshot of these values in a reduced form (i.e. as minima, sums, logical OR's, etc.).

iii) The reduction model we have assumed is a very general one — it makes no assumptions specific to the PDES problem. Non-local information, even in a reduced form, can benefit most parallel computations. For example, it can be used to implement common synchronization constructs such as barriers, which are used frequently in numerical computations. Other areas where the reduction model is expected to prove useful are dynamic load balancing [Kirk92] and isotach concurrency control for parallel systems [ReAW95], a technology that is expected to revolutionize parallel programming by eliminating the need for wasteful synchronization constructs such as locks. The usefulness of reductions is apparent in the fact that many state-of-the-art high performance architectures support reductions either in hardware (CM-5) or in software (T3D, Intel).

iv) The reduction model can be realized in practice. It can be implemented directly in shared memory or using a high-speed asynchronous reduction network. A recent proposal [ReAW95] to incorporate a reduction network into a network of workstations connected by ATM switches indicates reduction networks are compatible with future platforms.

9.4 Concluding Remarks

Historically, research in synchronization algorithms (protocols) for parallel discrete event simulation (PDES) has followed two tracks: conservative and optimistic. Neither approach is consistently efficient (i.e. performs well over a broad range of applications).

This is due primarily to the irregular, highly data-dependent and dynamic nature of synchronization requirements in PDES. Our thesis is that an adaptive approach based on low cost near-perfect state information has significant potential to yield a consistently efficient synchronization algorithm. We have substantiated that thesis.

We have introduced a new class of protocols called NPSI adaptive protocols. These are characterized by the use of low-cost near-perfect state information to adaptively control the optimism of processes in a PDES. We have proposed a framework for the design of NPSI adaptive protocols. Based on this framework, we have described an NPSI protocol, the Elastic Time Algorithm (ETA) and studied its performance. From this study, it is clear that NPSI adaptive protocols have significant potential to be consistently efficient. The investigation of NPSI adaptive protocols has only just begun. The design space for these protocols is large and must be explored intelligently. We have indicated areas for future research.

Appendix I

A spectrum of options for parallel simulation

Traditionally, PDES synchronization algorithms have been classified broadly as either *conservative* or *optimistic*. Reynolds [Reyn88] presents a detailed taxonomy of protocols based on nine design variables (or dimensions), which we summarize here.

Partitioning: determine clusters of LP's based on distinct sets of design variable bindings

The set of LP's is partitioned into the set $\{p_1, p_2, ... p_m\}$ where each p_i includes a non-empty set of LP's. Traditionally, $m = 1$, but in general, $m \geq 1$. Each p_i may have a different set of bindings for the other eight design variables. For instance, one set of LP's may operate aggressively while another may operate non-aggressively.

Adaptability: change design variable bindings based on knowledge of selected aspects of the simulation state

In the course of the simulation, an LP may change its bindings for the design variables. Thus, for example, it may employ a non-aggressive strategy for some time and then switch to an aggressive one.

Aggressiveness: processing messages based on conditional knowledge

When processing aggressively, an LP may execute events (process messages) without the guarantee that a message with a smaller timestamp will not arrive later (i.e. a causality error will not occur). A non-aggressive LP is one which executes events based only on unconditional knowledge of freedom from causality errors. Aggressive processing is central to Time Warp [Jeff85] while non-aggressiveness is found in conservative protocols such as Null Messages [ChMi79].

Accuracy: requiring that events within LP's *appear to be* executed without causality errors

An accurate algorithm is one that guarantees freedom from causality errors. The outcome of accurate execution is that the set of all final states of any parallel execution is

equivalent to the set of all final states of a sequential execution of the simulation. Note, while accuracy is desirable, it may not always be mandatory. It may be cost-effective to allow errors into the simulation (rather than incur the overheads of repair) when these errors are insignificant and can be hidden in the inherent variance of the simulation output.

Risk: propagating messages which have been processed based on aggressive or inaccurate processing

An LP with risk may send out messages that have been generated as a result of aggressive processing (which is not guaranteed to be correct, yet) or inaccurate processing (which is known to be slightly incorrect). An LP without risk sends out messages only after it can determine that it is safe to do so.

Knowledge Embedding: knowledge about LP's behavioral attributes is embedded in the simulation

Knowledge about the simulated system may be embedded in the simulation or the protocol. The latter approach is more flexible. This knowledge can be used during the simulation to determine the safety of events as in the Conditional Knowledge approach of [ChSh89] or the use of lookahead in non-aggressive windowing schemes [Luba89, AyRa92]. Use of embedded knowledge prevents a simulation protocol from being completely general.

Knowledge dissemination: LP's initiate the transmission of knowledge to other LP's

Knowledge dissemination occurs when an LP sends messages to other LP's that may create the opportunity for them to make progress. Examples of this are the null messages in [ChMi79] and the appointments scheme in [NiRe84]. Voluntary dissemination may generate messages that are never used.

Knowledge acquisition: LP's initiate requests for knowledge from other LP's

These are also called "demand-based" methods. Information is released based on demand for it. An LP requiring information to determine the safety of pending events sends out messages for the same. Knowledge acquisition and knowledge dissemination are orthogonal; either can be used to any degree independently of the use of the other.

Synchrony: degree of temporal binding among LP's

This variable determines how much the temporal progress of individual LP's is controlled by the that of other LP's, i.e., how often must LP's synchronize. In the first protocols, the LP's were completely asynchronous of each other. At the other extreme, a SIMD-like approach may be followed in which LP's synchronize after advance each unit of time. This is the time-stepped approach. A third option is termed *synchronous* in which LP's execute alternating periods of simulation and global synchronization. This is also called the *iterative* approach. An important class of iterative protocols are the *windowing* protocols in which a logical time window is established co-operatively and LP's execute events independently inside the time window.

Appendix II

Time Warp

Time Warp [Jeff85] is an optimistic protocol. The driving rule for each LP is: *whenever there is an event pending, process it*. Messages sent out during such *forward* processing are called *positive* messages. As messages arrive on channels, they are scheduled as events for execution. Thus, an LP does not block waiting for messages to arrive on all input channels (as in some conservative protocols). This eliminates the problem of deadlock. However, no assumptions are made about the order in which messages are received by an LP or the relative speeds of the processors on which the LP's are executing. Thus, it is possible for a *causality error* to occur, where an LP receives and processes messages out of timestamp order. The LP then *rolls back* over the incorrect computation and restarts from a previously saved correct state. Note, this requires that the LP's *checkpoint* their states periodically. When rolling back, an LP many have to *annihilate* some previously sent positive messages that are rendered incorrect due to the rollback. This is achieved by sending an *antimessages* for corresponding positive messages.

Important data structures

An LP maintains several data structures in order to be able to perform rollbacks. The essential ones are:

i) *Logical clock*: This indicates the time up to which this LP has been simulated. Due to rollbacks, the value of this clock may move backward as well as forward. This is a characteristic of aggressive protocols.

ii) *Events list*: This is a list of future events that the LP expects to execute, sorted in ascending order of the timestamps of the events. Events get scheduled on this list as a result of execution of other events, or arrival of messages from other LP's. As messages arrive, they are converted into events, which are incorporated into the events list. Events may even get deleted as a result of the execution of other events.

iii) *Saved state list*: The LP must save its state periodically so that it can restart execution from a prior, correct state in the event of a rollback. This list is also sorted in ascending order of the times at which the state was saved.

iv) *Output message list*: The LP maintains a list of antimessages corresponding to the positive messages it sent out. Some (or all) of these may be sent out during rollback. The list is sorted in ascending order of the timestamps of the antimessages.

Mechanisms

When processing forward, an LP moves ahead in logical time, removing events from the events list and executing them. The logical clock of the LP is set to the time of the event currently being executed. Whenever it sends out a positive message, it inserts the corresponding antimessage in its output message list. Antimessages differ from their positive counterparts in one respect only: the sign. Positive messages have a positive sign, while antimessages have a negative sign. Antimessages serve the purpose of undoing the effects of erroneous positive messages. An LP receiving an antimessage must undo the effects (if any) of the corresponding positive message. Periodically, the LP also saves its state in the saved state list.

When a causality error occurs due to the arrival of a message in an LP's logical past (called a *straggler*), the LP reverses the execution of all events with a timestamp greater than that of the straggler (hereafter referred to as the rollback time, T_r). This may cause several locally scheduled events to be deleted from the events list. If the straggler is an antimessage, the corresponding positive message is annihilated (removed and destroyed). The LP then discards all states saved after T_r , restores its state at T_r from the last state saved at or before T_r and sets its logical clock to T_r . Note that if the state was not saved at T_r or the event before it, the LP has to rebuild its state by re-executing all of the events between the last saved state and T_r . Finally, it sends out all of the antimessages in the output message list with timestamp greater than T_r . This completes the rollback process and the LP starts forward processing once again.

An LP may receive an antimessage while in one of three states:

i) The corresponding positive message has been received but not processed yet. In this case, the antimessage annihilates the positive message and the two disappear.

ii) The corresponding positive message has been received and processed. The antimessage (which is now a straggler) causes the LP to roll back to the time of the positive message, annihilate it and restart from that point. As a result of the rollback, it may send out antimessages, which cause secondary rollbacks in other LP's (*cascading* rollbacks).

iii) The corresponding positive message has not yet been received. The antimessage is stored and when the positive message does arrive, it is immediately annihilated and there is no rollback.

GVT

Global virtual time (GVT) is defined as the minimum of the logical clocks of all LP's and the timestamps of any messages that are outstanding (have been sent but not yet received and/or incorporated into the events lists of receiving LP's). GVT is an indicator of how far the simulation has progressed in logical time. Even though the logical clocks of individual LP's move backward as well as forward, GVT never decreases: it is monotonic non-decreasing. Thus, GVT defines a *commitment horizon* [Jeff85]. Once GVT crosses the time of any event, that event can never be rolled back again and can therefore be committed. Until this happens, no event can be committed that has effects that are permanent (I/O for example) or hard to undo. Once an event is committed, an LP can release all the memory held by that event. As GVT progresses, LP's discard all saved entities (events, input messages, output antimessages, states) with timestamps smaller than GVT. This is called *fossil collection*. An exception is the list of saved states. Here, at least one state older than or at GVT must be saved to allow the LP to restore its state properly, in case of a rollback to GVT.

Cancellation strategies

Several approaches have been suggested to perform cancellation of unintended messages [Gafn88, RFBJ90]. In *aggressive cancellation*, whenever an LP rolls back, it sends out antimessages for all erroneous positive messages before restarting forward processing (this is the scheme described above). In *lazy cancellation*, as the name suggests, the LP does not send an antimessage until it can determine if the corresponding positive message was indeed incorrect. Typically, it restarts forward processing and when it reaches the time of the original positive message, it determines whether that message was incorrect or not. If not, it sends out an antimessage.

Event preemption

Event preemption concerns the situation when a message arrives when an LP is processing an event such that the message will invalidate the execution of that event. There are two options regarding the receiving of messages: polling versus preemption. With polling, an LP checks to see if new messages have arrived after the execution of each event. In the situation described above, it may unnecessarily process an event and immediately roll it back. A more efficient way is to have an incoming straggler preempt the execution of the current event and cause the LP to roll back immediately. This is referred to as message or event preemption.

Appendix III

Proof of Non-decreasing α_i'

Recall the following definitions from Chapter 6:

σ_i = logical clock of LP_i — timestamp of the event being executed currently or just executed.

υ_i = minimum unreceived message time of LP_i, i.e. it is the smallest among the timestamps of all messages sent by LP_i that have not yet been received and/or incorporated into the events list of the receivers.

η_i = next event time of LP_i, defined as: σ_i while LP_i executes an event; otherwise, the smallest among the timestamps of events scheduled to be executed at LP_i.

$\alpha_i = MIN(\eta_i, \upsilon_i)$

$\alpha_i' = MIN_{LP_k}(\alpha_i) \ \forall LP_k \in$ the predecessor set of LP_i (defined in Section 4.2.1)

For a minimum value such as α_i', we say the individual values used in the minimum computation *define* the computed minimum value. We prove the following theorem:

Theorem: α_i' is non-decreasing.

Proof: By contradiction. For some LP_i, assume two successive values of α_i' (α^1 and α^2) represent a decrease — i.e., $\alpha^1 > \alpha^2$. By definition, $\alpha^2 = \eta_j$ for some LP_j predecessor of LP_i, or $\alpha^2 = \upsilon_k$ for some LP_k predecessor of LP_i, or both (with *j* possibly equal to *k*). We consider the two cases separately:

<u>$\alpha^2 = \eta_j$ for some LP_j predecessor of LP_i</u>

$\alpha^1 > \alpha^2 = \eta_j \Rightarrow$ the value of η_j defining α^1 (call it η_j^1) is greater than that defining α^2 (call it η_j^2). This decrease can occur only by the arrival of a message (either causing a rollback or scheduling an event with timestamp $< \eta_j^1$) sent by some LP_k. Since this message is considered unreceived by LP_k at least until it is incorporated into LP_j's events list

and LP_k is also a predecessor of LP_i (by transitivity), the value of υ_k defining α^1 must be $\leq \eta_j^2$. Thus, $\alpha^1 \leq \eta_j^2 = \alpha^2$.

$\alpha^2 = \upsilon_j$ for some LP_j predecessor of LP_i

$\alpha^1 > \alpha^2 = \upsilon_j \Rightarrow$ the value of υ_j defining α^1 (call it υ_j^1) is greater than that defining α^2 (call it υ_j^2). This decrease can occur only if LP_j sends a message with a timestamp $\upsilon_j^2 < \upsilon_j^1$. Since the timestamp of a message equals the timestamp of the send event (therefore, also the logical clock of the LP sending the message) and by definition, η_j equals the logical clock of LP_j while it executes the send event, the value of η_j defining α^1 must be $= \upsilon_j^2$. Thus, $\alpha^1 \leq \upsilon_j^2 = \alpha^2$. ■

Bibliography

[ArSm92] Arvind, D.K. and Smart, C.R., “Hierarchical parallel discrete event simulation in Composite Elsa”, *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, January 1992, 147-155.

[AyRa92] Ayani, R. and Rajaei, H., “Parallel simulation using conservative time window”, *Proceedings of the 1992 Winter Simulation Conference*, December 1992, 709-717.

[BaHo90] Ball, D. and Hoyt, S., “The adaptive Time-Warp concurrency control algorithm”, *Proceedings of the SCS Multiconference on Distributed Simulation*, January 1990, 174-177.

[Bell93] Bellenot, S., “Performance of a riskfree Time Warp operating system”, *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, May 1993, 155-158.

[BeJe85] Berry, O. and Jefferson, D, “Critical path analysis of distributed simulation”, *Proc. of the 1985 SCS Conference on Distributed Simulation*, January 1985, 57-60.

[Berr86] Berry O., “Performance evaluation of the Time Warp distributed simulation mechanism”, Ph.D. thesis, University of Southern California, 1986.

[Blel89] Blelloch, G., “Scans as primitive parallel operations”, *IEEE Transactions on Computers*, Vol. 38, November 1989, 1526-1538.

[ChMi79] Chandy, K.M. and Misra, J., “Distributed Simulation: A case study in the design and verification of distributed programs”, *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 5, September 1979, 440-452.

[ChMi81] Chandy, K.M. and Misra, J., “Asynchronous distributed simulation via a sequence of parallel computations”, *Communications of the ACM*, Vol. 24, No. 11, April 1981, 198-206.

[ChSh89] Chandy, K.M. and Sherman, R., “The conditional event approach to distributed simulation”, *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989, 93-99.

[CrKn85] Crockett, T. and Knott, J., “System software for the finite element machine”, NASA contractor rep. 3870, NASA Langley, 1985.

[Cray93] Cray Research, “CRAY T3D system architecture overview”, Cray Research Inc., March 1993.

[Cray95] Cray Research, “MPP Fortran programming model”, Available by anonymous FTP from: `ftp.cray.com/product-info/program_env/program_model.html`

[DaFu94] Das, S. and Fujimoto, R.M., “An adaptive memory management protocol for Time Warp parallel simulation”, *Proceedings of SIGMETRICS ‘94*, May 1994, 201-210.

[Dick93] Dickens, P.M., “Analysis of an aggressive global windowing algorithm”, Ph.D. Thesis, Department of Computer Science, University of Virginia, May 1993.

[DiRe90] Dickens, P.M. and Reynolds, P.F., Jr., “SRADS with local rollback”, *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, January 1990, 161-164.

[EaLZ86] Eager, D.L., Lazowska, E.D. and Zahorjan, J., “Adaptive load sharing in homogeneous distributed systems”, *IEEE Transactions on Software Engineering*, Vol. SE-12, May 1986, 662-675.

[FeZh87] Ferrari, D. and Zhou, S., “An empirical investigation of load indices for load balancing applications”, Report number CSD-87-353, Computer Science Division, University of California at Berkeley, 1987.

[FeTr94] Ferscha, A. and Tripathi, S.K., “Parallel and distributed simulation of discrete event systems”, University of Maryland at College Park Technical Report number CS-TR-3336, August 1994.

[Fuji90] Fujimoto, R.M., “Parallel discrete event simulation”, *Communications of the ACM*, Vol. 33, No. 10, October 1990, 30-53.

[Fuji93a] Fujimoto, R.M., “Parallel discrete-event simulation: Will the field survive?”, Feature article in *ORSA Journal on Computing*, Vol. 5, No. 3, Summer 1993, 213-230.

[Fuji93b] Fujimoto, R.M., “Future directions in parallel simulation research”, Rejoinder in *ORSA Journal on Computing*, Vol. 5, No. 3, Summer 1993, 245-248.

[FuHy95] Fujimoto, R.M. and Hybinette, M., “Computing global virtual time in shared-memory multiprocessors”, Technical report, College of Computing, Georgia Institute of Technology, 1995.

[FuNi92] Fujimoto, R.M. and Nicol, D.M., “State of the art in parallel simulation”, *Proceedings of the 1992 Winter Simulation Conference*, December 1992, 246-254.

[FuTG88] Fujimoto, R.M., Tsai, J.J. and Gopalakrishnan, G.C., “Design and performance of special purpose hardware for Time Warp”, *Proceedings of the 15th Annual Symposium on Computer Architecture*, June 1988, 410-408.

[Gafn88] Gafni, A., “Rollback mechanisms for optimistic distributed simulation systems”, *Proceedings of the SCS Multiconference on Distributed Simulations*, Vol. 19, No. 3, July 1988, 61-67.

[Gima89] Gimarc, R.L., “Distributed simulation using hierarchical rollback”, *Proceedings of the 1989 Winter Simulation Conference*, 1989, 621-629.

[GuAF91] Gupta, A., Akyildiz, I.F. and Fujimoto, R.M., “Performance analysis of Time Warp with multiple homogenous processors”, *IEEE Transactions on Software Engineering*, Vol. 17, No. 10, October 1991, 1013-1027.

[Gupt95] Gupta, A., “Effect of heterogeneity on Time Warp performance”, Unpublished document received from author at Intel corporation.

[Gunt94a] Gunter, M., “Understanding supercritical speedup”, *Proceedings of the 1994 Workshop on Parallel and Distributed Simulation*, July 1994, 81-87.

[Gunt94b] Gunter, M., Private communication, November 1994.

[HaTr94] Hamnes, D.O. and Tripathi, A., “Evaluation of a local adaptive protocol for distributed discrete event simulation”, *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994, Vol. III, 127-134.

[Hosh89] Hoshino, T., “Pax computer”, Addison-Wesley, Reading MA, 1989.

[Hybi95] Hybinette, M., Private communication.

[Jeff85] Jefferson, D.R., “Virtual time”, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, 404-425.

[JeRe91] Jefferson, D. and Reiher, P., “Supercritical speedup”, *Proceedings of the 24th Annual Simulation Symposium*, April 1991, 159-168.

[Kirk92] Kirks, D., “A new approach to load-sharing”, Dissertation proposal, Department of Computer Science, University of Virginia, September 1992.

[LaKe91] Law, A. and Kelton, D., “Simulation modeling and analysis”, McGraw-Hill, Inc., 2nd edition, 1991.

[LinJ92] Lin, J-J., “Efficient parallel simulation for designing multiprocessor systems”, Ph.D. thesis, University of Michigan, 1992.

[LinY92a] Lin. Y-B., “Memory management algorithms for optimistic parallel simulation”, *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, January 1992, 43-52.

[LinY92b] Lin, Y-B., “Parallelism analyzers for parallel discrete event simulation”, *ACM Transactions on Modeling and Computer Simulation*, Vol. 2, No. 3, July 1992, 236-264.

[LinY93] Lin Y-B., “Will parallel simulation research survive?”, Commentary in *ORSA Journal on Computing*, Vol. 5, No. 3, Summer 1993, 236-238.

[LiMi90] Lipton, R.J. and Mizell, D.W., “Time Warp vs. Chandy-Misra: A worst-case comparison”, *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, January 1990, 137-143.

[LiLa91] Lin, Y-B. and Lazowska, E.D., “A study of Time Warp rollback mechanisms”, *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 1, January 1991, 51-72.

[Livn85] Livny, M., “A study of parallelism in distributed simulation”, *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, January 1985, 94-98.

[Luba89] Lubachevsky, B.D., “Efficient distributed event-driven simulations of multiple-loop networks”, *Communications of the ACM*, Vol. 32, No. 1. January 1989, 111-123.

[LuWS89] Lubachevsky, B., Weiss, A. and Shwartz, A., “Rollback sometimes works . . . if filtered”, *Proceedings of the 1989 Winter Simulation Conference*, December 1989, 630-639.

[LuWS91] Lubachevsky, B., Weiss, A. and Shwartz, A., “An analysis of rollback-based simulation”, *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 2, April 1991, 154-193.

[Madi93] Madisetti, V.K., “Randomized algorithms for self-synchronization”, Private communication, 1993.

[MaWM88] Madisetti, V.K., Walrand, J. and Messerschmitt, D., “WOLF: A rollback algorithm for optimistic distributed simulation systems”, *Proceedings of the 1988 Winter Simulation Conferences*, December 1988, 296-305.

[MaHF92] Madisetti, V.K., Hardaker, D.A. and Fujimoto, R.M., “The MIMDIX operating system for parallel simulation”, *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, January 1992, 65-74.

[McAf90] McAffer, J., “A unified distributed simulation system”, *Proceedings of the 1990 Winter Simulation Conference*, 1990, 415-422.

[Mehl91] Mehl, H., “Speed-up of conservative distributed discrete event simulation methods by speculative computing”, *Proceedings of the 5th Workshop on Parallel and Distributed Simulation*, January 1991, 163-166.

[Movv93] Movva, R., “Controlled inaccuracies in parallel discrete event simulation”, Master of Science Thesis, Department of Computer Science, University of Virginia, May 1993.

[Nico91] Nicol, D.M., “Performance bounds on parallel self-initiating discrete-event simulations”, *ACM Transactions on Modeling and Computer Simulations*, Vol. 1, No. 1, 1991, 24-50.

[Nico93] Nicol, D.M., “The cost of conservative synchronization in parallel discrete event simulations”, *Journal of the ACM*, Vol. 40, No. 2, April 1993, 304-333.

[NiHe95] Nicol, D.M. and Heidelberger, P., “On extending parallelism to serial simulators”, *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, June 1995, 60-67.

[NiRe84] Nicol, D.M. and Reynolds, P.F., Jr., “Problem-oriented protocol design”, *Proceedings of the 1984 Winter Simulation Conference*, December 1984, 471-474.

[NiRe90] Nicol, D.M. and Reynolds, P.F., Jr., “Optimal dynamic remapping of parallel computations”, *IEEE Transactions on Computers*, Vol. 39, No. 2, February 1990, 206-219.

[ORSA93] *ORSA Journal on Computing*, Vol. 5, No. 3, Summer 1993, 213-248.

[Panc92] Pancerella, C.M., “Improving the efficiency of a framework for parallel simulations”, *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, January 1992, 22-27.

[Panc94] Pancerella, C.M., “Reduction operations in parallel simulations”, Ph.D. dissertation, Dept. of Computer Science, University of Virginia, May 19, 1994.

[PaRe93] Pancerella, C.M. and Reynolds, P.F., Jr., “Disseminating critical target-specific synchronization information in parallel discrete event simulations”, *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, May 1993, 52-59.

[PaWi92] Palaniswamy, A.C. and Wilsey, P.A., “An analytical comparison of periodic checkpointing and incremental state saving”, *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, May 1993, 127-134.

[PaWi93] Palaniswamy, A.C. and Wilsey, P.A., “Adaptive bounded time window in an optimistically synchronized simulator”, *Proceedings of the third Great Lakes symposium on VLSI*, 1993, 114-118.

[PeWM79] Peacock, J.K., Wong, J.W., and Manning, E.G., “Distributed simulation using a network of processors”, *Computer Networks* Vol. 3, No. 1, February 1979, 44-56.

[Ponn93] Ponnusamy, R., et.al. “Experimental performance evaluation of the CM-5”, *Journal of Parallel and Distributed Computing*, Vol. 19, 1993, 192-202.

[PrSu91] Prakash, A. and Subramanian, R., “Filter: An algorithm for reducing cascaded rollbacks in optimistic distributed simulations”, *Proceedings of the 24th Annual Simulation Symposium*, April 1991, 123-132.

[RaAT93] Rajaei, H., Ayani, R. and Thorelli, L-E., “The Local Time Warp approach to parallel simulation”, *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, May 1993, 119-126.

[ReJe89] Reiher, P.L. and Jefferson, D., “Limitation of optimism in the Time Warp operating system”, *Proceedings of the 1989 Winter Simulation Conference*, December 1989, 765-770.

[RFBJ90] Reiher, P., Fujimoto, R.M., Bellenot, S. and Jefferson, D., “Cancellation strategies in optimistic execution systems”, *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, January 1990, 112-121.

[Reyn82] Reynolds, P.F., Jr., “A shared resource algorithm for distributed simulation”, *Proceedings of the 9th Annual Symposium on Computer Architecture*, April 1982, 259-266.

[Reyn88] Reynolds, P.F., Jr., “A spectrum of options for parallel simulation”, *Proceedings of the 1988 Winter Simulation Conference*, December 1988, 325-332.

[Reyn92] Reynolds, P.F., Jr. “An efficient framework for parallel simulations”, *International Journal on Computer Simulation* Vol. 2, No. 4, 1992, 427-445.

[Reyn93] Reynolds, P.F., Jr., “The Silver Bullet”, Commentary in *ORSA Journal on Computing*, Vol. 5, No. 3, Summer 1993, 239-241.

[ReAW95] Reynolds, P.F., Jr., Aylor, J.H. and Williams, R.D., “Isotach engines: transparent coordination of distributed computations”, Research proposal, Departments of Computer Science and Electrical Engineering, University of Virginia, April 1995.

[RePS92] Reynolds, P.F., Jr., Pancerella, C.M. and Srinivasan, S., “Making parallel simulations go fast”, *Proceedings of the 1992 Winter Simulation Conference*, December 1992, 646-656.

[RePS93] Reynolds, P.F., Jr., Pancerella, C.M. and Srinivasan, S., “Design and performance analysis of hardware support for parallel simulations”, *Journal of Parallel and Distributed Computing*, Vol. 18, 1993, 435-453.

[SES] Scientific and Engineering Software, Inc., “SES/Workbench user’s manual”, Austin, TX.

[SoBW88] Sokol, L.M., Briscoe, D.P. and Wieland, A.P., “MTW: a strategy for scheduling discrete events for concurrent execution”, *Proceedings of the SCS Multiconference on Distributed Simulation*, July 1988, 34-42.

[Srin92] Srinivasan, S., “Modeling a framework for parallel simulations”, Master of Science Thesis, Computer Science Department, University of Virginia, May 1992.

[SrRe93a] Srinivasan, S. and Reynolds, P.F., Jr., “Non-interfering GVT computation via asynchronous global reductions”, *Proceedings of the 1993 Winter Simulation Conference*, December 1993, 740-749.

[SrRe93b] Srinivasan, S. and Reynolds, P.F., Jr., “Super-criticality revisited”, Technical report number CS-93-29, Dept. of Computer Science, University of Virginia, May 28, 1993.

[SrRe95a] Srinivasan, S. and Reynolds, P.F., Jr., “Super-criticality revisited”, *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, June 1995, 130-136.

[SrRe95b] Srinivasan, S. and Reynolds, P.F., Jr., “NPSI adaptive synchronization algorithms for PDES”, To appear in the *1995 Winter Simulation Conference*.

[SrRe95c] Srinivasan, S. and Reynolds, P.F., Jr., “Adaptive algorithms vs. Time Warp: An analytical comparison”, To appear in the *1995 Winter Simulation Conference.*

[Stei91] Steinman, J.S., “Interactive SPEEDES”, *Proceedings of the 24th Annual Simulation Symposium*, 1991, 149-158.

[Stei93] Steinman, J.S., “Breathing Time Warp”, *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, May 1993, 109-118.

[Ston90] Stone, H., “High performance computer architecture”, Addison-Wesley, Reading, MA, 1990

[Theo84] Theofanos, M., “Distributed simulation of queueing networks”, Master’s Thesis, Department of Computer Science, University of Virginia, May 1984.

[TuXu92] Turner, S.J. and Xu, M.Q., “Performance evaluation of the Bounded Time Warp algorithm”, *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, January 1992, 117-126.