Techniques for Efficient Parallel Simulation and their Application to Large-scale Telecommunication Network Models

A Thesis
Presented to
The Academic Faculty

by

Kalyan S. Perumalla

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
December 2, 1999

Techniques for Efficient Parallel Simulation and their Application to Large-scale Telecommunication Network Models

Approved:

____

Richard M. Fujimoto, Chairman

____

Mostafa Ammar

____

Kenneth Mackenzie

____

Andrew Ogielski

____

Karsten Schwan

Date Approved ______________

Preface

In 1995, when I was searching for an interesting thesis topic in the second year of my Ph.D. program, Professor Richard Fujimoto was looking for people to participate in a special multi-institution project on telecommunication network modeling and simulation. Finding it as an interesting venture, I went aboard. Soon, another ambitious project on scalable simulation systems went into action, with which too I got involved. Both projects shared the vision of addressing the practical, down-to-earth challenges involved in taking parallel simulation technology into real-life network modeling and simulation problems. Issues arose, some minor, and others non-trivial, which demanded solutions immediately so that the projects stayed on course to meet their goals. My research focused on identifying the relevant issues, and addressing them as they arose. The research performed towards these two successful projects in the past four years became the context in which this thesis work evolved.

This thesis is organized into five chapters. Chapter 1 serves as an introduction to the parallel simulation problems addressed in this thesis. The features of telecommunication network simulation are also presented. Chapter 2 describes the reverse computation approach for rollback support in optimistic simulations. Chapter 3 presents the issues related to process-orientation support in optimistic simulations. Chapter 4 describes the challenges encountered, along with their solutions, in the design and development of a set of complex network protocol models. Finally, in Chapter 5, the key points of this thesis are summarized, and additional future work is identified.

Acknowledgments

I am greatly indebted to Dr. Richard Fujimoto, my *guru brahma*, for his support, guidance and trust all through my work with him. Learning from him, and working with him is simply a pleasure. I am also grateful for the opportunity he provided to me as research scientist to work with him on the simulation projects.

While I interned at Bellcore, Andy Ogielski provided warm support for my research ideas and approaches, and Sandeep Bhatt helped identify ATM PNNI protocols as a nice case study for large-scale network modeling effort, all of which contributed towards my thesis work. David Nicol provided the regular research challenges that served as springboards for looking harder for better solutions. Exciting discussions during many afternoons with Chris Carothers made research more enjoyable and interesting. Chris provided the spark that served as ignition for the reverse computation work. I thank my thesis committee members for their insightful comments and suggestions.

My parents, wife and sisters were my emotional partners throughout the ordeal of my doctoral study. This thesis owes its existence to their passionate support during all the ups and downs. No words can ever do justice in describing what their support means to me.

Contents

List of Tables

List of Figures

Summary

It is widely recognized that parallel simulation technology is necessary to address the new simulation requirements of important applications such as large-scale network models. However, current parallel simulation techniques possess limitations in the type and sizes of models that can be efficiently simulated in parallel. In particular, optimistic parallel simulation of fine-grained models have been plagued by large state saving overheads, both in event-oriented and process-oriented views, resulting in unsatisfactory parallel execution speed for many important applications. In the absence of alternative solutions, it was generally believed that optimistic approaches are inapplicable in such applications. This thesis asserts that, by use of appropriately constructed novel techniques, it is indeed possible to perform fast optimistic parallel simulation of fine-grained models, using both event-oriented and process-oriented views. In support of this claim, techniques are presented here that significantly lower the overheads, thereby enabling the capability to efficiently simulate large-scale, fine-grained models in parallel. On sample models, when compared to previously known approaches, the techniques presented here improve the simulation speed by a factor of 3 or more, while simultaneously reducing the memory requirements by almost half.

The first technique addresses the high overheads of state saving mechanisms that are traditionally used in supporting rollback operations in optimistic parallel simulation. An alternative approach called *reverse computation* is identified for realizing rollback, which is demonstrated to significantly improve the parallel simulation speed

while greatly reducing the memory utilization of the simulation.

The next technique concerns the process-oriented worldview, which is extremely useful in many domains, such as telecommunication network protocol modeling. An approach called *stack reconstruction* is developed to address the high execution overheads traditionally associated with process-oriented views, and its effectiveness is demonstrated in achieving a high rate of process-context switching during optimistic simulation.

Additional contributions of this thesis include the identification and solution of other typical problems encountered in the design, development and parallel simulation of models for real-life telecommunication network protocols. This work serves to demonstrate the readiness and feasibility of applying parallel simulation technology to today's large and complex models.

The parallel simulation techniques described in this thesis are applied and analyzed in the context of telecommunication network simulation, using representative network models. The techniques, however, are not restricted to network simulation, but are equally applicable to other domains as well. For example, the parallel simulation of any network of queues can benefit from the reverse computation system presented here. In fact, the reverse computation system is relevant to other application areas as well, such as in database recovery, and debugging environments. Similarly, the stack reconstruction approach is applicable to any multi-threaded system that requires an efficient incremental checkpointing facility for its thread states.

Chapter 1

INTRODUCTION

1.1 Background

1.1.1 Discrete Event Simulation

Simulation is the process of imitating a system. The imitation is performed at lesser or greater detail than the behavior of the original system. Simulation is useful for many reasons, such as for design, analysis, testing and understanding of complex systems. It provides a controlled environment for studying various aspects of the system under consideration. It has been rightly said[1] that "simulation is useful in confirming what is known, and exploring what is unknown." The ultimate goal of any simulation is to be able to answer precisely-stated questions about any given system. Simulation is a very commonly used tool in many application areas, such as air traffic control. Although simulation, in general, can be performed in many ways (*e.g.*, by prototyping), this thesis is only concerned with computer software-based simulation.

Computer software-based simulation is usually preceded by a modeling exercise, in which the elements of the original system, which is the subject of simulation, are first represented in some formal fashion. The system thus abstracted is then executed

[1] *anonymous*

in a controlled fashion on a computer. There exist different ways of modeling for simulation, such as continuous and discrete models. In this thesis, the focus is on discrete event models which are expressed in terms of discrete time. In discrete event simulation, the system behavior is modeled in terms of its actions at discrete points along a time scale. The evolution of the state of the system is fully expressed in terms of these actions. Each action, typically, is expressed as an instantaneous change in the system state, with causal dependencies among the actions. Thus, after effecting an instantaneous action, the model may dynamically determine a new set of future actions which are affected by this action. Each action is called an *event*. Since these events occur at discrete points in simulated time, this type of a simulation is called a *discrete event simulation*.

Sometimes, the time to execute a simulation model for a desired amount of simulated time can become large, especially if the modeled behavior is quite detailed and/or the modeled system configuration is huge. In such cases, it becomes desirable to find ways to reduce the execution time. There are many ways in which the reduction in simulation execution time is accomplished, each method having its own advantages and weaknesses. One of the more general methods is *parallel discrete event simulation*.

1.1.2 Parallel Discrete Event Simulation

Parallel discrete event simulation is concerned with the problem of utilizing multiple processors simultaneously in order to execute a single discrete event simulation model. Typically, it is desirable to put multiple processors to use in order to reduce the time to complete the execution of the simulation model, so that answers are obtained faster compared to sequential execution (*i.e.*, using a single computer). The model execution is partitioned across the different processors, and the processors proceed with the simulation in parallel. Since all the processors must share the same simulated timeline, which is common throughout the model, some sort of synchronization protocol must be practised by all the processors, so that the final answers from the parallel execution are exactly the same as those obtained from a correct (sequential) execution. Otherwise, if the processors are not synchronized correctly, it becomes difficult to interpret the final answers obtained from the parallel execution. Moreover, the final answers may vary from one execution run to another, because of the randomization effects of the underlying computation and communication platforms, making the answers even more unreliable.

An excellent introduction to parallel discrete event simulation can be found in [27]. Parallel simulation is a rich area for research, containing numerous interesting research topics, such as synchronization, repeatability, verification, validation, usability, performance tuning, synchronized data distribution, interoperability, programmability, automation and so on, to name just a few. The book [29] provides a splendid coverage of the most important issues, solutions and associated algorithms in parallel distributed simulation systems.

Parallel discrete event simulation protocols can be broadly categorized as optimistic or conservative, depending on whether (transient) incorrect computation is ever permitted to occur during the execution.

1.1.3 Conservative Parallel Simulation

In conservative style of parallel simulation, no computation is ever performed unless it is guaranteed that it is performed in the correct simulation time order. In case such a guarantee is not possible to make for any given event, then that event computation is blocked until such a time the guarantee can be made. In order to permit concurrency in conservative simulation, each event must be associated with a *lookahead* value which specifies a lowerbound on the timestamps of new events that this event generates into the future. Hence, lookahead specification is an important issue in conservative parallel simulation. In certain applications, it is sometimes possible to extract lookahead information automatically from the model code, while in most others, the user must participate in providing such information to the simulation runtime system. Other issues in conservative simulation protocols include potential deadlock discovery and recovery, since processors potentially block waiting for each other during runtime. Nevertheless, an advantage of using conservative simulation protocols is that the user can mostly assume the more familiar sequential execution semantics when developing and testing simulation models. One of the earliest conservatively synchronized protocols is described in [17], while more recent and sophisticated ones are reviewed in [28].

1.1.4 Optimistic Parallel Simulation

Optimistic parallel simulations, on the other hand, permit potentially incorrect computation to occur, but carefully undo or *roll back* such computation after realizing that it was in fact incorrect. This thesis is concerned primarily with optimistic style of parallel simulation, although some of the techniques are applicable to conservative parallel simulation as well as to sequential simulation.

Optimistic style of simulation has the property that it initially relaxes the dependencies among events, but discovers the individual dependencies dynamically at runtime (on an event-by-event basis). If and when any dependency violation is detected, corrective action is taken to rectify the violation. The corrective action involves local correction at the processor where violation is detected, as well as at other processors, if any, to which the violation propagated. This corrective action is performed via rollback mechanisms – state restoration mechanism for local correction, and message cancellation mechanism for remote correction.

The original optimistic synchronization approach, called Time Warp, was described in [35], and later, other variants and improved algorithms appeared (*e.g.*, [58, 61]). Some of the variants attempt in various ways to constrain the amount of optimism, since uncontrolled optimism can lead to a lot of incorrect computation that needs to be abandoned later. Others separate *risk* and *aggressiveness*, *i.e.*, they provide control on whether incorrect computation (called risk) is permitted to occur at a processor and/or allowed to propagate (called aggressiveness) to other processors.

Like with any parallel processing solution to complex problems, optimistic style of parallel simulation requires care in implementing it well. First, it is important to ensure that the runtime overheads due to parallel processing do not actually slow down the simulation. Otherwise, if the overheads are too high, then, parallel simulation may not deliver any speedup until the number of processors exceeds a breakeven value, which is clearly unappealing. Typically, rollback support operations constitute the biggest portion of the runtime overheads in optimistic simulation, which must be carefully minimized. Secondly, the amount of memory utilized for parallel simulation must be ideally as close as possible to that required for sequential execution. Large memory usage generally implies increased hardware cost and reduced runtime performance. Generally, the smaller the memory footprint of the simulation, the better is its performance and affordability. However, memory cannot be reduced to arbitrarily small levels because that would leave little memory for the processors to store the intermediate results generated by optimistic processing, thereby constraining optimistic execution, and consequently hindering parallel execution. Thus, the tradeoff between memory and computation speed must be settled carefully. Finally, the programming model must be sufficiently powerful to facilitate in the development of large and complex simulation models. The process-oriented view is generally more useful in this aspect, and hence it is desirable to blend efficient support for process-oriented view in conjunction with optimistic processing.

1.2 Problem Statement

For optimistic parallel simulations, three important factors are hindering their widespread use. First, researchers have long experienced difficulty achieving satisfactory execution performance for simulating important applications which are fine-grained in nature. This was mainly attributable to the high overhead of the state-saving techniques that are traditionally used. Secondly, optimistic process-oriented parallel simulations have long been viewed as necessarily slow. This was due to the lack of clear understanding of ways to avoid stack saving overheads required for optimistic process-context switching. Finally, successful feasibility demonstrations were not prevalent to help convince that optimistic techniques can indeed be applied to real-life, real-size problems with significant reduction in execution time. Each of these concerns is addressed in this thesis.

1.2.1 State Restoration

The *event computation* in simulation applications is one in which a set of operations modifies a set of memory items, called the *state*. Hence, in order to roll back the local effects of a computation, it is sufficient to restore the modified memory items to their values before the computation. Anti-messages are used to roll back any communication performed during the computation.

The most commonly used technique for realizing state restoration is *state-saving*. In this technique, the original value of the state is saved before it is modified by the event computation. Upon rollback, the state is restored by copying back the saved value. State-saving is a well-studied technique, with several variants such as incremental and periodic state-saving, and has been in practice for a long time.

Traditional state-saving operations, however, constitute significant overheads in fine-grained simulations (i.e., those simulations with very small amount of computation per event). Although many interesting applications, such as queueing network simulations, are fine-grained in nature, optimistic style of simulation could not be satisfactorily applied to such applications due to the high cost of traditional state-saving approaches. Fine-grained simulations actually experienced slowdown instead of speedup as a result of state saving overheads. Such bad performance features of traditional approaches led to the general belief that optimistic simulation was not a promising solution for fine-grained applications. Even in the case of medium to coarse grained applications, the generally large memory requirements of state saving serve as a deterrent to adopting optimistic simulation.

The generally discouraging aspects of traditional approaches point to the need for new alternatives to support state restoration in optimistic simulation. The problem is to find novel methods that entail significantly less overheads compared to previous approaches, such that optimistic simulation is not constrained by the cost of operations required for rollback support. Moreover, it is imperative that the memory usage by the simulation is minimized.

1.2.2 Process Orientation

An issue that is orthogonal to the style of simulation is that of the abstraction used in simulation models. Models expressed in terms of events and their associated (instantaneous) computations are said to follow the event-oriented view. Another popular abstraction is a process-oriented view in which multiple threads of computation interact with each other by exchanging events. The process-oriented view differs from the event-oriented view in that, for each process, a stack of contexts (*i.e.* control and data states) is maintained between event exchanges; no such facility exists in event-oriented views.

An advantage of using the process-oriented view is that models tend to be smaller, and easier to develop and understand, making it more appealing to the modelers. On the other hand, it is generally perceived that the event-oriented view can be more efficiently implemented than the process-oriented view, especially in the context of optimistic parallel simulations.

In recent simulation applications, the models tend to be too complex to express using the low-level primitives of the event-oriented view, thus making the process-oriented view a natural choice for developing such large and complex systems. Process-oriented views are, however, generally perceived to be expensive in the context of optimistic parallel simulations. This is mainly due to the fact that processes entail state-saving overheads for the maintenance of additional control flow information and transient data. These overheads can be quite large in naive implementations, unless they are carefully reduced to the minimum necessary. For example, the method of copying entire byte-stretch of a process stack before its optimistic suspension incurs too high a runtime and memory cost for fine-grained applications. In a typical fine-grained application, if the event computation takes only a few microseconds (*e.g.*, 5μsecs) and stack copy operation takes several microseconds (*e.g.*, 25μsecs), optimistic process suspension constitutes a 500% overhead! Clearly, this is unappealing. The problem is to find an efficient alternative in order to make optimistic simulation viable for models using the process-oriented view.

1.3 Contributions

This thesis presents contributions that address the preceding concerns in parallel simulation. A detailed account of the contributions is presented in section 5.1, while only the highlights are described here.

First, a technique called reverse computation is presented which is a significant improvement over the traditional technique of state saving. On sample models, reverse computation is observed to improve the speed of simulation by more than a factor of 3 while reducing the memory requirements by almost half. Secondly, a technique called stack reconstruction is presented which is effective in improving the execution performance in the parallel simulation of process-oriented models. Compared to previous stack copying methods, stack reconstruction is observed to reduce typical optimistic process context switching time in fine-grained models from dozens of microseconds down to less than 5 microseconds. Additionally, optimistic parallel simulation is applied to the design, development and parallel simulation of models of real-life telecommunication network protocols. This exercise serves to demonstrate the feasibility of applying parallel simulation technology to real-life models.

1.4 Telecommunication Network Simulation

The techniques presented in this thesis are applied and tested in the domain of telecommunication network simulation. Simulation is an important tool in the design and analysis of telecommunication networks. It is a practical tool for analyzing, explaining, anticipating or predicting the behavior of networks that are too complex for analytical treatment. Hence, network designers and researchers have been routinely resorting to simulation-based network design and analysis. Simulation is a practical methodology especially with the advent of very high-speed, large-scale networks that are too expensive to analyze by measurements or prototyping/emulation and too complex for comprehensive mathematical analysis.

However, the network simulation community is faced with significant challenges with respect to its modeling and simulation capabilities, due to the unprecedented complexity and scale of the networks of interest. The new generation of network simulation problems imposes significantly increased demands on the sizes of the networks to be simulated as well as on the lengths of the simulated time-scales. First, network traffic must be simulated for very long time-scales in order to accurately overcome long-lasting transient phenomena and to capture long-range correlation [67]. Next, truly large sized networks must be simulated in order to accurately study issues such as scalable routing, survivability, or packet loss correlation in multi-hop multi-domain networks such as in global ATM networks [9]. The need for such modeling and simulation capabilities is immediate in nature, in order to accommodate the need for planning, growth management and network management for future networks, such as the Next Generation Internet (NGI) [45].

It is widely recognized that current sequential simulation techniques do not provide sufficient capability to address these new simulation requirements. Whereas current design problems concern the simulation of large, multi-hop networks carrying millions of multi-protocol data flows, network simulations have been traditionally performed for relatively small network configurations and short time scales. This is primarily due to the performance limitations of the majority of simulators. Parallel simulation techniques can be expected to greatly help in improving the simulation capability for such systems. The widespread availability of multiprocessor workstations and compute-servers also aids in the possibility of making parallel network simulators more commonly accessible. For large network simulations, parallel execution can reduce execution times from days to a few hours.

There exist other ways to speed up telecommunication network simulations. One approach is to use coarser models of the network traffic. For example, in ATM (Asynchronous Transfer Mode) simulations, fluid-level or burst-level models can be used in place of cell-level models, which can reduce the simulation time by an order of magnitude or more[39, 30]. Although care must be taken in dealing with the resultant inaccuracy, it is an effective approach for reducing the simulation time. Similarly, network node aggregation techniques can be used in order to reduce a large network configuration into a more manageable but equivalent node model[36]. All these alternative approaches to speeding up simulations involve model-specific optimizations. Moreover, these approaches are orthogonal to the parallel simulation techniques, and hence could conceivably be used together with the techniques presented here.

The reverse computation approach is applied to representative network models, such as ATM multiplexer trees and Personal Communications Services (PCS) networks. The stack reconstruction approach is tested with the ATM PNNI (Private Network to Network Interface) protocol models. Stack reconstruction is not applicable in the ATM multiplexer and PCS models, since they are event-oriented in nature. Reverse computation could have also been applied to the ATM PNNI models. But since the PNNI models are coarse-grained in nature, it was not expected to deliver any appreciable performance improvement, and hence reverse computation was not used in conjunction with the PNNI models.

1.5 Guide to Document

The rest of the thesis is organized as follows. Chapter 2 describes the reverse computation approach for rollback support in optimistic simulations. Reverse computation is illustrated using a simple motivating example, followed by the methodology for automatically appylying the reverse computation approach to complex models using a special compiler, along with a discussion of important simulation-specific compiler optimizations. Performance results are presented that empirically evaluate the speed and memory improvements gained with reverse computation. The relationship of reverse computation for rollback with other similar work, such as fault tolerant systems and reversible computing, is also described.

In chapter 3, issues pertaining to process-orientation support in optimistic simulations are presented. The process-orientation worldview is described, together with details of different implementations, such as multi-threading and continuations-based approaches. The merits and drawbacks of each are identified, and the stack construction approach is identified as well-suited for fine-grained, process-oriented optimistic simulation. Performance results are presented that demonstrate the benefits of stack reconstruction.

Chapter 4 documents the challenges encountered, along with their solutions, in the design and development of a set of complex network protocol models. The implementation is discussed in the context of the TeD language, with the goals of model reuse and efficiency of parallel simulation. Finally, in Chapter 5, a detailed account of the contributions is listed. Additional future work and open issues are identified whose solutions can serve as finishing touches to the techniques presented in this thesis.

Chapter 2

REVERSE COMPUTATION

In optimistic parallel simulations, state-saving techniques have been traditionally used to realize rollback. In this chapter, an alternative approach called reverse computation *is proposed, and its execution performance is compared against that of state-saving. Using compiler techniques, reversible computations are automatically generated and optimized to transparently reap the performance benefits of reverse computation. For certain fine-grain models, such as queuing network models, it is shown that reverse computation can yield significant improvement in execution speed coupled with significant reduction in memory utilization, as compared to traditional state-saving. On sample models using reverse computation, an improvement by a factor of 3 or more is observed in execution speed over traditional state-saving.*

In optimistic parallel simulations, incorrect event computation is undone via rollback mechanisms. During event computation, only a specific set of memory items, collectively called the *state*, are modified. Hence, in order to roll back a computation, it is sufficient to restore the modified memory items to their values before the computation.

The most common technique for realizing rollback is *state-saving*. In this technique, the original value of the state is saved before it is modified by the event computation. Upon rollback, the state is restored by copying back the saved value. An alternative technique for realizing rollback is *reverse computation*. In this technique, rollback is realized by performing the inverses of the individual operations that are executed in the event computation. The system guarantees that the inverse operations recreate the application's state to the same value as before the computation.

In this thesis, it is demonstrated that the use of reverse computation for realizing rollback can lead to much more efficient executions compared to state-saving. Fine-grained applications (i.e., those with very small amounts of computation per event) are examples in which the performance improvement can be most pronounced. This is due to the fact that traditional state-saving operations constitute significant overheads in fine-grain simulations. Also, due to its reduced memory requirements, reverse computation results in a more efficient use of storage hierarchies. Reverse computation can greatly reduce the forward computation overheads by transferring most of the traditional overheads to the reverse computation path.

Here, it is demonstrated that the reverse computation approach incurs insignificant forward computation overheads and low state memory requirements in fine-grain models. The parallel simulation performance of reverse computation is observed to achieve better caching effects, with as much as two to six-fold improvement in several model configurations when compared to copy state-saving, periodic state-saving and incremental state-saving. Finally, it is demonstrated that this approach can be automated using compiler-based techniques that can automatically generate both a reversible version of the event computation code and its reverse, from a model's high-level description. Moreover, it is shown that the performance of automated reversal can approach that of manually optimized versions.

When reverse computation is used to simulate coarse-grain models, it is unclear if the improvement in execution speed can be as pronounced, because state-saving overheads are not so high in coarse-grain models. However, coarse-grain models still stand to benefit from reduction in state memory utilization when reverse computation is used.

In Section 2.1, the details of the reverse computation technique are presented using a simple illustrative application, followed in Section 2.2 by the automation techniques for applying to more complex applications. In Section 2.3, the performance of reverse computation and state-saving are compared. In order to place this work in context, in Section 2.4, work related to general reverse computing in theory and practice is identified. This work opens several interesting challenges and questions, which are identified in Section 2.5.

2.1 Overview

In this section, the reverse computation approach is illustrated with a simple example. For simplicity, the generalized treatment of more complex models is postponed until Section 2.2.

2.1.1 Motivating Example: ATM Multiplexor

Consider a simple model in Figure 1 of a non-preemptive ATM multiplexor, containing a buffer of size B. Suppose we are interested in measuring the cell loss probability, and the delay distributions on the queue [47].

The state of the system might be as shown in Figure 1 (a). The `qlen` variable is used to keep track of the current buffer occupancy; `sent` and `lost` are variables

```
int qlen;
int sent;
int lost;
int delays[B];
```

(a) state

```
if( qlen < B )
  {
  delays[qlen]++;
  qlen++;
  }
else
  {
  lost++;
  }
```

(b) cell arrival

```
if( qlen > 0 )
  {
  qlen--;
  sent++;
  }
```

(c) cell transfer

Figure 1: A Simple ATM Multiplexor Model

that accumulate statistics respectively of the total number of cells transferred to the output link and the total number of cells dropped because of a full buffer. The array `delays` measures the number of cells experiencing a given amount of delay, which in combination with the `sent` counter gives the cell delay distribution.

In order to model the behavior of the ATM multiplexor, two types of event handlers are used in the model. The cell arrival event handler processes newly arriving cells, as shown in Figure 1 (b). Upon a cell arrival, if the queue has no more room, then the counter `lost` is incremented representing that the cell has been dropped. Otherwise, the array element `delay[qlen]` is incremented representing that one more cell experienced a delay of `qlen` emission time units followed by an increment to `qlen` which represents that a cell has been added to the queue. The cell transfer event handler processes cell departure events, as shown in Figure 1 (c). Here, if the queue is not empty, then a cell is dequeued (i.e., `qlen` is decremented) and sent over the output link (i.e., `sent` is incremented).

Note that, for both event handlers, the code to schedule the cell arrival and cell departure events is not shown.

2.1.2 Approach

```
int qlen;
int sent;
int lost;
int delays[B];

bit b1;
bit b2;
```

(a) state

```
if( qlen < B )
  {
  b1 = 1;
  delays[qlen]++;
  qlen++;
  }
else
  {
  b1 = 0;
  lost++;
  }
```

(b) cell arrival

```
if( qlen > 0 )
  {
  b2 = 1;
  qlen--;
  sent++;
  }
else
  {
  b2 = 0;
  }
```

(c) cell transfer

Figure 2: Modified ATM Multiplexor Model

Now consider the model shown in Figure 2, which is obtained by slightly modifying the original model of Figure 1. The difference between the two models is that two additional *bit* variables have been added to the state of the original model, and these variables are used to note whether the `if` statements were executed or not. The two bit-variables correspond to the two `if` statements in the model, such that `b1` $= 1$ if `qlen` $<$ `B` and 0 otherwise. Likewise, `b2` $= 1$ if `qlen` > 0 and 0 otherwise[1].

If model is considered carefully, it can be seen that the *state* of the original model is fully captured by the bit variables `b1` and `b2`. In other words, the state-trajectory

[1]In fact, only one bit variable would be sufficient in this model, since the event handlers are mutually exclusive; but we shall use two variables for clarity in the discussion.

of the set S of the variables $\{qlen, sent, lost, delays\}$ has a one-to-one correspondence with that of the set $S' = \{b1, b2\}$. The point here is that the values of the variables in S can be easily recovered based only on the values of S'. To recover, the event computations can be run backwards, which will restore the variables of S to their before-computation values. More abstractly, the bit variables `b1` and `b2` are used to make the original model *reversible*. Indeed, it is easy to find the reverse code for each of the event handlers of the modified model, which is shown in Figure 3. For example, the reverse code shown in Figure 3 (a) performs a perfect undo of the operations of the cell arrival event handler given in Figure 1 (b). Thus, it is sufficient to maintain the history of the bits `b1` and `b2`, instead of the whole set of state variables S of the original model.

```
if( b1 == 1 )
  {
  --qlen;
  --delays[qlen];
  }
else
  {
  --lost;
  }
```

(a) Reverse cell arrival

```
if( b2 == 1 )
  {
  --sent;
  ++qlen;
  }
```

(b) Reverse cell transfer

Figure 3: Reverse Code for ATM Multiplexor Model

It is clear that the size of the state to be saved is dramatically reduced, from several hundreds of bytes (for S) to just 2 bits (for S'), which can be saved with negligible overhead in the forward computation. As an example, assuming one full word is needed to represent 2 bits on most machines, and if $B = 100$, then the state is reduced by a factor of $(100 + 3)/1 = 103$ when compared to copy state-saving. Even if incremental state-saving techniques are applied to this model, several bytes are needed for saving the changed data values, whereas two bits are sufficient for reverse computation.

2.1.3 Application Properties

Some important observations can be made to help us understand some of the properties of the model that allowed us to reduce the state so dramatically.

- **Property 1** The majority of the operations that modify the state variables are "constructive" in nature. That is, the undo operation for such operations requires no history. Only the most current values of the variables are required to undo the operation. For example, operators such as $++$, $--$, $+=$, $-=$, $*=$ and $/=$ belong to this category[2]. More complex operations such as *circular shift* (*swap* being a special case), and random number generation also belong here.

  In the multiplexor model, all the assignment operations are constructive. Hence, little extra information is needed to reverse those operations.

- **Property 2** The complexity of the code is such that the "control state" of the code occupies less memory than the "data state" of the variables.

[2]The $*=$ and $/=$ operators require special treatment in the case of multiply or divide by zero, and overflow/underflow conditions.

In the multiplexor model, only two bits were necessary to record the control flow information. In contrast, the data state that is modified is much larger.

If property 1 is not satisfied in the model because of the presence of non-constructive operations such as plain assignment or modulo computation, the reverse computation method can in fact degenerate to the conventional state-saving operations. Such non-constructive operations are called *destructive assignments*. A straightforward method to reverse a destructive assignment is to save the old contents of the left-hand-side as a record of the "control information" for that assignment statement, which makes it degenerate to state-saving. However, later in the discussion, optimizations shall be presented that are sometimes possible to prevent the degeneration of destructive assignments to traditional state-saving.

If property 2 is not satisfied because the code is "too complex" (i.e., the amount of control state is more than the data state), traditional state-saving techniques can be utilized. On the other hand, property 2 suggests that this mechanism is well-suited for simulation models in which the event computations are small.

Queuing network models are an excellent example of the domain of models in which the preceding two properties are satisfied to a large extent. Consequently, it is clear that reverse computation is well suited for the optimistic simulation of queuing network models.

2.2 Automation Techniques

In the case of the multiplexor example, the code is small enough to come up with its reverse code by inspection. Now the more general case will be considered in which the code is complex, requiring a methodical, automated solution for generating the reverse code and for reducing the state size.

Reversibility is a useful property whose applicability extends beyond optimistic parallel simulation to several other domains, such as debugging systems and database recovery. In all these applications, the need arises to undo erroneous or unwanted changes to the state of the program. Reversibility of a program requires that information is never destroyed during program execution. However, programs written in popular general-purpose languages, such as **C** and Pascal, tend to be irreversible in nature. Due to the very nature of the language constructs, some information is destroyed as the program executes, making the program difficult to execute in reverse. Nevertheless, any program can be made reversible by carefully augmenting it in such a way that the information being destroyed by the program is actually saved on the side using additional non-intrusive instructions. The augmented program thus not only retains the semantics of the original program, but also becomes reversible. Automated means for reversibility are useful and important since the application code can be too complex for manual transcription and optimization by human inspection. Techniques are described next for automatically rendering a given block of code reversible, and important optimizations are presented that serve to enhance the efficiency of the reversed code.

2.2.1 Compilation vs. Interpretation

In general, there are multiple ways in which a given block of computation can be made reversible. One way to categorize them is based on whether the reversing execution is realized using interpreted or compiled means.

Compiled Approach

In the compiled method for reversal, a given block of source code is statically transformed via compilation into a forward computation block and a corresponding reverse computation block. In such source-to-source translation, the forward computation block preserves the semantics of the input source code in addition to making it reversible. The reverse block then corresponds to its inverse. Both the forward and reverse blocks are then compiled just as normal code. This approach is illustrated in figure 4 for part of the multiplexer model.

<table>
<tr>
<td><pre>if(qlen < B)
  {
  delays[qlen]++;
  qlen++;
  }
else
  {
  lost++;
  }</pre></td>
<td><pre>if(qlen < B)
  {
  b = 1;
  delays[qlen]++;
  qlen++;
  }
else
  {
  b = 0;
  lost++;
  }</pre></td>
<td><pre>if(b == 1)
  {
  --qlen;
  --delays[qlen];
  }
else
  {
  --lost;
  }</pre></td>
</tr>
<tr>
<td>Original</td>
<td>Forward</td>
<td>Reverse</td>
</tr>
</table>

Figure 4: Example Illustrating Compiled Approach to Reverse Computation

Interpreted Approach

In the interpreted method for reversal, the forward computation code is augmented with special instrumentation code. During the actual forward execution, the specially added code generates a sequential log of all the individual operations that are executed. For reversal, the generated log is then interpreted in the reverse order, and appropriate inverse operations are invoked. This approach is illustrated in figure 5 for part of the multiplexer model. Typically, the operation codes along with actual operand values are saved during forward computation, which are later interpreted during reverse execution, and their corresponding inverse operations are applied to the operands.

`if( qlen < B )`
`  {`
`  delays[qlen]++;`
`  qlen++;`
`  }`
`else`
`  {`
`  lost++;`
`  }`
	`if( qlen < B )`
	`  {`
	`  LOG(INCR,`
	`      &delays[qlen]);`
	`  delays[qlen]++;`
	`  LOG(INCR,&qlen);`
	`  qlen++;`
	`  }`
	`else`
	`  {`
	`  LOG(INCR,&lost);`
	`  lost++;`
	`  }`
		`for all log entries`
		`in reverse order:`
		`switch( operation )`
		`  {`
		`  case INCR:`
		`    {`
		`    (*p)--;`
		`    }`
		`    ...`
		`  }`

Original	Forward	Reverse

Figure 5: Example Illustrating Interpreted Approach to Reverse Computation

In general, the compiled approach is better suited in the context of high-performance parallel simulations, and hence the compiled approach is adopted here. For such high-performance applications of reversibility, the undo capability is required to be of low overhead with regard to memory and time. For example, in optimistic parallel simulations that execute hundreds of thousands (or even millions) of events per second, the rollback overhead per event must be kept very low. The low overhead requirements of high-performance applications precludes the use of interpretation-based approaches to reversibility. In such applications, the benefits of compiled reverse code are highly desirable. On the other hand, in other applications such as interactive debugging, in which flexibility and control are important, interpretation-based approach to reverse execution are applicable, as described in [11]. The interpreted approach is not discussed further in this thesis. The source-to-source translation system described here uses the compilation approach and avoids interpretation, to keep the runtime overheads low.

The details of the compiled approach are described next. First, the compilation techniques are described that are suitable in general for any application that requires support for reversible computations. Following that, the application of the compilation techniques to the parallel simulation problem is described.

2.2.2 Compilation

The compilation for reverse computation is performed in three phases — normalization, transformation and optimization — as illustrated in figure 6. The compiler

Figure 6: Reverse Compilation Phases

normalization and transformation rules are described in greater detail in [53] in the context of the C language. Only excerpts from [53] are provided here for a high level description.

The source code of the original model is fed through the special reverse compiler. From the input model, the compiler generates two separate outputs. The first output is an instrumented version of the input model, which contains the necessary code to make the input code reversible (e.g., the code in Figure 2). The second output is the reversing code that serves to undo the effects of the input model (e.g., the code in Figure 3). In the actual simulation, the instrumented code is used in place of the original code. The reversing code is invoked to roll back an event. The goal of the compiler is to generate the most efficient versions of both the instrumented code and the reverse code such that the state size is minimized while simultaneously reducing the runtime execution overhead.

2.2.2.1 Normalization

For convenience, it is useful to first translate the input program into another program that is semantically equivalent but which uses a reduced set of language constructs. Since most languages provide multiple redundant variants of the same primitive as convenience constructs, mapping multiple variants to a single form simplifies the development effort. For example, multiple variants of jump instructions can be mapped to a single type of jump instruction (*e.g.* `goto`). Similarly, initializer expressions can be moved out of declarations and included as explicit assignments.

This phase, called the *normalization* process, is performed as the first step in the compilation process. The advantages of normalization are that it is simpler to make the normalized code reversible, and then, generating the reverse code becomes easier.

2.2.2.2 Transformation

In the transformation phase, the normalized code is made reversible, and in addition, its reverse is also generated. The transformation starts with each function definition in the input, and every statement in the function is recursively transformed. This constitutes the main functionality of the compiler.

2.2.2.3 State Size Upperbounds

A simple set of translation rules that can be used by the compiler is shown in Table 1. The most common types of statements used in high-level languages are listed, along with their corresponding instrumented and reverse code outputs. Against each of the statements, the state size that is achievable for that statement type is listed. Since not all operations of the input model are perfectly reversible, it is necessary to add control state information to be able to reverse them. However, as shall be seen in Section 2.2.8, the better the understanding of the semantics of the code, the better the ability to reduce the state size. Hence, the reduction in state size can vary depending on the sophistication of the compiler. The translation rules of Table 1 thus place an *upper bound* on the state size, which could potentially be improved via optimizations. A reverse **C** compiler called `rcc` has been implemented that realizes these upper bounds for the **C** language[53].

The instrumented forward computation code, as well as reverse code, are generated by recursively applying the rules of Table 1 to the input model. The significant parts of these rules are their state bit size requirements, and the reuse of the state bits for mutually exclusive code segments. Each of the rules is described in detail next.

- **T0**: The `if` statement can be reversed by keeping note of which branch is executed in the forward computation. This is done using a single bit variable `b`, which is set to 1 or 0 depending on whether the predicate evaluated to true or false in the forward computation. The reverse code can then use the value of `b` to decide whether to reverse the `if` part or the `else` part when trying to reverse the `if` statement.

Since the bodies of the `if` part and the `else` part are executed mutually exclusively, the state bits used for one part can also be used for the other part. Hence, the state bit size required for the `if` statement is one plus the larger of the state bit sizes, x_1, of the `if` part and x_2 of the `else` part, i.e., $1 + max(x_1, x_2)$.

- **T1**: Similar to the simple `if` statement (**T0**), an `n`-way `if` statement can be handled using a variable `b` of size $lg(n)$ bits. Thus, the state size of the entire `if` statement is $lg(n)$ for `b`, plus the largest of the state bit sizes, $x_1 \ldots x_n$, of the component bodies, i.e., $lg(n) + max(x_1 \ldots x_n)$ (since the component bodies are mutually exclusive).

- **T2**: Consider an n iteration loop, such as a `for` statement, whose body requires x state bits for reversibility. Then n instances of the x bits can be used to keep track of the n instances of invocations of the body, giving a total of $n * x$ bit requirement for the loop statement. The inverse of the body is invoked n times in order to reverse the loop statement.

- **T3**: A loop with variable number of iterations, such as a `while` statement, can be treated the same as a fixed iteration loop, but the actual number of iterations executed can be noted at runtime in a variable `b`. The state bits for the body can be allocated based on an upper limit n on the number of iterations. Thus, the total state size added for this statement is $lg(n) + n * x$. If an upper limit on the number of iterations is unknown, alternative approaches can be used, such as described in [53].

- **T4**: For a function call, no instrumentation is added. For reversing it, its inverse is invoked. The inverse is easily generated using the rules for **T7** described later. The state bit size, x, is the same as for **T7**.

  In the simple case in which the function call graph is a tree, the state bit sizes can be completely determined *statically*. In the case of models in which the function call graph is a directed acyclic graph (DAG), the (maximum) state bit size requirements can still be statically determined. However, in the more general case of an arbitrary function call graph (implying the presence of direct and/or indirect recursion), it is difficult to statically determine the maximum state bit sizes.

- **T5**: Constructive assignments, such as `++`, `--`, `+=` and so on, do not need any instrumentation. The reverse code uses the inverse operator, such as `--`, `++`, `-=` respectively. These constructive statements do not require any state for reversibility.

- **T6**: Each destructive assignment, such as `=`, `%=` and so on, can be instrumented to save a copy of its left hand side into a variable `b` before the assignment takes place. The size of `b` is $8k$ bits for assignment to a k-byte left hand side variable (*lvalue*). This is similar to Steinman's incremental state-saving technique [63].

- **T7**: In a sequence of statements, each statement is instrumented depending on its type, using the previous rules. For the reverse code, the sequence is reversed, and each statement is replaced by its inverse, again using the corresponding generation rules from the preceding list. The state bit size for the entire sequence is the sum of the bit sizes of each statement in the sequence.

- **T8**: Determining the state size requirements in the presence of arbitrary jump instructions is difficult. In simple cases such as one in which no `goto` label in the model is reached more than once during an event computation, the following analysis can be used. Since at most one index per `goto` label is stored, the bit size requirement of this scheme is $lg(n+1)$ where n is the number of `goto` statements that are the sources of that single target label. Note that even if a label is the target of only one jump instruction, at least one bit is required, to distinguish between reaching the label normally (falling-through) and reaching the label as a result of the jump instruction.

  The more general case of models containing arbitrarily complex use of jump instructions is treated in [53].

- **T9**: Any legal nesting of the previous types of statements can be treated by recursively applying the corresponding generation rules. The state bit size is also obtained by the corresponding state-bit composition rule.

2.2.2.4 State Size Determination

To determine the amount of state needed to reverse a given event computation, the following procedure is used. Since the model code is a sequence of statements, start with T7 (or, alternatively, T4), and recursively apply the rules of Table 1. This is done while reusing the bits on code segments that are mutually exclusive (as indicated by the MAX() operation in the table). The analogy of *register allocation* is applicable here. The state can be seen as a sequence of bits, which correspond to registers of a computer. The bits are allocated to the state that is required to record control-flow information. Just like registers, these bits can be allocated in an intelligent manner so that mutually exclusive statements can reuse the same bits. For registers in general computing, the savings are in computation time; for control bits in optimistic simulations, the savings are in state copying operations and in state size reduction.

It is easily observed that the statements with potentially higher state bit sizes are destructive assignments, nestings of conditional statements within loops, nested loops inside loops, and destructive operations among inter-dependent jump instructions. In fine-grain models, it is unlikely that complex code involving nested or complex loops will arise. Hence, the higher state requirements of such complex code is not a serious problem for these models. However, destructive assignments are not uncommon. The most common occurrence of destructive assignments is in random number generation, which is addressed in section 2.2.7.

2.2.3 Optimization

Several optimizations are possible after the input code is subjected to the normalization and transformation phases of compilation. The more the sophistication of the compiler with respect to the optimizations, the better is the efficiency of the forward and reverse code. Some of the optimizations are described next.

Irreversible Slices

In general, certain operations performed in the input code must be ignored for reversal purposes. There are two reasons for this:

1. Irreversible operations: In any program, certain operations can be assumed to be irreversible for practical purposes. For example, output statements may not be reversed. A message-send over the network cannot always be physically withdrawn.

2. Environment: In certain applications some operations are indeed reversible, but the application has other (probably more efficient) means of reversing those operations. For example, in optimistic parallel simulation, the simulation kernel provides efficient mechanisms for retracting those events that are scheduled by an event that is eventually rolled back. Such operations must be ignored during reverse compilation.

Without loss of generality, both the preceding types of operations can be deemed to be irreversible for the purposes of reverse compilation.

The portions of the program that only lead to such irreversible operations can be *sliced away* during reversal. In other words, for those portions of the forward computation that lead solely to irreversible computation:

1. no attempt is made to make them reversible during forward computation, and

2. such code is ignored during reverse computation.

Applying this optimization can result in reduced overheads for reversibility.

Example: Consider the following code fragment in which the value of a variable `y` is computed and printed.

Original	Optimized Transform	Optimized Reverse
`{`	`{`	`{`
`int x;`	`int x;`	`int x;`
`s1`	`s1`	`RESTORE(x);`
`{`	`{`	`rs4`
`int y;`	`int y;`	`{`
`s2`	`s2`	`rs3`
`y = f(x);`	`y = f(x);`	`rs2`
`print(y);`	`print(y);`	`}`
`s3`	`s3`	`rs1`
`}`	`}`	`}`
`s4`	`s4`	
`}`	`SAVE(x);`	
	`}`	

The value of `y` is computed as a function of `x`, and printed as output. The slice of the program that only leads to the irreversible printing consists of the variable `y` and the statements `y=f(x);` and `print(y);`. All the components of this slice can be safely eliminated in the reverse code.

Eliminating Reversal of Initialization

Even though initializations are destructive assignments, they never need to be undone, and hence don't need to be state saved. This is because, initially, the variables contain *don't care* values, whose exact values are immaterial for the correctness of the program. For reversal, the same variables can hold any values, which are not necessarily the same *don't care* values that were held before initialization, since any value trivially qualifies as a *don't care* value. For example, consider the following code fragment, and its unoptimized and optimized reverse codes.

Original	Unoptimized Transform	Unoptimized Reverse	Optimized Transform	Optimized Reverse

`{`
`  int x;`
`  x = 0;`
`  s`
`}`
	`{`
	`  int x;`
	`  SAVE(x);`
	`  x = 0;`
	`  s`
	`  SAVE(x);`
	`}`
		`{`
		`  int x;`
		`  RESTORE(x);`
		`  rs`
		`  RESTORE(x);`
		`}`
			`{`
			`  int x;`
			`  x = 0;`
			`  s`
			`  SAVE(x);`
			`}`
				`{`
				`  int x;`
				`  RESTORE(x);`
				`  rs`
				`}`

The unoptimized version saves the pre-initialization value of `x` before initializing it to zero, and restores `x` to the saved value during reverse execution towards the last statement. Since there is no other statement in the reverse code that depends on the pre-initialization value of `x`, such restoration is useless. Hence, both the saving and restoration codes for pre-initialization values are eliminated in the optimized version.

This optimization can result in significant memory savings, since local variables are very commonly declared and initialized inside functions.

Note that this optimization is not restricted to local (stack) variables, but also applicable to variables allocated on the heap. The same applies to memory buffers whenever they are re-initialized for irreversible reuse. In fact, a simple generalization of this optimization is the following:

> For memory items that contain *don't care* values, (destructive) assignment of values to them need not be reversed.

Invariant Expressions

Determining if an expression is invariant in a block of code is useful in performing certain optimizations to reduce overheads. For example, in some cases, it might not be necessary to remember the result of evaluating an `if` condition. This is true if the condition expression remains invariant in both branches of the `if` statement (i.e. the condition expression is guaranteed to compute exactly the same value both in forward and in reverse). This fact can be determined by checking the bodies of the true and false branches of the `if` statement to see if any of the variables used in the condition are indeed modified or potentially modifiable in the branches. If none of the variables contained in the condition is affected in the bodies, then the result of the condition need *not* be saved, but instead the exact same condition expression can be re-evaluated to determine the condition value both in forward and reverse codes. This optimization reduces memory requirements, but at a runtime cost of reevaluating the condition expression. In case the condition expression is not invariant in the body of the `if` statement, the result of the condition needs to be saved during the forward execution. However, for this purpose, only one bit is needed to remember the truth value of the condition.

Similarly, for a `switch` statement, an additional variable can be introduced to remember the value of the `switch` expression, lest it is changed in the body of the `switch`. This additional variable can be eliminated if the `switch` expression can be determined to always be invariant in the `switch` body.

A simple *sufficient* condition for the invariance of any expression is when all the variables of the expression are explicitly declared by the programmer as invariant (for example, using the `const` qualifier in C/**C**$^{++}$). This trivially ensures that the expression is always invariant.

Common Sub-expression Elimination

Even though reverse compilation can increase the number of sub-expressions and variables in order to render all expressions side effect free, the potential increase of sub-expressions is mitigated by the fact that common sub-expression elimination will later be performed by the usual **C** compiler when the generated code is compiled.

State Bit Compression

Run-length encoding and decoding can be performed on the bits that are saved for reversibility. This can result in significant memory savings, for example, in case of loops involving conditional statements. Consider the following code pattern that commonly occurs in programs.

```
done = 0;
while( !done )
{
  if( expr )
  {
    done = 1;
  }
  else
  {
    ...
  }
}
```

In the preceding code fragment, the loop is executed until a condition `expr` becomes true. If the condition expression is modified in the `if` branches, then its truth value (1 bit) must be saved for each iteration. Suppose the loop is executed n times. It is clear that all the first $n-1$ bits will hold a 0 value, and only the last bit will contain a value 1. If run length encoding is used on the saved bits, then the first $n-1$ bits can be reduced to $O(\lg n)$ bits.

2.2.4 Fault Detection

A useful side benefit of reverse execution is that it can be used in trapping errors in either forward code or the reverse code, or even in the implementation of the reverse compiler. This is achieved by checking the following simple *necessary* correctness condition at runtime[10]:

Suppose a variable `var` is initialized to `expression` in the forward execution. At the end of the reverse execution, the final value of the variable `var` must equal `expression`.

The preceding simple condition, when applied to every local and global variable, is useful in detecting errors in the code or the compiler. Another simple correctness condition is the following, which is a generalization of the end-of-tape condition described in [10]:

Suppose the bit/byte tape is at position P before the forward execution of a function f, and later the reverse of the function is executed. Then, by the end of the reverse execution, the tape must be rewound to the same position P.

2.2.5 RCC: Reverse C Compiler

The reverse compilation techniques outlined in previous sections, along with those described [53], have been implemented in a reverse **C** compiler called `rcc`. The compiler takes a list of file names containing ANSI **C** code as input. For each of the input files, the compiler generates the reversible forward code as well as the reverse code. Specifically, for each function `f()` in the input, two functions are generated in the output. The first retains the signature of the input function, and is made reversible. The second function is the inverse of the input function, with the same parameter list, but returning `void`.

Although the compiler is mainly developed for use in parallel simulation, the compiler can be easily retargeted to any other application. A simple `pragma` interface to the compiler allows the user to specify optimization hints to perform various optimizations. The compiler retains the desirable feature of requiring no modifications to the input **C** source of the model (except for the insertion of `pragma` directives), in order to make the model reversible. In fact, the `pragma` directives serve as a performance enhancing feature; the correctness of the reversible and reverse code is not influenced by the `pragma` hints. Although currently the performance optimization is mostly driven by `pragma`s, the compiler can be enhanced with more sophisticated dependency analysis to automate the optimization process.

2.2.6 Application to Parallel Simulation

The `rcc` reverse **C** compiler was used for automatically generating reverse computation corresponding to event computations in the optimistic parallel execution of simulation applications. In order to use `rcc` in this setting, the event processing functions were provided as input to `rcc`, and their augmented-forward and reverse functions were obtained as output, which were then used in the simulation. Each event processing function `f()` receives exactly one event as input, and modifies a set of variables called state, and can generate additional events during the event processing. In case the event needs to be rolled back, the reverse function of `f()` gets invoked, in order to undo the changes performed by `f()` on the state. The events that were scheduled by that function will be canceled by the GTW kernel. Hence no additional action is needed for reversing the generation of those events.

2.2.6.1 Event Model Optimizations

The generation of new events as part of an event is a very common and fundamental operation in discrete event models. Two important optimizations can be performed with respect to event generation:

1. **Event scheduling slice**: In optimistic simulation, those events that were generated by an event that is eventually rolled back are retracted by the simulator kernel automatically, using efficient data structures and other simulation techniques[27]. Hence the portion of the event processing code that solely contributes towards the generation of new events is a message-send slice (see section 2.2.3), which can be sliced away in the reverse code. For example, when scheduling new events, a future time-stamp must be specified at which the new event will be scheduled. The time-stamp may be generated using, say, an arithmetic expression involving a random number generator. The time-stamp may in fact be evaluated and stored in a temporary local variable, and then that variable may be used while scheduling the new event. This time-stamp variable, and its computation contribute solely towards the event scheduling, and hence belong solely to the event-sending slice. Thus, the time-stamp variable and its computation can be safely removed from the reverse code.

2. **Event data initialization slice**: Similar to the event scheduling slice, the event initialization slice involves the initialization of the new event data. All the code that initializes the event data can be safely sliced away in the reverse code, since such code need not be reversed.

The elimination of event initialization and scheduling slices together with other optimizations can result in significant reduction in the reverse code overheads.

Example

Consider the following code fragment extracted from an actual simulation model of a PCS network[14], highly simplified here for illustration.

Original Model

```
while( f(x) )
{
  x += g();

  if( h(x) )
  {
    /*schedule
     an event*/
    ...
  }
}
```

In this model, the variable `x` is updated in a loop until a certain value of `x` is reached. As part of the loop, for each value of `x` that matches a certain condition, a new event is scheduled.

A straightforward transformation of this model can result in unoptimized versions of forward and reverse code, as illustrated in the following.

Unoptimized Forward	Unoptimized Reverse

`int n=0;`
`while( f(x) )`
`{`
`  char c;`
 	
`  n++;`
 	
`  x += g();`
 	
`  if( c = !!h(x) )`
`  {`
`    /*same as`
`      original*/`
`    ...`
`  }`
`  SAVE(c);`
`}`
`SAVE(n);`
	`int n;`
	`RESTORE(n);`
	`for( ; --n >= 0; )`
	`{`
	`  char c;`
	`  RESTORE(c);`
	`  if( c )`
	`  {`
	`    /*undo event`
	`      scheduling*/`
	`    ...`
	`  }`
	`  x -= rg();`
	`}`

However, if the optimization for event-scheduling slice is applied in this model, the body of the `if` statement becomes empty in the reverse code, making it a null statement. This is because event scheduling code can be ignored for reversal. This removes the need to save the condition value in the forward code, thus removing the reversibility overheads of the `if` statement altogether. The optimized version is as follows.

Optimized Forward	Optimized Reverse
`int n=0;`	`int n;`
`while( f(x) )`	`RESTORE(n);`
`{`	`for( ; --n >= 0; )`
`  n++;`	`{`
	`  x -= rg();`
`  x += g();`	`}`
`  if( h(x) )`	
`  {`	
`    /*same as`	
`      original*/`	
`    ...`	
`  }`	
`}`	
`SAVE(n);`	

Reduced Overheads

The unoptimized reverse code requires remembering the result of evaluating the `if` condition. If the loop executes n iterations, then n bits are required for reversibility. In addition, the destructive assignments in the event generation code must be state saved, requiring additional bits. However, in the optimized version, the event scheduling code is eliminated in the reverse, based on the optimization of event scheduling and initialization slicing. This implies that the result of the `if` condition is *never* saved or restored. Hence, the optimized version requires at least n less bits than the unoptimized version.

The scheduling of the event requires event allocation, event data initialization and event scheduling. If these are not eliminated from the reverse, they will need to be state saved in the forward code (which uses several destructive assignments to accomplish the allocation, initialization and scheduling). All such overheads are eliminated in the optimized version.

2.2.7 Reversible Random Number Generation

Random number generation is central to all simulation models. Several random number streams may be used in the same simulation, to model various phenomena. A random number stream is generated by repeatedly invoking a specified function on a *seed* variable. The function modifies the value of the seed every time the function is invoked. Thus, a seed variable is needed for every random number stream used in the simulation model. The size of the seed variable varies with the type and quality of the random number generator (RNG).

In optimistic simulations, if an event computation invokes an RNG, and eventually the event is rolled back, it is necessary to roll back the random number generation. Otherwise, the simulation results can be unpredictable and unrepeatable. In order to be able to roll back the random number generation, traditionally, the seed value is state-saved. Incremental state-saving techniques are used in case the model contains many seeds.

However, if the reverse computation approach is used in order to avoid state-saving, the following problem is quickly encountered — RNGs rely on lossy/destructive assignments such as modulo operations. This implies that a straightforward application of reverse computation techniques can degenerate to incremental state-saving. To get around this problem, those RNGs are needed which do not rely on state-saving to reverse. On an abstract level, RNGs can be reasonably expected to be *reversible* without the need for state-saving, since, after all, random number streams are nothing but statically laid out cyclic sequences of numbers. It should be possible to traverse forward and backward along the cycles with the same ease. Fortunately, certain important class of RNGs do possess forward and reverse generation functions[41], which can be easily applied for reverse computation!

Significance of Reversible RNG

Due to new statistical requirements arising out of the large size and inherent complexity, recent simulation applications require RNGs with stronger statistical properties and longer periods, which together imply an increase in the seed size of the RNG. For example, in [43], the "Mersenne Twister" (MT19937) RNG is presented. This RNG has an extremely long period of $2^{19937} - 1$, and is of high statistical quality. However, it requires 624 words of space for seeds. Whenever random number generation is performed in the simulation (which could potentially happen as frequently as *every* event), the seeds need to be saved. For a traditional Time Warp system using this generator, 2496 bytes of state would need to be saved per event just to support the undo operation for the RNG. One might think that incremental state-saving could be employed here, but the way this RNG is structured, some bits from each word are subject to change every time a random number is generated, thus making it difficult to optimize using incremental state-saving techniques. Assuming the reverse recurrence can be found for MT19937, which its creators believe is possible, no state saving is required for the seeds, which reduces the memory requirements of such applications considerably. Because of the reduction in state-saving overheads, system performance will improve as well.

2.2.8 Reverse Code Efficiency

The reversibility of random number generators, even though they contain destructive assignments, leads to the following third property of the models that can help prevent reverse computation from degenerating to state-saving:

- **Property 3:** The non-reversibility of the individual steps that compose a computation do not necessarily imply that the computation, when taken as a whole, is not reversible.

Property 3 suggests that even if the individual steps of a computation are not efficiently reversible (i.e., either property 1 or 2 is violated), then one should look to a higher-level to see if the computation is not reversible from that level. An interesting question to consider in the future is the definition of an automatic mechanism for identifying code sequences which are individually not reversible, but for which a reversible code sequence can be determined when considered in a larger context.

This observation holds for several other common operations that contain destructive assignments. For example, a shift operation on an array of n elements can require n state-saving operations using incremental state-saving techniques. The same operation requires saving only one element using reverse computation. In fact, a circular shift requires no state when reverse computation is used, whereas incremental state-saving can require n state-saving operations (the commonly used swap operation is only a special case of circular shift). Similarly, insertion or deletion operations (which contain destructive assignments such as pointer assignments) into tree data structures (e.g., priority queues) can require several state-saving operations using incremental state-saving, whereas, no state is needed when reverse computation is used. This is because those operations naturally possess perfect inverses (e.g., *delete* and *insert* are inverses of each other).

An important outcome of this work is the recognition that reverse computation is well-suited for queuing network models. Many of the operations in queuing network models are either constructive operations (increment, decrement, etc.), or reversible groups of destructive assignments (random number generators, queue operations, etc.). Also, the event computations in these models tend to be of fine-granularity. This implies that reverse computation is an excellent approach for optimistic parallel simulation of queuing network models.

2.3 Performance Evaluation

In order to study the performance of reverse computation relative to state-saving, two flavors of reverse computation were compared with three variants of state-saving. All the variants have been implemented using the Georgia Tech Time Warp (GTW) optimistic parallel simulator for shared memory multiprocessors[20]:

- **GTW-RC**: GTW with reverse computation, in which the reverse code for the application models is generated manually and optimized by inspection

- **GTW-RCC**: GTW with reverse computation, in which the reverse code for the application models is automatically generated using a special reverse compiler
- **GTW-CSS**: GTW with copy state-saving, in which a copy of the entire state is made before every event
- **GTW-PSS**: GTW with periodic state-saving, in which a copy of the entire state is made every p^{th} event
- **GTW-ISS**: GTW with incremental state-saving, in which a copy of only the modified portions of the state is made during every event.

In addition, the following two sequential versions of GTW were used for comparison purposes:

- **GTW-SEQ**: Optimized sequential simulator with GTW interface
- **GTW-NONE**: Parallel version of GTW, with rollback support turned off (i.e., with neither state-saving nor reverse computation), so that it can be run sequentially, but not in parallel.

First, the details of these variants are presented, followed by the details of the study that compares their performance characteristics. All the experiments were performed on a 16 processor, SGI Origin2000, shared-memory multiprocessor, with 8 MB of level-two cache per processor, and 4 GB of total memory. In all cases, the total number of events committed was deterministic and consistent with sequential runs, and the performance was found to be repeatable.

2.3.1 Implementation

The reverse computation technique has been implemented in GTW, which is originally based on state-saving to realize rollback. To use reverse computation for rollback, three significant modifications were made to the GTW kernel.

First, the GTW application programmer interface was extended to include the addition of a reverse event handler for every LP, corresponding to the LP's event handler. The arguments to the reverse event handler include the current state of the LP, and a list of events sent during the forward computation. Whenever a processed event is rolled back for an LP, GTW invokes the corresponding LP's reverse event handler.

Next, GTW's core rollback mechanism required some significant changes as well. GTW uses a technique called *direct cancellation* [26] to support the "de-scheduling" of previously scheduled events by an event that was rolled back. This technique allows one to keep a direct pointer to the event that needs to be canceled. Because of this, an optimized rollback mechanism can be supported that doesn't require one to search though the processed event-list of an LP. Instead, if the event that is to be canceled has been processed, the rollback mechanism simply restores the version of LP state that was made prior to processing this event. The other processed events that come after the canceled event are marked as unprocessed and placed back into pending event-list. For supporting reversing computations caused by secondary rollbacks (i.e., rollbacks caused by event cancellations), this optimized technique is unsuitable. To "undo" a sequence of event computations using reverse computation requires that each event be "unprocessed" in the precise reverse order in which it was processed.

Consequently, the direct cancellation mechanism was modified so that it starts with the last event processed by the LP and moves through the LP's processed event-list in reverse time stamp order, invoking the reverse event handler for each event to undo its changes to state. The changes to the primary rollback mechanism (i.e., rollbacks caused by straggler events) to incorporate reverse computation were straight-forward, since the processed event list for an LP is scanned in reverse time stamp order.

The last major change to the GTW system was that all memory allocation for saving state (both copy state and incremental state-saving) was turned off. Also, the copy-state operation during forward event processing was turned off as well. Instead, a small bit vector was added to every event, which served as the working bits needed for saving the state information created by the instrumented model code, as described in Section 2.2. For example, the two bits, `b1` and `b2` of the multiplexor model in Section 2.1.2 are in fact mapped to the lower order bits of this event bit vector. The application can declare and use additional bits by specifying them as application-specific event data.

2.3.2 Reverse Event Handlers

In order to use the reverse computation support in GTW, it is necessary to define the reverse function for every application function that is invoked during event processing. To this end, first, the reverse functions were written manually by inspection, following the rules in Table 1 (the applications are described later in this section). This manual configuration for reverse computation is referred to as **GTW-RC**. Next, a reverse **C** compiler called `rcc`, was used to automatically generate reverse functions from **C** functions[53]. This automated configuration for reverse computation will be referred to as **GTW-RCC**.

2.3.3 State-Saving

The default state-saving technique in GTW is copy state-saving, in which a copy of the entire state is made before an event is executed. In CSS, the state is saved every time an event is processed. A variation of copy state-saving is called periodic state saving[27]. Periodic state-saving is a generalized technique in which state is saved only periodically, say, every p^{th} event, instead of every event as is done with copy state-saving. This implies that some events save state before processing, and others do not. The former set of events can be rolled back easily by restoring the state to the saved values. The latter set of events need special treatment, since they do not have saved state. The state restoration for these events is achieved by starting with a past processed event that does have saved state, and then re-executing the sequence of events from that past event to the event just before the rolled back event.

We incorporated periodic state-saving into GTW as a generalization of copy state-saving. The application can choose between copy and periodic state-saving by specifying its state-saving period to be equal to or greater than unity, respectively. The implementation is optimized for copy state-saving when the period equals unity. No source code changes are necessary in the application models to switch between copy and periodic state-saving. We shall refer to the copy and periodic state-saving configurations as **GTW-CSS** and **GTW-PSS** respectively.

Incremental state-saving is an alternative state-saving technique in which only the modified portions are saved just before modification. GTW includes an implementation of incremental state-saving in which the modifications are logged as pairs of integral address-value pairs, and stored in a log array for each LP. We shall refer to this state-saving configuration as **GTW-ISS**. Although GTW allows both copy state-saving and incremental state-saving to be used simultaneously together in the same application, we used them mutually exclusively, because of the uniformly small state sizes in our applications.

2.3.4 Applications

For the performance study, we use two applications: (i) a cascading network of Asynchronous Transfer Mode (ATM) multiplexors (ii) a Personal Communications Services (PCS) network.

- **ATM Multiplexor Cascade**: The first application consists of a 3-level cascade of ATM multiplexors, as described in [55]. The model is parameterized by a factor n, such that n^3 cell sources feed into n^2 multiplexors which in turn feed into n multiplexors, which finally feed into one multiplexor. The factor n is the number of inputs of each multiplexor. The GTW source code for the ATM multiplexor model was obtained from the Northern Parallel Simulator (Nops) group at Dartmouth [55]. Their implementation on GTW realizes each network element as an LP. The state size of each LP is 112 bytes. The application data contained within each message is 8 bytes. The event granularity of this application is very low (a few microseconds for small n).

- **PCS Network**: In the second application, a PCS network is simulated as described in [14]. The service area of the network is populated with a set of geographically distributed transmitters and receivers called *radio ports*. A set of radio channels are assigned to each radio port, and the user in the *coverage area* sends and receives phone calls using the radio channels. When a user moves from one cell to another during a phone call a *hand-off* is said to occur. In this case the PCS network attempts to allocate a radio channel in the new cell to allow the phone call connection to continue. If all channels in the new cell are busy, then the phone call is forced to terminate. For all experiments here, the *portable-initiated* PCS model was used, which discounts *busy-lines* in the overall call blocking statistics. Here, *cells* are modeled as LPs, and PCS subscribers are modeled as messages that travel among LPs. PCS subscribers can travel in one of 4 directions: north, south, east or west. The selection of direction is based on a uniform distribution. The state size for this application is 80 bytes with a message size of 40 bytes and the minimum lookahead for this model is *zero* due to the exponential distribution being used to compute call inter-arrivals, call completion and mobility.

The computation granularity of the ATM multiplexor model is very small, but, the communication among the LPs is feed-forward in nature, yielding excellent lookahead properties. The PCS network, on the other hand, has medium event granularity and possesses more complex communication patterns with much larger message sizes and a *zero* lookahead. Consequently, PCS is a more representative example of how a "real-world" simulation model would exercise the rollback dynamics of reverse computation.

2.3.5 Forward computation

In practice, one would like the serial performance of the parallel simulator to be as close to the optimized sequential as possible. With that in mind, our first set of experiments uses the ATM multiplexor model and compares the serial performance of GTW-NONE[3], GTW-RC, and GTW-CSS against GTW-SEQ on this model to determine the impact these different approaches have on forward computation rates. We did not use incremental state-saving in this comparison since it resulted in slower performance than full copy saving-saving. The cause of low performance of incremental state-saving was a consequence of the LP state being so small (only 112 bytes)[32]. We did not use the PCS network model, since it is of a higher granularity than the ATM multiplexor model, and hence less stringent than the ATM multiplexor model on the forward computation overheads.

Figure 7 shows the event rate as a function of fan-in for the four simulators. There are several key observations based on this performance data. First, we observe that the performance of GTW-RC is equal to GTW-NONE. The reason these two systems perform equivalently is because the few extra bits stored in the forward computation to support reverse computation has negligible impact on the overall event granularity of the ATM Multiplexor application. However, if we compare GTW-RC with GTW-CSS, a much different picture emerges — GTW-RC is consistently faster than GTW-CSS, the primary reason being that we have completely eliminated the overhead of state-saving.

If one were to eliminate state-saving overheads in an optimistic simulator, as we achieved in GTW-RC, we may expect to observe performance that is about equal to

[3] GTW-NONE is very much like a conservative parallel simulator being run serially.

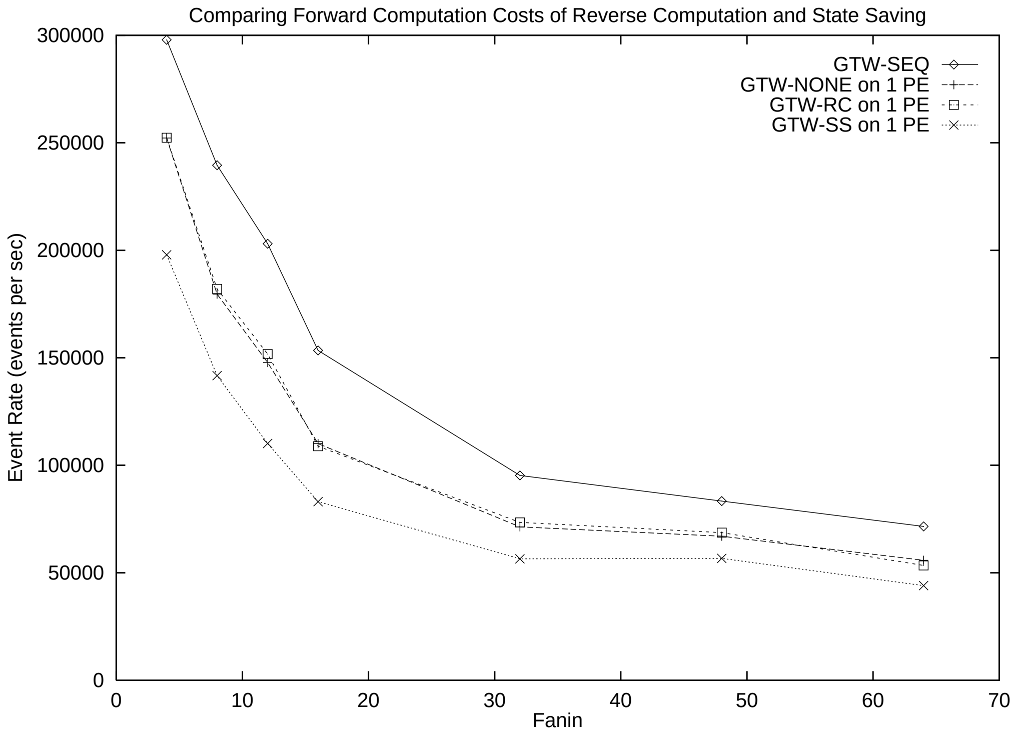


Figure 7: Forward Computation Performance of State-saving and Reverse Computation on the ATM Multiplexor Model

that of the optimized sequential simulator. But, clearly that is not the case here — across all fan-in values, the sequential simulator is faster, and, in one case, as much as 30% faster. To investigate this phenomenon, we profiled GTW-RC and GTW-SEQ to see where these two systems were spending most of their CPU cycles. Profiling revealed that the memory footprint of GTW-RC is much larger than that of GTW-SEQ. This is because the sequential simulator commits and immediately reuses an event memory buffer upon processing that event. But, GTW-RC (and GTW-CSS) only commits an event memory buffer when global virtual time (GVT) sweeps past the event time-stamp, which is approximately once every 1000 events. The consequence of waiting for GVT is that GTW-RC "touches" more pages of memory than GTW-SEQ, which results in more first and secondary data cache misses, as well as translation look-aside buffer (TLB) misses and page faults.

Finally, we observe that as the fan-in increases, the performance of the different simulators begins to converge. To explain this phenomenon, we need to understand how an increase in fan-in effects the system. Recall, there are n^3 sources in the multiplexor network. Each source generates two messages — one for self rescheduling, and the other when a cell is generated to send to the target multiplexor. Consequently, there are at any one instance, at least n^3 events in the system. Thus, the event population grows as the cube of the fan-in, n. As we approach fan-ins of 48 and above, the event-list management overheads begin to dominate, which decreases the impact state-saving overhead has on overall system performance.

In summary, in the fine-grained multiplexor model, we observe that reverse computation almost completely eliminates the state-saving overheads from the forward computation.

2.3.6 Parallel Simulation Performance

In this next series of experiments, we compare the *parallel* simulation performance achieved by reverse computation and state-saving on the ATM multiplexor model and on the PCS network model.

- **ATM Multiplexor**: For the experiments with the ATM multiplexor model, we chose a representative fan-in of 16 (to get a non-trivial network size that still keeps the event granularity sufficiently small) and varied the number of processors (2, 4, 8, 12 and 16). The experiments were performed separately for

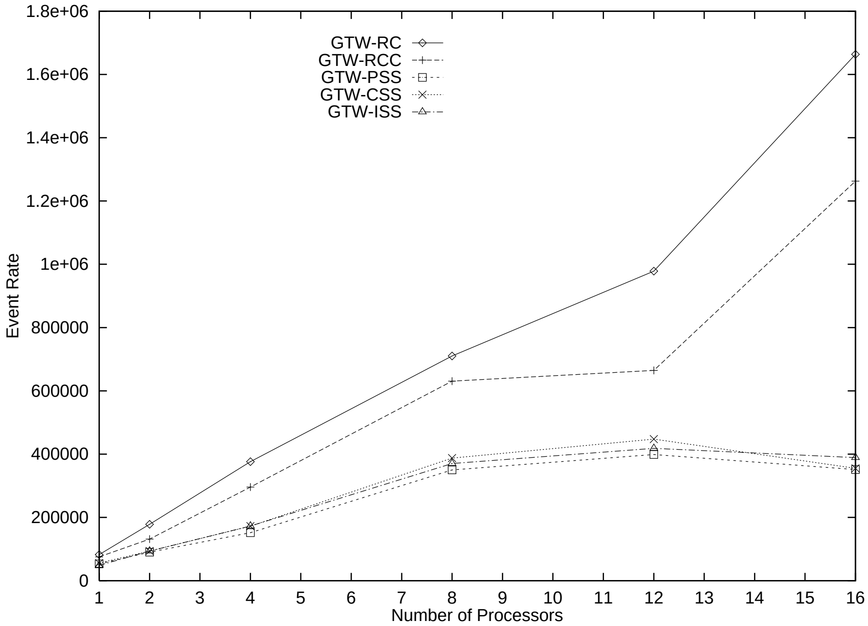


Figure 8: Event Rate of Reverse Computation and State-saving on the ATM Multiplexor Model (Fan-in 16)

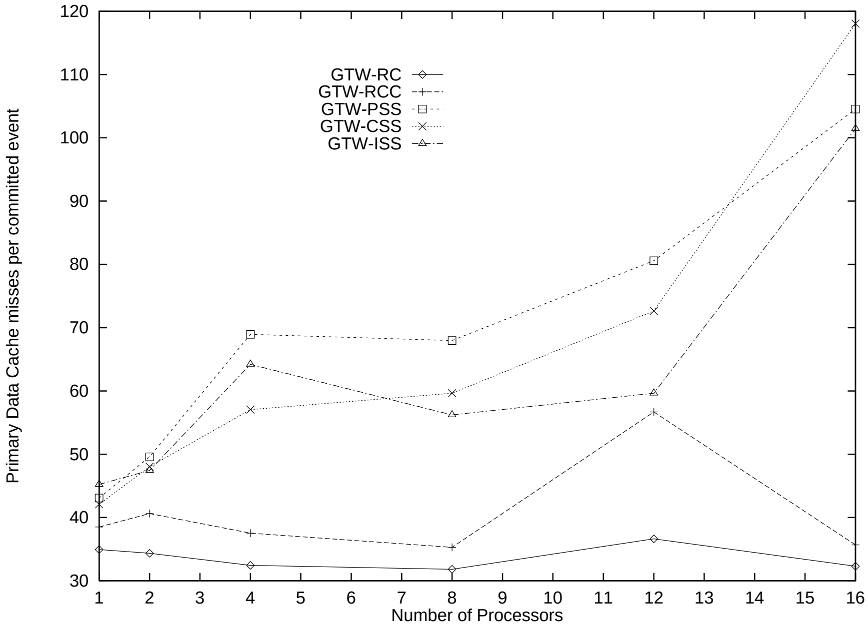


Figure 9: Primary Data Cache Performance of Reverse Computation and State-saving on the ATM Multiplexor Model (Fan-in 16)

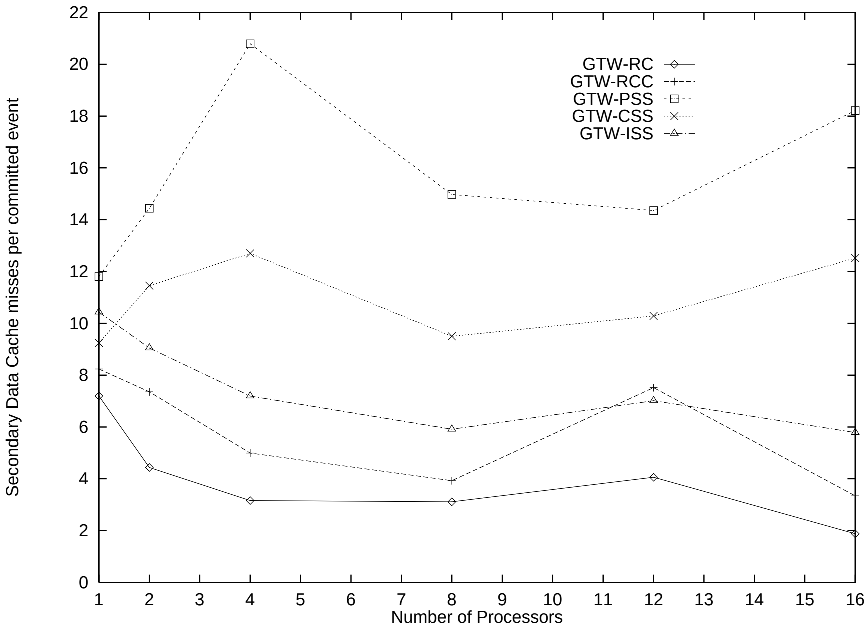


Figure 10: Secondary Data Cache Performance of Reverse Computation and State-saving on the ATM Multiplexor Model (Fan-in 16)

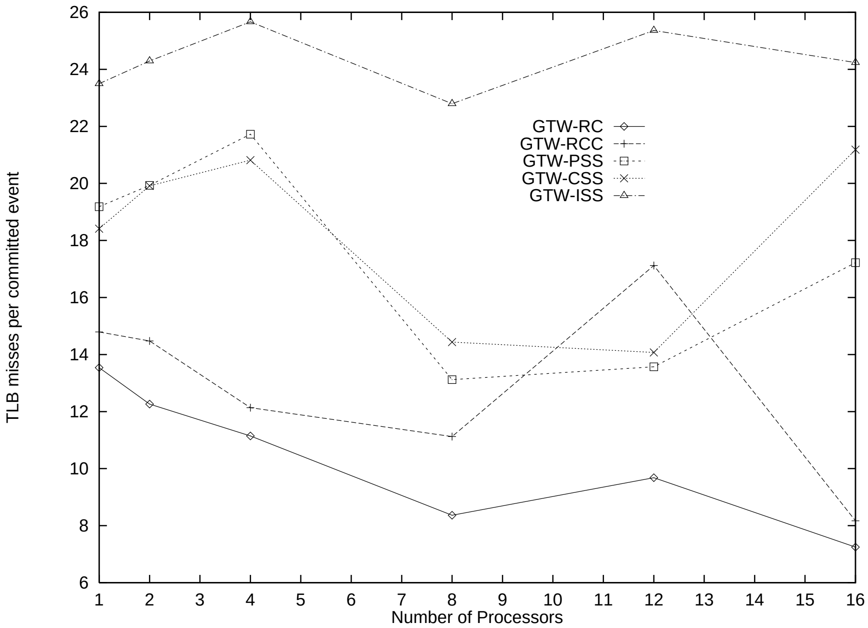


Figure 11: Translation Look-aside Buffer Performance of Reverse Computation and State-saving on the ATM Multiplexor Model (Fan-in 16)

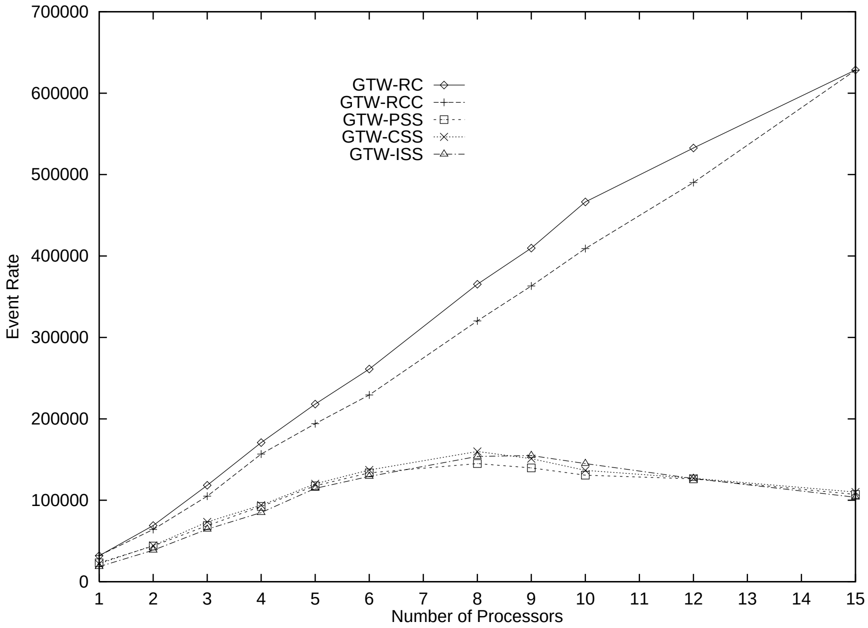


Figure 12: Event Rate of Reverse Computation and State-saving on the PCS Network Model

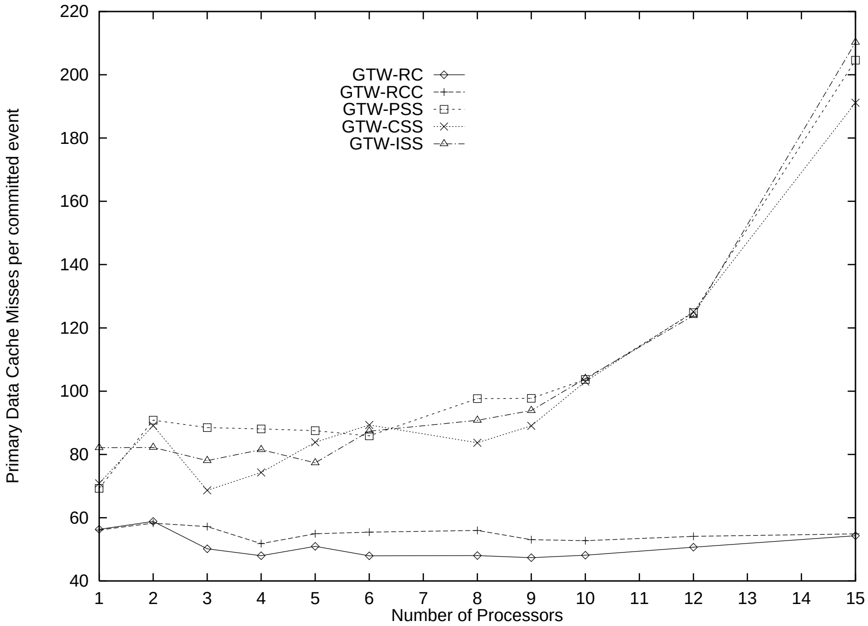


Figure 13: Primary Data Cache Performance of Reverse Computation and State-saving on the PCS Network Model

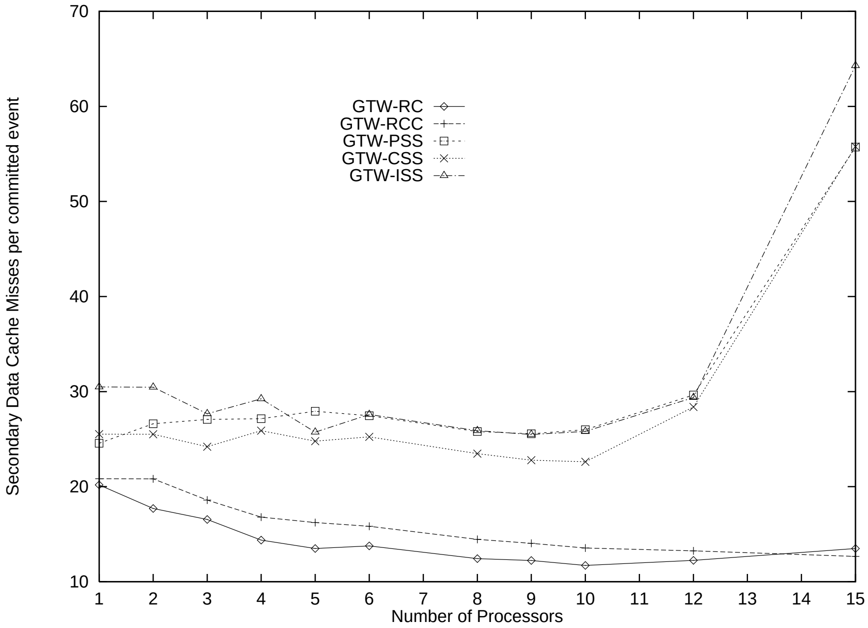


Figure 14: Secondary Data Cache Performance of Reverse Computation and State-saving on the PCS Network Model

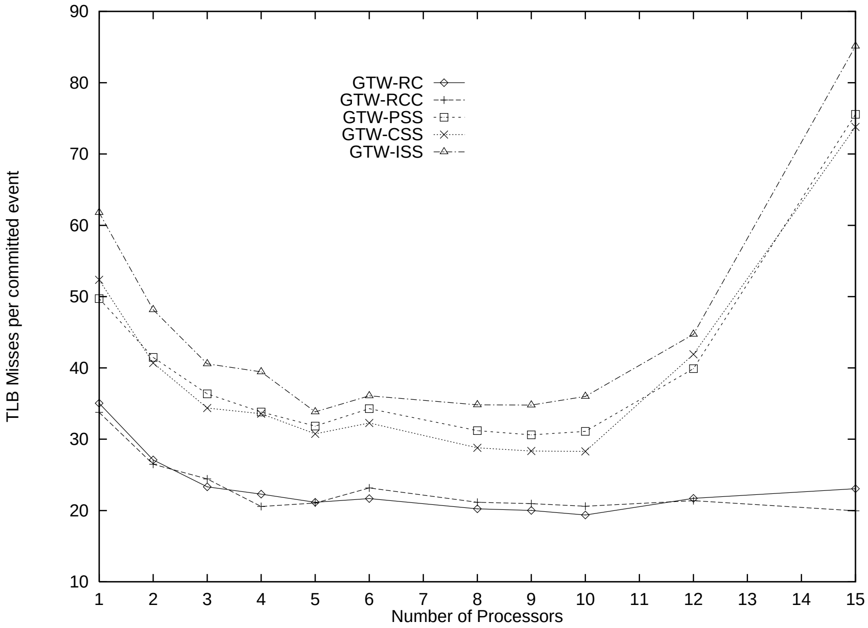


Figure 15: Translation Lookaside Buffer Performance of Reverse Computation and State-saving on the PCS Network Model

each of the GTW configurations – GTW-RC, GTW-RCC, GTW-CSS, GTW-PSS and GTW-ISS. Figure 8 compares the event rate obtained with all the configurations, on a varying number of processors.

- **PCS Network**: We also simulated the PCS model in parallel, and compared the parallel performance of state-saving and reverse computation, using the different GTW configurations on a varying number of processors. For these experiments the following PCS network settings were used. The PCS model was configured with a 64x64 LP grid for 8 processors, a 72x72 LP grid for 12 processors, and a 60x60 LP grid on 15 processors. For all LP configurations, the number of initial events per LP was 25. These LP configurations were chosen because they allowed an even number of LPs to be mapped to each processor to preclude introducing an unbalanced workload. The event rate performance for this set of experiments is shown in Figure 12.

Given the modest, nevertheless good, improvement in serial performance when using reverse computation, we expected to see a similar modest enhancement with respect to parallel simulation performance. However, we were surprised to see that reverse computation improved GTW's performance on the ATM Multiplexor model by up to 300% as compared to state-staving, and up to 500% on the PCS network model. We observe that in the 16 processor case of the ATM Multiplexor model, GTW-RC increased the event rate by a factor of more than 4 compared to GTW-CSS[4]. Similarly, in the 15 processor case of the PCS network model, GTW-RC increased the event rate by a factor of more than 5 compared to GTW-CSS. All the performance

[4]The raw event rate using reverse computation for that case was over 1.6 million events per second!

data were obtained by repeating the simulation runs several times. The performance results were found to be repeatable, with negligible variance. In all cases, we observe that GTW-RC is consistently and significantly faster than GTW-CSS, GTW-PSS and GTW-ISS.

These observations raised the next question, namely, why does reverse computation improve performance by such a large factor? We hypothesized that it is memory system related, assuming that reverse computation has a smaller memory footprint than state-saving and hence requires fewer resources to be expended by the memory subsystem. To verify our hypothesis and to precisely identify the source of the performance variation, we used the `perfex` performance tool. Here, we configured `perfex` to make use of the hardware counters internal to the MIPS R10000 processor to obtain extremely accurate performance statistics. We note that because the hardware counters were used, we observed neither slow down in performance, nor perturbation in the model performance due to the `perfex` monitoring software. Figures 11, 9 and 10 show the number of TLB, primary data cache, and secondary data cache misses for the ATM Multiplexor model. Figures 15, 13 and 14 show the corresponding statistics for the PCS network model. In these figures, the number of misses is normalized by a constant, which is the total number of events committed by the simulation. We observe that GTW-RC incurs significantly fewer primary and secondary data cache misses and TLB misses per event, compared to GTW-CSS, GTW-PSS and GTW-ISS. The net effect of the poor memory subsystem behavior of the state-saving variants is that the event rate degrades as the number of processors is increased. Some of the factors behind the low performance of state saving are explained next. More detailed analysis of the detrimental effect of state-saving on the simulation performance on shared memory multiprocessors is presented in [16].

Copy State-Saving

When copy state-saving is used, the footprint of the simulation is increased because of the additional memory required for state maintenance for each event. The increase in the additional memory size manifests itself in terms of an increase in the number of TLB misses incurred by the simulation per event. Furthermore, the simulation touches more memory pages per event, due to the act of making a copy of the state. This further increases the chance of cache misses and TLB misses. The misses contribute to contention at the shared memory, with the net effect of rapidly reducing the performance as the number of processors increases. These phenomena are indicated by the rapid increase in the number of primary data cache misses in the ATM Multiplexor simulation, as shown in Figure 9. Similarly, in the case of PCS network simulation, the number of primary and secondary data cache misses, along with the number of TLB misses steadily increase as the number of processors increases, as shown in Figures 13, 14 and 15. The net effect is that the event rate of the simulation deteriorates as more processors are added to the simulation.

Periodic State-Saving

Since periodic state-saving avoids saving state too often, it is fair to expect that its performance would be better than that of copy state-saving. This is because periodic state-saving can potentially reduce the overhead during the forward computation, and reduce the memory consumed for state-saving. This is especially true in applications in which the state size is significantly greater than the event size[6, 54]. However, in applications–such as used in this study–in which event size is comparable to (or greater than) state size, periodic state-saving can have the counter-effect of actually increasing the memory utilization relative to copy state-saving. This is attributable to the fact that events cannot be reclaimed until global virtual time (GVT) goes past the earliest among all logical processes of their latest state-saved event. With copy state-saving, events can be reclaimed as soon as GVT sweeps past their time stamps, which allows them to be reused immediately thereafter. In contrast, with periodic state-saving, a processed event can be reclaimed only if there exists another processed event with saved state whose timestamp is less than GVT. Hence, the working set of simulations using periodic state-saving is greater in size relative to that of copy state-saving, due to a greater number of events maintained between GVT computations. (In fact, this relation is preserved independent of the frequency with which GVT is computed.). The net result is that periodic state saving does not help in significantly reducing the state-saving overheads relative to copy state-saving. This is confirmed by the similarity of the memory subsystem performance of GTW-CSS and GTW-PSS, as shown in Figures 9, 10, and 11, and in Figures 13, 14, and 15.

Incremental State-Saving

Since incremental state-saving avoids copying the entire state, and copies instead only the pieces that have been modified, it can be normally expected to incur less overhead than copy state-saving. This is because, incremental state-saving can potentially consume less memory than copy state-saving in applications with large state sizes. If only a small portion of the state is modified per event, then incremental state-saving can result in significant reduction in state-saving overheads per event relative to copy state-saving. However, in applications–such as used in this study–in which the state size is small, the overheads of maintaining a log of changes (lists of address-value pairs) is significant, making it no better than copy state-saving. This is confirmed by the similarity of the memory subsystem performance of GTW-CSS and GTW-ISS, as shown in Figures 9, 10, and 11, and in Figures 13, 14, and 15.

Automated Reverse Code

Finally, we note that the hand-coded reverse code (GTW-RC) performs slightly better than the automated reverse code (GTW-RCC). This is because the hand-coded reverse code incorporates more optimizations than the automatically generated reverse code. For example, bit operations are more customized for each application in the hand-coded version, whereas the automated version uses more generalized bit operations. Consequently, the automated version adds a slight amount of additional computational overhead over that of hand-coded version. This is evident in the slightly reduced event rates for GTW-RCC relative to GTW-RC, as shown in Figures 8 and 12. The fact that the memory characteristics of GTW-RC and GTW-RCC remain the same is indicated by the similarity of their memory performance results, as shown in as shown in Figures 9, 10, and 11, and in Figures 13, 14, and 15.

2.3.7 Performance Summary

The results presented here, when considered in their totality, indicate that the performance of optimistic parallel simulation has reached an acceptable level for this class of extremely low event granularity applications. Previously, researchers in the area of parallel and distributed simulation have indicated difficulty in achieving acceptable levels of performances from Time Warp systems with small event-granularity (e.g. [68]). They observed that state-saving costs were dominating and stifling performance. Now, with reverse computation, it appears to have become harder to make such arguments against using optimistic approaches on such applications.

As future generations of processors become faster and the performance gap between memory and processors widens, reverse computation can be expected to achieve even higher performance compared to state-saving.

2.4 Related Work

Reverse computation has been previously studied in various contexts. Research into *reversible computing* is aimed at realizing reversible versions of conventional computations in order to reduce power consumption [7, 57]. The `R` language is a high-level language with special constructs to enforce reversibility so that programs written in that language can be translated to machine code of reversible computers [24]. Another interesting application of reversible computation is in garbage collection. The `Psi-Lisp` language presented in [5] uses reversible constructs to efficiently implement garbage collection. Other applications for reversible execution are in the areas of database transaction support, debugging support and checkpointing for high-availability software [40, 62, 11]. More recent work is concerned with source to source translation of popular high-level languages, such as **C**, to realize reversible programs. However, almost all of the solutions suggested in these application areas translate either to constraints on language semantics to disallow irreversible computations, or to techniques analogous to state-saving techniques (specifically, copy-on-write techniques) of optimistic parallel simulations. Some of them operate at a coarse level of virtual memory pages. The optimizations are roughly analogous to those used in incremental state-saving approaches in parallel simulations. Moreover, since these solutions are not specifically geared towards parallel simulations, they are not optimized for minimizing the state size, and do not adequately exploit the semantics of constructive operations.

In [10], reversible computing has been suggested as a method for testing failures in real-time systems, but with admittedly high forward and reverse computing overheads, and without treatment of complex instructions such as inter-mixing jumps. An initial attempt at automatically generating symbolic inverses of *reversible* functions is made in [23], but it relies on heuristics for correctness. A more theoretical approach is taken in [18], by using inversion of invariants to prove the correctness of inverse programs. A debugging system is described in [11] that executes **C** programs in interpreted mode in forward and reverse directions. Although their approach using interpretation is well suited for debugging systems, the performance characteristics of their techniques are unclear when applied to high-performance simulations. An interesting use of reversible computing is in its application to the automatic differentiation of functions expressed in a high-level computer language, such as **C**/**C**$^{++}$[33, 34]. For this, reverse execution of certain intermediate computations is necessary, which is achieved via operator-overloading techniques of **C**$^{++}$.

The state-saving techniques presented in [32] utilize a limited form of optimization using the reverse computation approach and is the first work we are aware of to specifically discuss reverse computation for simulation, but no performance results are provided. Our work starts where [32] ends, and is concerned with techniques for minimizing the state size for realizing reversibility, and simultaneously minimizing the runtime execution overheads. Finally, in [65], a *rollback relaxation* scheme is presented that automatically identifies certain types of history-independent logical processes and optimizes the performance of rollback activity for those processes. Our approach is different in that it addresses logical processes which are not necessarily stateless, and seeks to optimize run-time performance and memory utilization by minimizing the essential state required by such processes.

2.4.1 Relation to Perfectly Reversible Computing

It may be noted that specialized languages can indeed be carefully defined in such a way that their constructs never permit any destruction of information during program execution. In other words, programs written in such languages are perfectly reversible. However, such languages are not the focus of this work, since they are not commonly used. Moreover, there is an important distinction between the work described here and other related work on perfectly reversible computing. In perfect reversibility, a forward function `f()` and its reverse `rf()` are defined to be commutative on the program state S:

$$\mathtt{rf}(\mathtt{f}(S)) \equiv S \equiv \mathtt{f}(\mathtt{rf}(S))$$

In other words, `f()` and `rf()` are defined to be perfect inverses of each other[5]. However, in applications of our interest, such as parallel simulation, the property of

[5]If S is n bits long, the number N of distinct programs $g() : S \rightarrow S$ is given by: $N = \sum_{k=1}^{2^n} \binom{2^n}{k}\binom{2^n}{k-1}$. The functions `f()` and `rf()` are two among them.

commutativity is in fact not necessary. Instead, only the left half of the previous identity is sufficient for practical purposes:

$$\texttt{rf}(\texttt{f}(S)) \equiv S, \text{ but } S \neq \texttt{f}(\texttt{rf}(S))$$

For example, consider the following fragment in which an irreversible program fragment `f'()` is transformed to a reversible version `f()` by adding a bit variable `b`, while `rf()` is the reverse of `f'()`:

<table>
<tr><th>Original</th><th>Reversible</th><th>Reverse</th></tr>
<tr>
<td><pre>
int x = 0, y = 1;

f'()
{
    if(x > y)
        y += x;
    else
        x += y;
}
</pre></td>
<td><pre>
int x = 0, y = 1;
bit b;

f()
{
    if(b = (x > y))
        y += x;
    else
        x += y;
}
</pre></td>
<td><pre>
int x = 0, y = 1;
bit b;

rf()
{
    if(b)
        y -= x;
    else
        x -= y;
}
</pre></td>
</tr>
</table>

In this, the state $S = \{\texttt{x}, \texttt{y}, \texttt{b}\}$. It is clear that $\texttt{rf}(\texttt{f}(S)) = S$. This is because `x` and `y` are restored to their original values, and `b` (being an added variable) remains in a *don't care* state. However, $\texttt{f}(\texttt{rf}(S)) \neq S$.

Perfect reversibility supports the power of the retracing of execution by starting from the final state and reverse computing towards the initial state. Such power is not always necessary for the practical applications such as optimistic simulation, in which forward reversibility is sufficient. The reduced power of this type of "half" reversibility can translate to savings in runtime and memory overheads.

2.4.2 Reversible Stochastic Processes

In traditional Time Warp simulations, rollback is used to undo incorrect computation in such a way that the state is restored to the exact same value that the state had before the incorrect computation was performed. However, in certain models, it may not be necessary to restore the exact same value; the model may be stochastically resilient to deviations, and therefore the final simulations results will be close to those of an exact simulation. For example, consider the models containing time reversible processes, say, certain Markov chain processes[37], which have been well studied in the context of queueing networks. In such models, the reversed system may be indistinguishable from the original system in a stochastic sense. Hence, instead of performing an exact inverse operation during rollback, it may be possible to simply re-execute the forward process itself again, and indirectly achieve the reverse effect. Intuitively, state transitions in a Markov chain process can be reversed either by computing the exact inverse transitions, or alternatively by making further normal transitions which effectively manage to reach the original state.

The are two potential benefits of using the forward computation for reversal. First, it reduces and/or eliminates the need to save state information during the forward computation. Next, to apply reverse computation, it is not even necessary to go through the process of finding the reverse code corresponding to the forward.

The drawback of this approach is that it tends to be very model-specific and therefore difficult to extend to other models in general.

2.5 Remarks

Reverse computation is well suited for models containing constructive assignments. However, without adequate care, it can degenerate to traditional state-saving if a sufficiently large number of destructive assignments, which are hard to reverse, are present in the model. In fact, in certain cases, it can perform worse than incremental state-saving, due to the fact that optimizations, such as the merging of multiple writes to the same variable into a single save operation, are possible using incremental state-saving techniques, but not readily possible with reverse computation.

There is a commonly implemented optimization in copy state-saving: when a rollback spans several processed events, it is sufficient to merely switch a few pointers in order to restore the entire state to its value corresponding to the earliest rolled back event. This helps in considerably reducing the rollback cost. In contrast, when reverse computation is used, each one of the rolled back events must be reversed one at at time, in the reverse order of processing. This can potentially make the rollback cost much higher than that of copy state-saving. Advanced inter-event analysis is necessary in order to reduce such overhead in reverse computation.

On the other hand, previously, optimistic simulations were considered to be unsuitable for fine-grain applications because of the high state saving overheads. It has been shown that reverse computation is an appealing alternative approach that makes efficient optimistic simulation of fine-grain applications feasible.

Some classes of applications are identified in which reverse computation is natural. In these applications, automatic techniques are easily found that essentially exploit the source code as state. Examples include quantum computer simulation, and queuing network simulation. In the case of queuing network models, it is recognized that a majority of the common operations are indeed reversible. In particular, the reversibility of the most common operation, namely, random number generation is addressed. In addition, it is observed that other queue manipulation operations, such as insert, delete and shift, are in fact more memory efficient with reverse computation than with state-saving.

In other classes of applications, this approach also serves as an automatic compiler-based state-compression technique. State compression is useful for enhancing the performance of optimistic simulations in limited memory environments. Considering that CPU resources are cheaper and more abundant than memory resources, certain important classes of applications (such as queuing networks) can be expected to be executed using optimistic parallel simulation on a network of, say, palm-top computers. The state-compression is useful even in the context of state-logging conservative parallel simulations and sequential simulations. For interactive (play-log-replay) applications, there can be significant benefits in terms of reduction in memory requirements of the state log. Since the applications tend to be simulated for long times, an order of magnitude difference in the size can be quite significant. (In this case, we are still investigating the gains of state-compression as opposed to using standard compression programs, such as `gzip`, on the log of regular uncompressed state.)

Most importantly, the reduced memory requirements due to state-compression allow us to explore new applications that were considered too expensive to simulate using state-saving-based optimistic simulations. However, several open issues remain to be explored, which are identified in chapter 5.

Table 1: Summary of State Bit Sizes for Various Statement Types
Generation rules and upper bounds on state size requirements for supporting reverse computation. s, or $s_1..s_n$ are any of the statements of types T0..T7. inv(s) is the corresponding reverse code of the statement s. b is the corresponding state-saved bits "belonging" to the given statement. The operator $=@$ is the inverse operator of a constructive operator $@=$, (e.g., $-=$ for $+=$).

Type	Description	Application Code			Bit Requirements		
		Original	Instrumented	Reverse	Self	Child	Total
T0	simple choice	if() s_1; else s_2;	if() {s_1; b=1;} else {s_2; b=0;}	if(b==1) {inv(s_1);} else {inv(s_2);}	1	x_1, x_2	$1 + max(x_1, x_2)$
T1	compound choice (n-way)	if() s_1; elsif() s_2; elsif() s_3; else() s_n;	if() {s_1; b=1;} elsif() {s_2; b=2;} elsif() {s_3; b=3;} else {s_n; b=n;}	if(b==1) {inv(s_1);} elsif(b==2) {inv(s_2);} elsif(b==3) {inv(s_3);} else {inv(s_n);}	$lg(n)$	x_1, x_2, ..., x_n	$lg(n)+ max(x_1, ... x_n)$
T2	fixed iterations (n)	for(n) s;	for(n) s;	for(n) inv(s);	0	x	$n * x$

T3	variable iterations (maximum n)	while()
		s;
			b=0;
			while()
			{s; b++;}
				for(b) inv(s);	$lg(n)$	x	$lg(n)+n* x$

T4	function call	foo();	foo();	inv(foo)();	0	x	x
T5	constructive assignment	$v@ = w$;	$v@ = w$;	$v = @w$;	0	0	0
T6	k-byte destructive assignment	$v = w$;	$\{b = v; v = w;\}$	$v = b$;	$8k$	0	$8k$

T7	sequence	s_1;
		s_2;
		s_n;
			s_1;
			s_2;
			s_n;
				inv(s_n);
				inv(s_2);
				inv(s_1);
					0	$x_1 + ... + x_n$	$x_1 + ... + x_n$

T8	jump (label lbl as target of n goto's)	goto lbl;
		s_1;
		goto lbl;
		s_n;
		lbl:
		s;
			b=1; goto lbl;
			s_1;
			b=n; goto lbl;
			s_n;
			b=0; label:
			s;
				inv(s);
				switch(b)
				{
				case 1: goto $label_1$;
				case n: goto $label_n$;
				}
				inv(s_n); $label_n$:
				inv(s_1); $label_1$:
					$lg(n+1)$	0	$lg(n+1)$

T9	Nestings of T0-T8	Apply the above recursively			Apply the above recursively		

Chapter 3

STACK RECONSTRUCTION

Process oriented views are widely recognized as very useful for modeling, but difficult to implement efficiently in a simulation system, as compared to event oriented views. In particular, the complexity and run-time overheads of the implementation have prevented the widespread use of process oriented views in optimistic parallel simulations. In this chapter, the conventional approaches to implementing process-oriented views are reviewed, and some of the sources of problems in those approaches are outlined. In addition, an approach called stack reconstruction *is identified, which is well suited for portably and efficiently supporting optimistic process-oriented views. Benchmark simulations using its implementation, which is incorporated in the* TED *modeling and simulation system, confirms the low overheads of this approach, and demonstrates its capability to simulate over one million processes in a process-oriented model.*

Three widely recognized world views for simulation are: event-oriented view, process-oriented view and activity-scanning view [44]. The first two views are the more widely used among the three views. All three views are equivalent in the sense that process-oriented and activity-scanning views can be translated into semantically equivalent event-oriented views. An advantage of using the process-oriented view is that models tend to be smaller, and easier to develop and understand, making it more appealing to the modelers. On the other hand, it is generally perceived that event-oriented views can be more efficiently implemented than the other two views, especially in the context of optimistic parallel simulations.

In recent modeling and simulation efforts, there is a clear demand for the capability to support very large scale simulation. The need for *parallel* simulation is clear in these application domains. In addition, the models tend to be too complex to express using the low-level primitives of the event-oriented view, thus making process-oriented view a natural choice for developing such large and complex systems[60]. This makes it important to find ways to efficiently support the very large number of processes in a process-oriented modeling and simulation system.

Specifically, our experiences with modeling large and complex telecommunication networks using the TeD language [49, 46] served as our initial motivation to find efficient implementation alternatives for process-orientation. For example, in many interesting configurations (such as global Internet) of network models in TeD, the number of processes easily exceeds one million, warranting efficient support for large-scale process orientation. Also, modelers demand support for full process orientation, such as the ability to invoke `wait` statements over nested procedure calls, at almost arbitrary points in a procedure body. TeD permits the direct embedding of **C**$^{++}$ code in the models [50], further constraining the range of alternatives that can be explored to trade-off modeling power for efficiency.

The interaction of parallel simulation techniques with the implementation of large-scale process-oriented views (supporting millions of processes per simulation) generates challenges, such as minimizing the memory size and memory copy requirements of the implementation. Process-oriented views are generally perceived to be expensive in the context of optimistic parallel simulations. This is mainly due to the fact that processes entail state-saving overheads for the maintenance of additional control flow information and transient data. These overheads can be quite large in naive implementations, unless they are carefully reduced to the minimum necessary.

Related Work

A number of implementations of process-oriented views (which are mostly language or preprocessor-based) have been reported in the recent years. The Maisie language [4] supports the concept of process in the form of an entity description, along with support for the optimistic parallel simulation of Maisie entities. However, Maisie does not include direct support for the suspension of a stack of nested procedure calls (wait statements cannot be used inside functions invoked by entities). The macros-based approach of IMPORT/SPEEDES [66] also seems to limit simulation time advances to the main process body. Apostle [12] is a new language that implements process-orientation using continuations, with support for optimistic parallel simulations. Apostle, however, has specialized semantics, which do not carry forward well to our domain of interest, which is $\mathbf{C}^{++}$. Other languages and packages, such as MODSIM and Parasol support process-orientation views, but without great success in optimistic simulations. More recent work includes the Nops system [55] which reports low overhead implementation of processes. Built over the Cilk language [25], it focuses on an efficient implementation of process-oriented views. to support TeD language models, Nops, however, only supports *conservative* parallel simulation, and it has no direct support for advancing simulation time over a stack of nested procedure calls.

Here, an approach called *stack reconstruction* is identified as well suited for portably and efficiently implementing optimistic parallel simulations of "true" process oriented views in an expressive language such as **C**$^{++}$. First, definitions are made of what constitutes true process orientation, in section 3.1. Next, some conventional implementation approaches are briefly reviewed in section 3.2, identifying their problems in the context of optimistic parallel simulations. Then the stack reconstruction approach is described with details of implementation, followed by some salient performance results which indicate the low overheads of the approach.

3.1 Process Orientation

A process is a distinct flow of control, containing a combination of computation and synchronization operations. Processes typically synchronize with each other by exchanging events. Process orientation is an elegant way of capturing context information under the conventional procedural programming paradigm. (Process or event orientation is orthogonal to object orientation. In an object-oriented setting, procedures in fact correspond to method calls.) The main body of a process can invoke procedures which in turn can invoke other procedures.

3.1.1 Functionality

The ideal modeling capabilities of a process are listed in tables 2 (a) and (b). Note that the features are generally orthogonal to each other (i.e. it is possible to pick and choose a subset of the features that will be adopted by a process-oriented system). A *pure* process-oriented view is one in which all the features **F1** through **F5** are

Table 2: Features of an Ideal Process-oriented System

F1	Procedures can declare and use local variables
F2	Procedure calls can be nested
F3	Procedures can be recursive and re-entrant

(a) Programming style

F4	Primitives to advance simulation time can be invoked in any procedure
F5	Primitives to advance simulation time can be invoked wherever a conditional, looping or other statements can appear.

(b) Time control

supported. The first three features are those that are expected of any modern languages supporting the procedural programming style, and expected by most modern programmers. The last two features are specific to the simulation domain — time advancing primitives are those that serve to advance simulation time, such as `wait`, or `hold` statements and other such variants. It is the interaction of the programming style with the simulation time advances that gives rise to interesting challenges in implementing process-oriented views efficiently.

3.1.2 Process Type Continuum

Although pure process-oriented views are useful in some complex models, more restrictive definitions of processes can be made for use in some models which do not warrant the full power of pure process-orientation.

By varying the combination of features that are supported, the efficiency of implementation can be varied. For example, if the features **F4** and **F5** are unused by a process-oriented model (i.e., simulation time is advanced only at the top–level statements in the main process body, as opposed to under **if** statements or inside a procedure), then such a model can be implemented in a way that incurs no more overheads than if that model was re-written using event-oriented view. In fact, such processes are nothing but event-oriented models expressed in a way that better exposes context information. Such models with lower demands on the expressive power do appear in real-life modeling, such as cited in [51].

Similarly, if the feature **F1** is relaxed, and a rule is enforced that all local variables are *immutable* (i.e., never change after initialization), or if local variables are not supported at all by the modeling language, then, issues such as re-entrancy and recursion become easier to handle in the implementation.

Other simplifications (such as in Maisie, IMPORT and Nops) preclude the feature **F4**, but do support feature **F5**. In other words, although simulation time cannot be advanced in procedures, it can be advanced at any point in the process body (say, under conditional and looping statements). In such implementations, a stack of suspended procedures must be indirectly *emulated* using a chain of stack-less processes. The issues of local variables and recursion are also simpler to handle, especially in optimistic simulations, due to the reduction in the amount of information to state-save, and, the issues of local variables and re-entrancy can be effectively delegated to the host language, such as $\mathbf{C}^{++}$.

3.2 Implementation Alternatives

There exist several techniques for implementing process-oriented views, of which the important ones are outlined. One approach is to view the control and data information of a process as a black box, and preserve its contents across suspension and resumption points — this is the *threads*-based approach, such as in [42]. Another approach is to define the modeling language semantics in such a way as to effectively remove the need for a stack — the *continuations*-based approach of [12] is an example. Yet another approach is to transparently maintain auxiliary information that is barely sufficient to restore the native stack of a suspended process — which is the *stack reconstruction* approach described here. Each of these approaches is described next.

3.2.1 Process Stack

Many modern languages (compilers, to be precise) use an optimization, called *physical* or *native stack*, for the very frequent type of operation: procedure call. A native stack is an encoding of program counter, return addresses, local variables and argument lists[38]. Although the logical stack can be implemented in other ways (say, using linked lists), it is very often represented in contiguous memory locations for performance reasons. The native stack is used to efficiently implement the procedure call semantics by pushing and popping invocation information on and off the stack. Since the memory size of local variables and argument lists varies across procedures, the native stack serves to optimize the memory allocation and deallocation operations, by exploiting the contiguous memory feature of the native stack.

The preceding way in which native stacks are used is tightly coupled to the operation of executable code that most compilers generate, and tightly integrated into the way many operating system services (such as signals) operate.

Since the procedure stack is precisely what is necessary to support process-oriented views, it is natural to attempt to utilize native stacks to implement the simulation processes. This is the approach taken in implementations based on *threads*.

3.2.2 Threads

Threads are light-weight computation abstractions widely used in many domains such as high performance computing and multi-media servers. Threads provide support for multiple process stacks which can communicate and synchronize with each other. Several multi-threading packages exist that provide operating system-level threads and/or user process-level threads [64]. Portable implementations of threads are available, which typically make use of standard facilities such as the POSIX calls `setjmp()` and `longjmp()`. Threads can be scheduled, suspended and resumed. Each thread typically contains a stack of procedure activation frames, although some optimized packages use special thread synchronization semantics to avoid using physically distinct stacks for each thread.

One way to implement simulation processes is to use a single thread for each process. The thread suspension and resumption primitives can be used to achieve the simulation time advances in the simulation process code. Thus, for example, a `wait` statement in the simulation process will be mapped to suspending the thread of the simulation process, and handing control over to the simulator's scheduler. At the instant the waiting condition is satisfied, the process is resumed just after the wait statement, by resuming the thread of the simulation process.

The advantage of using threads to implement simulation processes is that little additional implementation work is necessary to save and restore the stacks of the simulation processes. The simulator only acts as the thread scheduler. There are, however, several drawbacks to threads-based implementation. Either thread stacks bump into each other (thread stacks typically do not grow), or the memory requirements can be high to support a very large number of processes. Thread migration is either unsupported or expensive. More importantly, conventional threads are difficult to optimize for optimistic parallel simulation (as discussed in more detail in the next section). The design of off-the-shelf thread systems may not be well suited to scale to millions of active threads. Special large-scale multi-threading systems exist, which could potentially be useful in sequential and conservative parallel simulations; but they have not been tested for use in optimistic parallel simulation.

Fundamentally, conventional threads are general-purpose computation abstractions, with potentially complex inter-thread synchronization, and scheduling disciplines. Simulation processes, however, have simple and well defined suspension and resumption semantics (based on simulation time advances), and a simple scheduling discipline (usually, least timestamp first). Whereas support for preemptive threads incurs overheads such as saving register state, simulation processes on the other hand need never incur such overheads, due to their simpler scheduling discipline.

3.2.3 Continuations

Continuations [1] constitute another efficient mechanism for implementing processes. Suppose the modeling language is defined in a way that allows the compiler to cast all language constructs (such as conditional or looping statements) into separate blocks of non-interruptible operations. In such a case, the process can be implemented as a special form of a finite state machine, in which each state dynamically designates its successor, called a *continuation*. If the issue of local variables is ignored for simplicity, it is clear that the process context is fully represented just by the identity of the current continuation (pointer to a function) of the process. Further optimizations are possible whereby explicit storage of the per-process continuation information can be avoided, and implicitly recorded on the run-time stack of the simulator [12].

To effectively implement this technique, either special language constructs have to be defined, or compilers of existing languages have to be modified [1].

3.3 Stack Reconstruction

Another approach to implementing process-oriented views is what we call *stack reconstruction.* The underlying idea is that, instead of saving and restoring the contents of the native stack, separate information is maintained at runtime such that the native stack can be reconstructed to the same state in which it was when the process was suspended. This not only allows us to throw away the unnecessary contents of the native stack, but also permits us to easily capture modifications to the process state, which is essential for state-saving operations in optimistic parallel simulation.

A compiler-based solution is used for supporting this approach transparently, leaving the models uncluttered with the implementation details. It is assumed that the models are translated into some general purpose programming language code, such as **C**$^{++}$ which is then compiled to result in executable models. **C**$^{++}$ is used as the target language in the examples.

To understand how the stack reconstruction approach works, consider the pseudocode fragment in figure 1 of a process-oriented model containing two procedures `one()` and `two()`. For simplicity, the treatment of local variables and procedure arguments is postponed until later in the discussion. The procedure `two()` contains

```
procedure one()
{
    ...
    wait(c2)
    ...
}
procedure two()
{
    s1
    if(...) {
        s2
        wait(c1)
        s3
    }
    for(...) {
        s4
        call one()
        s5
    }
}
process p()
{
    ...
    call two()
    ...
}
```

Process `p` invokes procedure `two()`, which in turn waits on a condition, and invokes procedure `one()`.

Figure 16: Model Code of a Process Making Nested Procedure Calls

some (arbitrary computation) statements, `s1`, `s2`, $\cdots$, `s5`. In addition, it contains a simulation advance (`wait`) statement inside a conditional (`if`) statement, and a procedure call to `one()` inside a looping (`for`) statement.

Consider the execution of process `p` when it invokes the procedure `two()`. If and when the procedure execution reaches line 12, it must be suspended at the **wait** statement, and when the wait condition is satisfied, it must be resumed at line 13 with the statement `s3`. Similarly, when the execution reaches line 17, the procedure `one()` must be invoked; again, process execution may have to be suspended if the procedure `one()` invokes some other `wait` statement, and resumed at the correct position in procedure `one()` when the process is resumed.

In the stack reconstruction approach, the compiler identifies and marks all the positions in a procedure at which a process suspension can occur. Lines 12 and line 17 qualify as suspension points of procedure `two()`. The compiler then assigns ordinal numbers 1 and 2 to the two suspension points. When the execution of procedure `two()` is suspended, it is sufficient to remember the ordinal number (or, jump index, JI) of the point where it was suspended. Using this number, we can directly jump (using a combination of `switch()` and `goto` statements at the beginning of the procedure) to the point where the execution was left off. This is demonstrated in figure 2 which lists the code generated from the model of figure 1. Note that the resumption point for a `wait` statement is just beyond the `wait` statement, whereas the resumption point for a procedure `call` statement is exactly at the same procedure `call` statement, which results in a re-invocation of the procedures.

The unrolling of native stack occurs when the process is suspended — all the procedures actually perform a `return`, returning control to the simulation system. The

```
int two()
{
    switch(JI) {
        case 0: goto start;
        case 1: goto lbl_1;
        case 2: goto lbl_2;
    }
    start: /*continue*/;
    s1
    if( ... ) {
        s2
        wakeup = c1;
        JI = 1;
        return SUSPENDED;
        lbl_1: /*continue*/;
        s3
    }
    for(...) {
        s4
        lbl_2: flag = one();
        if( flag == SUSPENDED )
        {
            JI = 2;
            return SUSPENDED;
        }
        s5
    }
    JI = 0;
    return DONE;
}
```

Figure 17: Extracts of Code Generated for Procedure `two()`

reconstruction of native stack occurs when the process is resumed — the procedure call chain is correctly reconstructed using function re-invocation, with the help of the saved ordinal numbers.

If each procedure can have at most 256 suspension points in its body (which is a reasonable limit for human-written models), a single byte is sufficient to record the jump index for each procedure. An array of jump indices can be used for each suspended process to record the jump indices of its active procedures. (The jump indices represent *forward* addresses albeit, more memory efficient, as opposed to the *return* addresses of conventional native stacks.)

3.3.1 Local Variables

Now let us consider the implementation of local variables. We maintain a pointer to a memory buffer (*frame*) along with each jump index. References to the local variables in the procedure body are translated to indirect (pointer) references to the frame. In optimistic parallel simulations, the frame remains allocated until the global simulation time (GVT) sufficiently advances to guarantee that the frame deallocation will not be rolled back.

Procedure arguments can be viewed as special type of local variables, which are initialized automatically by the compiler based on the procedure invocation. Hence, the compiler can treat them as such, and follow the same techniques as for local variables to save modifications to the arguments. A value of 0 for the jump index can be used by the compiler to distinguish between invocation and reconstruction, to enable it to initialize the arguments upon invocation of a procedure, and skip the initialization if the procedure is re-invoked during stack reconstruction.

3.3.2 Optimistic Simulation

The important feature of stack reconstruction that helps in optimistic simulations is that it allows to conveniently and transparently trap modifications to the logical stack. The logical stack consists of jump indices and local variables. Jump indices are incrementally state saved by the compiler, since the compiler manages them itself. Local variables can also be incrementally state saved using transparent incremental state saving techniques, such as those using overloaded assignment operators [59], by trapping modifications to local variables in the procedure body. The stack reconstruction approach is also appealing for optimistic parallel simulation due to its reduction in the amount of state-saved information.

In native stacks, it is the case that not only more extensive information is stored on the stack, but also it is hard to gain precise access to any and all modifications to that information. These factors together preclude efficient state-saving of modified information in optimistic simulations. Thus, if native stacks are used to implement process-oriented views in optimistic simulations, it becomes unavoidable to perform a brute-force blind copy of the whole native stack for state-saving, resulting in large and expensive state-saving costs for every process context-switch. This has been the fundamental reason for the perception that process-orientation is expensive to support under optimistic parallel simulations. It is now clear that with a minimal intermediate translator, which maintains jump indices and local variables, the state-saving costs can be reduced to the minimum required.

This approach also allows for lazy and tight memory allocation, which is important when simulating a very large number of processes. In our implementation, we perform lazy allocation of memory — memory for the frame is not allocated until the moment the frame is actually required, thus resulting in tight memory utilization, as opposed to preset stack limits of native stacks.

Another advantage of the stack reconstruction approach is its ability to support efficient run-time migration of active processes across heterogeneous platforms. Since machine-independent formats are used to represent the logical stack of procedure calls, it is both easy and efficient to pack a process stack, and move and restore it on the destination machine, even if the source and the destination machines use incompatible native stack representations. A further advantage is that the stack reconstruction implementation is completely portable, independent of the operating system or the native language compiler formats. It also has the desirable feature of not interfering with other compiler optimizations (such as tail recursion and register allocation).

3.3.3 Related Schemes

Other parallel simulation systems do implement variations on the scheme of using `goto`s (Maisie's code generator, IMPORT's macros, Cilk's frame-allocator), but do not allow nested procedure calls. Recent work on fault-tolerance systems [56] also utilizes another variant of this scheme to "walk up and down the stack" for portably checkpointing the C-runtime stack.

3.4 Performance

The stack reconstruction technique has been incorporated into the implementation of true process-oriented views in the TeD modeling and simulation system[50]. To test its capabilities and run-time performance, three different scenarios were used: (1) to measure its process context switching costs in pure process-oriented models (2) to compare its performance against event-oriented models (3) to stress-test the approach with respect to size, using models containing over a million processes.

3.4.1 Context Switching Cost

The *active* depth of a stack is the depth of the stack (number of procedures on the stack) at the moment the process is suspended. When the stack reconstruction approach is used, it is clear that the cost of suspending or restoring a process is a function of the active depth of the restored (or suspended) stack, since the active functions are re-invoked (or unrolled) to restore (or free up) the stack. Figure 3 shows the variation of the context switching cost with the active depth of the stack, measured using a synthetic model containing processes having exactly same active depth of stack. The switching cost includes the costs of suspending a process and resuming another process, and also the cost of building and maintaining runtime information to reconstruct the stack of process. The benchmarks were run on an SGI Origin multiprocessor with `R10000` processors.

From the figure, it is seen that the context switching takes less than $10\mu s$ for processes with relatively small active depths of stack. In our experience with modeling large and complex telecommunication network protocols, the processes in our models

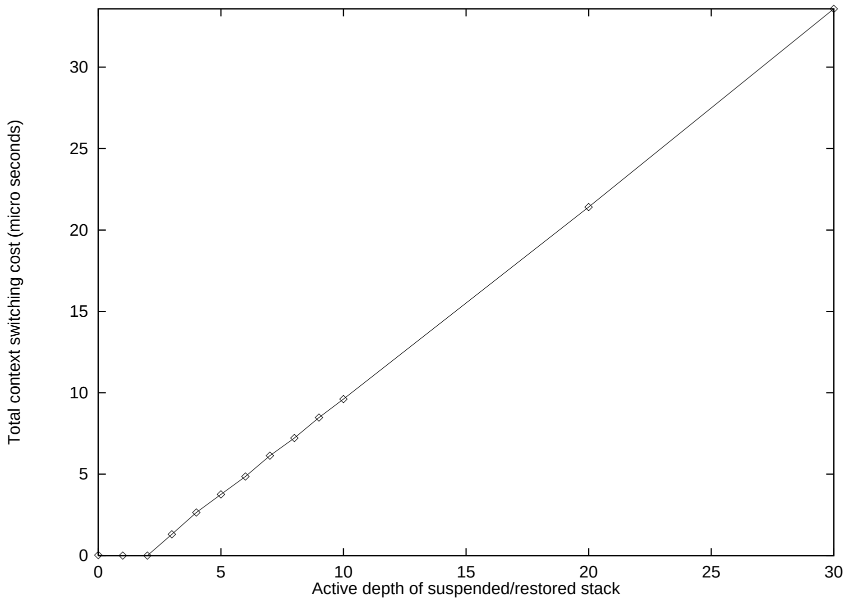
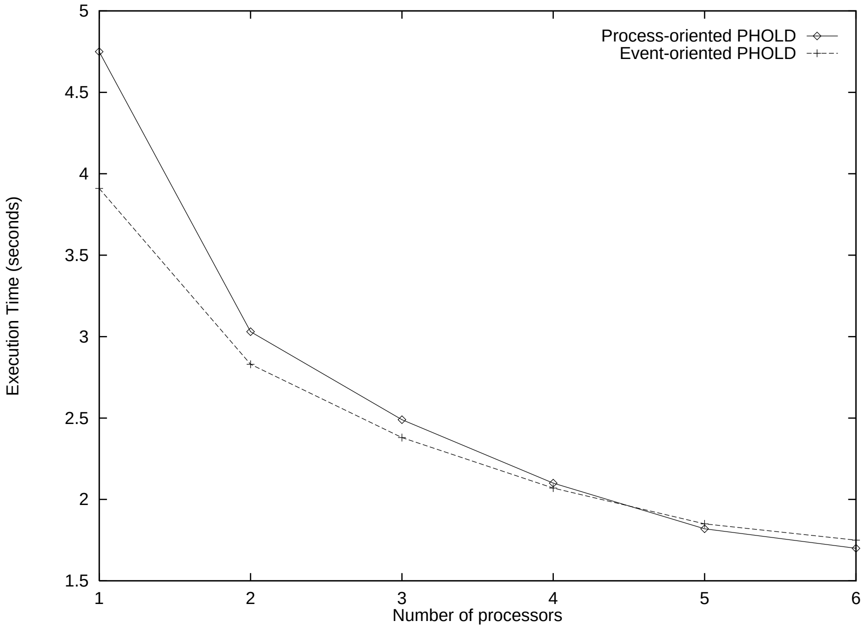


Figure 18: Variation of Process Context-switching Cost with Active Depth of Stack

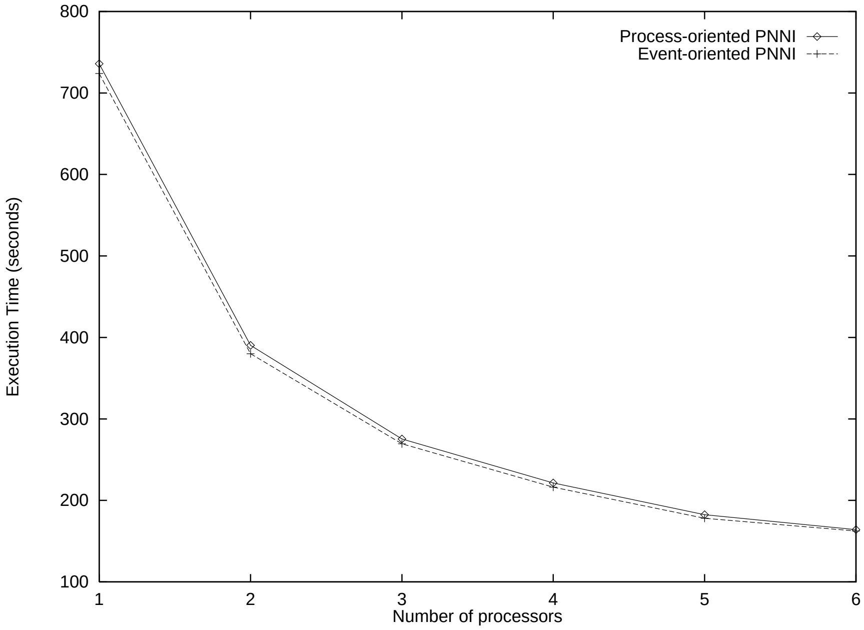
never exceeded an active stack depth of 5, giving a context switching time of less than $4\mu s$. Figure 3 also serves as a means of locating the tradeoff points when the stack reconstruction approach performs better or worse than other alternative approaches, such as using off-the-shelf thread packages.

3.4.2 Comparison with Event Orientation

On certain simpler types of process-oriented models, the run-time performance of our implementation achieves the superior performance of equivalent event-oriented models. This is demonstrated in figures 4 (a) and (b), which plot the elapsed time of our process-oriented implementation versus that of equivalent event-oriented expression of the same models. One synthetic and one real-life model were used in this scenario. The figures correspond to the optimistic parallel simulation of the models, using Time



(a) PHOLD



(b) PNNI

Figure 19: Performance of Stack Reconstruction Method on Process-oriented Models as compared to Equivalent Event-oriented Models

Warp. The synthetic model is the well-known PHOLD application containing processes that exchange events in a way that conserves the total number of events in the model. The second model is the PNNI telecommunication network application [51] containing network models capturing the ATM Forum's standard on internetwork protocols called the PNNI (Private Network to Network Interface). Both the models share the feature that the simulation advance primitives (**wait** statements) appear only at the outer-most level in the process body. Since such processes can be easily translated into efficient event-oriented models without any process-orientation overheads, one can reasonably expect to simulate such *quasi* processes faster than pure processes. The performance results shown in figure 4 demonstrate that indeed our process-oriented view implementation achieves almost the same speed as the equivalent event-oriented translation of these models. On the multi-processor runs, the rollback behavior of the PNNI models remained the same, thus making the results directly comparable. On 5 and 6 processors, the PHOLD model experienced 3% and 4% fewer rollbacks respectively on the process-oriented translation as compared to the event-oriented translation, making the process-oriented translation in fact marginally better than the event-oriented translation. This rollback behavior has been observed to be invariant across repeated simulation runs.

3.4.3 Large Scale Process Orientation

Using the stack reconstruction implementation, it was possible to perform optimistic parallel simulations of models containing processes in excess of one million. A simple model was used, written in the TED language, of a wireless Personal Communication Services (PCS) network, in which each sector in the PCS network is modeled as an entity, while each mobile inside a sector is modeled as a process that accesses the state of the sector entity. Sample network configurations of a square grid were used, containing 225 sectors along each side of the grid, giving a total of 50,625 sectors. Each sector contains 20 mobile processes, giving a total of more than a million processes. The mobile behavior is expressed as a random walk over the sector grid, superimposed by a call initiation sequence modeled using inter-call generation times and call holding times. This demonstrates that large-scale optimistic parallel simulation of process-oriented models is indeed feasible to implement efficiently.

3.5 Remarks

Based on the results of our stack reconstruction approach, it is observed that process-oriented views can be implemented in optimistic parallel simulations as efficiently as in conservative parallel simulations. This is possible by carefully reducing the state-saving operations to the absolute minimum. The stack reconstruction approach facilitates the capture the modifications to the process-state in order to transparently support incremental state-saving. It also results in very efficient process context switches for processes with relatively small *active* stack depths. In addition, it brings with it the added benefits of reduced memory overheads, portability, and process-migratability across heterogeneous platforms. This approach can be easily implemented using a preprocessor or can be incorporated into existing simulation languages.

In models containing processes with large active stack depths, the tradeoff points need to be explored at which other alternative implementations perform better than the stack reconstruction approach.

Chapter 4

PNNI: A REFERENCE APPLICATION

The application of parallel simulation technology to any complex simulation problem is a non-trivial undertaking. Numerous factors contribute to the difficulty, some of which are not known in advance until they are encountered in an actual implementation exercise. With a view to identifying such sources of difficulty in the application of optimistic parallel simulation methods to real-life models, an actual model development and simulation exercise of non-trivial size was performed. The target application chosen for this is the ATM (Asynchronous Transfer Mode) Forum's standard protocol suite called the PNNI (Private Network to Network Interface). This application is relatively complex and hence well-suited for the aforementioned purposes. The successful completion of the modeling and simulation of the ATM PNNI *protocol suite, together with demonstration of encouraging simulation speedup, makes this an important reference effort. The success indicates that the parallel simulation technology has matured and it is feasible to put parallel simulation to use in real-life, real-size applications such as network models.*

Parallel simulation methods have been studied for the past decade, and the theory behind the methods is well understood. However, the theory has not been adequately matched with practice so far. Relatively few successful demonstrations of the technology were made that apply the parallel simulation theory to real-life applications such as large-scale telecommunication network simulation. To serve as a reference effort in this context, a detailed implementation of a relatively complex network model was carried out. The application chosen for this specific purpose was the ATM PNNI protocol analysis. As will be shown later, extra care and effort was needed over and above the known methodologies, in order to finally achieve the desired simulation performance for this application.

The Private Network-Network Interface (PNNI) protocol suite[2] is an international draft standard proposed by the ATM Forum. The protocol defines a single interface for use between the switches in a private network of ATM (asynchronous transfer mode) switches, as well as between groups of private ATM networks. Corresponding to the network and transport layers of the OSI Reference Model, PNNI supports two categories of protocols: (1) topology discovery and (re)configuration, and (2) dynamic routing via virtual circuit connections. The main feature of the PNNI protocols is *scalability*, i.e. the complexity of routing does not increase drastically as the size of the network increases. There is an inherent trade-off between scalability versus quality of routing. Given the complex nature of broadband data traffic, simulations are the only feasible avenue to study the scalability of the PNNI protocol and to design and tune strategies for hierarchical addressing, topology aggregation and call admissions.

A virtual PNNI testbed has been built for the twin purposes of large-scale simulation studies of PNNI networks, as well as for exposing the problems associated with efficiently simulating the testbed in parallel. Two critical factors in this exercise were *performance* and *reusability*. For the network sizes and time scales of interest, the simulations must run on parallel and distributed platforms. Equally important is that the model descriptions be reusable and independent of the simulation-engine code. This is because the value of the testbed to the user lies in the model descriptions, not the simulation engine. The testbed is expected to be reused to study higher-level protocols as well as scenarios in the future. Moreover, it is desirable that the model not be tied to a single simulation engine — it is preferable to leverage advances in parallel discrete-event simulation technology without having to rewrite the testbed.

The testbed uses the new network description language TED [49, 50]. TED is designed with the goals of reuse, transparent modeling, and efficient parallel execution. The **C**$^{++}$ version of the TED software system was used that utilizes the GTW (Georgia Tech Time Warp) library as the underlying parallel simulation engine. The simulations run on shared memory multiprocessors.

The modeling and simulation aspects of the testbed are documented here. Specifically, the focus is on the issues that arise in (1) separating model code from simulation specifics, and (2) organizing the model for parallel simulation, without interfering with transparency.

The remainder of this chapter is organized as follows. Section 4.1 presents an overview of the TED language and the PNNI protocols, and describes the challenges of building transparent PNNI models. Section 4.2 describes the TED design of the PNNI testbed, while Section 4.3 describes some of the implications of the testbed on the simulation engine. Finally, section 4.5 describes a simulation study and presents results of parallel simulations.

4.1 Transparent PNNI Models

TeD (Telecommunications Description) is a language for developing transparent model descriptions which are (i) not tied to any specific simulation engine, and (ii) not cluttered with references to simulation code. While TeD was conceived for telecommunication network models, its basic constructs permit more general use.

Besides developing a virtual PNNI testbed, the modeling experience also served as one of the first complex tests of the TeD modeling and simulation software system. The feedback from the experiences resulted in some significant improvements to the simulation software. The end result is that the PNNI models are truly transparent and the performance of the simulations is encouraging.

The remainder of this section gives high-level overviews of TeD, PNNI, and the challenges of developing transparent models.

4.1.1 TeD Overview

TeD supports a parsimonious set of concepts: *event*, *channel*, *entity*, *architecture*, *process* and *component*. Entities, the basic building block for TeD models, are object-oriented encapsulations. These abstractions are in the spirit as the VHDL language[8] for hardware description.

An entity specifies the abstract interface of a network element or protocol. Entity interfaces are defined in terms of channels through which events flow; mappings among channels interconnect entities. The behavior of an entity is specified by its architecture which consists of processes and components (sub-entities). Inheritance between entities and architectures allow entity (architecture) types to inherit from other entity (architecture) types. Entity and architecture type definitions can be compiled to create a model database. Instantiations and compositions among database objects establish desired network configurations. Models can be customized via parameters that are assigned user-defined values after compilation.

TeD constructs facilitate the development of transparent network models. At the same time, the disciplined (and restricted) ways to compose entities makes it possible to compile models for parallel execution. By enforcing the discipline for composition within the language itself, the dual concerns of model specification and execution are effectively separated. This separation allows the development of complex models which can exploit state-of-the-art technologies for parallel and distributed simulation. A sample code fragment from the PNNI models is illustrated in figure 20.

4.1.2 PNNI Overview

The PNNI protocol is a complex set of ATM routing and signaling mechanisms proposed in the draft specifications v1.0 [2]. It represents one of the most sophisticated signaling protocols devised to date[19], aimed at supporting QoS-based routing along with unprecedented levels of scalability.

The PNNI specifications are based on a hierarchical addressing scheme which is used for maintaining network topology information and for call routing. In a global network it is infeasible for each node to contain a complete description of the entire network — the costs of storing and updating this information are prohibitive. On the other hand, a hierarchical scheme allows nodes to have reduced "views" of the network; thus, changes to the network need not be propagated to every node.

```
process #3: PNNINode:TopoMain: hello_send
{\{
   for(int l=0; l < STATE(curr_pgl_level); l++)
   {
       CNodeAddress self;
       get_self_addr( &self, l );
       for(int i = 0; i < MAX_DEGREE; i++)
       {
           int vcc = DCONST(FIRST_RCC_SVC) + l*MAX_DEGREE+i;
           if(is_established(vcc))
               CHANNEL(vcc_out[vcc]) << EVENT(Hello,(&self));
        }
   }
   double jitt_intvl = DCONST(hello_interval) *
                  (0.75 + 0.5*STATE(hello_jitter_rng).next());
 \}
 wait for $jitt_intvl$;
}
```

Figure 20: A TED Process in the PNNI Node Model

The PNNI hierarchy organizes network nodes (at level-0) into level-1 peer groups; level-1 groups are grouped together into level-2 peer groups, and so on. Figure 21 gives an example of a PNNI hierarchy with 4 levels. The level-0 nodes have addresses of length 3, level-1 groups have addresses of length 2, and so on (there are no constraints on the *number* of nodes in each peer group). The entire network is a level-4 group (with an empty address).

Within every peer group, one member is elected as the peer group leader (PGL). Peer group leaders play a prominent role in establishing and maintaining the topology views for network nodes; they are not used for call routing. The role of each PGL is played by a network node, which we call the role playing physical node (RPPN). The RPPN of a PGL can be determined by recursively following PGLs down the hierarchy

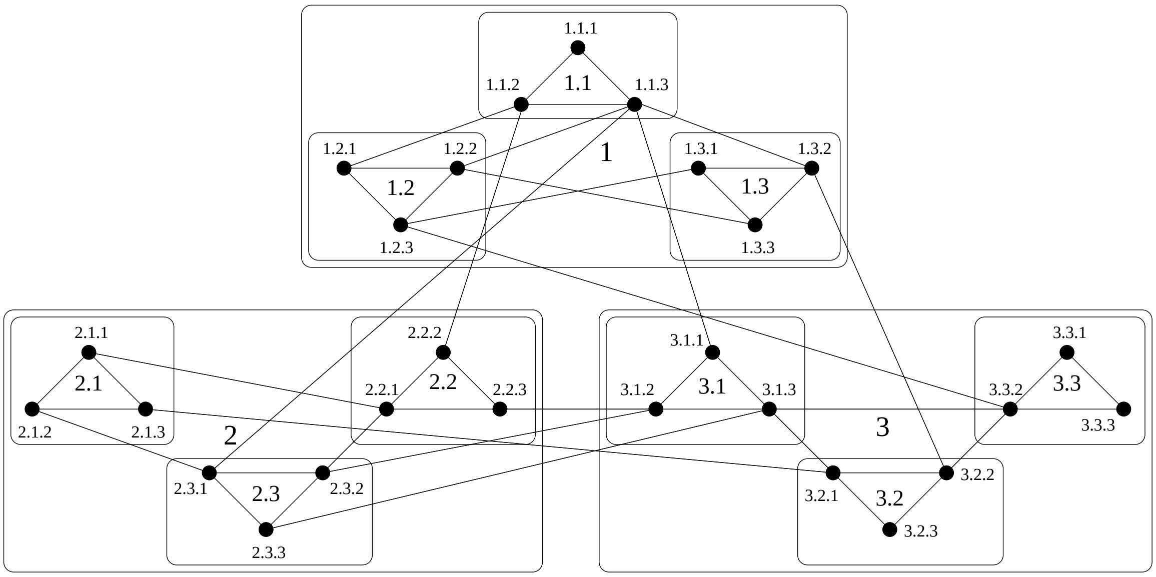


Figure 21: An Example PNNI Network

to a network node. All interactions conceptually within the hierarchy are carried out among the RPPN nodes in the network.

According to the PNNI specifications, each network node must have complete knowledge of the nodes and edges within its level-1 peer group. This is achieved by having each node flood its information within its level-1 peer group. For higher levels, $\ell > 0$, level-ℓ RPPNs of PGLs that are part of the same level-$(\ell+1)$ peer group exchange aggregated information via routing control channels (RCCs) established by the call setup mechanisms.

Finally, the aggregated information received by the RPPN of a level-ℓ PGL is sent to the network nodes below it in the hierarchy. The exchange of topology information is carried out using short `Hello` messages and longer `PTSE` messages. The topology information that is gathered and maintained continually at each node is used in the distributed routing protocols for call admission, call setup and call teardown.

This information is periodically updated during network operation. The effect of the hierarchy is that each node has detailed information about nearby nodes but only approximate information about more distant nodes.

4.1.3 Modeling and Simulation Challenges

The PNNI protocols are challenging, both for modeling and efficient parallel execution. Section 4.3 describes how the following challenges were met within the TeD – GTW system.

Recursive layer dependencies. The conventional "layered" approach to modeling network element behavior is inadequate to express the complex behavior of the PNNI nodes. For example, establishing the hierarchy requires that RCCs be set up; this requires call routing and admission which, in turn, depends on the hierarchy being set up. This requires that the modeling framework support recursive dependencies.

Complex model behavior. PNNI node behavior involves complex timing, ordering and synchronization of protocol messages. The informal description of the protocol functions in the PNNI draft specification illustrates this complexity, with several timers, decisions and assumptions. The challenge was to formulate these interactions within the TeD discipline, compromising neither the transparency of the TeD model nor the efficiency of the parallel execution.

Complex and large states. PNNI nodes contain large amounts of state information that change in unpredictable ways. The state information itself is complex, and requires modeling support for transparent processing. The large state size requires incremental state saving techniques in the context of optimistic parallel simulation. Indeed, a transparent incremental state saving facility had to be incorporated within the Georgia Time Warp (GTW) system.

Complex communication patterns. In typical PNNI networks, there are several classes of interaction for configuration discovery, topology exchange, call setup, data transfer, and so on. Moreover, the interaction of a node with its neighbors varies with the type of the neighbor and the class of the interaction. The communication patterns are complex and unpredictable. To further aggravate the situation, typical networks (such as the one used in our study) are difficult to partition; this creates hurdles for load balancing for efficient parallel simulation.

Variability in event size and processing. Event sizes vary considerably with type. For example, `Hello` packets are considerably smaller than `PTSE` packets. This variation is important because parallel simulators typically fix event sizes for efficient memory management; forcing all events be the size of the largest event lowers performance because of the overhead of memory copying. Similarly, the computation performed by a PNNI node on receiving a message (event processing) varies with the type of the message. It is important that load balancing mechanisms for parallel simulators not assume uniform costs for event processing.

Complexity of development process. In developing complex simulation models, facilities for automatic checks of model correctness are indispensable. For example, "assertions" are a common technique by which pre-conditions and post-conditions are specified in the models, which are automatically verified at run-time by the simulator, and the simulation halted upon the detection of the first assertion violation. In the context of optimistic parallel simulations, however, *transient* erroneous conditions may be caused due to optimistic processing, in addition to *non-transient* modeling errors. Generalized correctness checking techniques in the modeling systems are thus needed.

4.2 PNNI Models in TED

Each PNNI node is modeled as a TeD entity. The entity interface, shown in Figure 22, consists of (i) a parameterized array of channels, one for each ATM link to a neighboring node, and (ii) a `user` channel for the local point of attachment of a user node. The channel types in the array correspond to messages in the PNNI protocol between PNNI nodes. The user channel type includes messages that deal with network services in terms of switched virtual circuit (SVC) setup and teardown. An arbitrary PNNI network configuration can be built by instantiating one entity per network node and connecting channels appropriately.

```
entity PNNINode( int NUM_LINKS )
{
    channels
    {
        inout ATMLink physical[
           $PARAM(NUM_LINKS)$ ];
        inout UserChannel user;
    }
}
```

Figure 22: TeD Entity Interface of a PNNI Node

As outlined in Section 4.1, each node participates in establishing the PNNI hierarchy and possibly acting as the PGL for multiple levels within the hierarchy, in periodic exchange of topology information, and in call admission and routing. The corresponding behavioral specification of a PNNI node in TED partitions nicely into the linear inheritance hierarchy shown in Figure 23.a. Figure 23.b shows the components of the PNNI node architecture and the inheritance among these components.

When an event appears on one of the channels in the array, it is processed by the `SVCService` architecture. The `SVCService` architecture models link-to-SVC and SVC-to-link mapping services, so that messages are identified by SVC name rather than link identifiers. Depending on its type, the received message is forwarded to one of three architectures: `TopologyMaintenance`, `RoutingService`, or `UserService`.

The `TopologyMaintenance` architecture inherits from the `SVCService` architecture and models the PNNI topology maintenance protocols using the SVC service. It models PGL instantiation and PGL communications, and builds and maintains the topology database over time for the node as well as for every PGL for which the node is an RPPN.

A set of internal channels model the SVCs internal to the node. The array of internal SVC channels are logically divided into several classes. The first class (RCC) of SVCs is used for communication between the nodes at all levels of the peer group hierarchy. The next class (SIG) is used to exchange signaling messages. The next class (PATH) is used as internal SVCs to serve as connectors at intermediate nodes along a path of any logical SVC connection. The last class (LOCAL) is used for user-level end-point communication that serves as origin or destination points for SVCs requested by a user node. The first `N` SVCs (where `N` is the number of neighbors) of the

RCC class and SIG class are dedicated SVCs that are mapped using bootstrapping information.

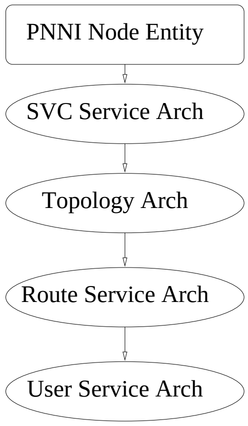
The `RoutingService` architecture inherits from the `TopologyMaintenance` architecture, thus inheriting the topology database maintenance behavior. It adds the call admission and routing algorithms to service SVC setup requests coming from the neighboring nodes, or from user nodes, or from the `TopologyMaintenance` processes themselves.

The `UserService` architecture inherits from the `RoutingService` architecture, thus inheriting the entire PNNI protocol functionality for a node. To the functionality, it adds the SVC setup, teardown, and data transfer services for user nodes.

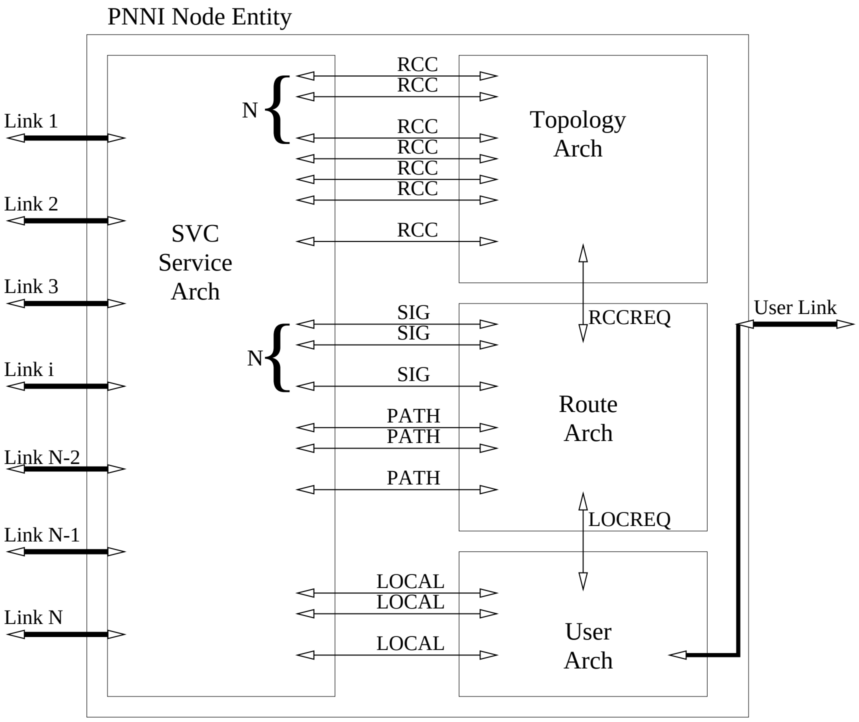
4.3 Experiences

In this section, solution approaches are presented to the challenges outlined in Section 4.1.

Recursive Layer Dependencies. The mutually recursive dependence between topology maintenance and routing service is resolved by using an internal channel (RCCREQ) in the `TopologyMaintenance` architecture (see Figure 23.b). This channel is used by the `TopologyMaintenance` architecture to forward SVC setup requests. The inheriting architecture, `RoutingService`, caters to these requests coming on the internal channel. Similarly, the routing service architecture caters to SVC setup requests originating from the `UserService` architecture that arrive on another internal channel (LOCREQ) dedicated for that purpose.



a. Entity and Architecture Inheritance Hierarchy



b. Internal Compositional View of Architectures

Figure 23: Illustration of PNNI Node Model Organization in TeD

Section 4.1 mentioned the recursive dependence between topology database maintenance and signaling in the PNNI protocol. For *auto-configurability,* the dynamic formation of the peer group hierarchy requires the dynamic set up of SVCs between PGLs. But, at the same time, the signaling support that is required for setting up the SVCs depends on the formation of the peer group hierarchy in the first place.

Two facts underlie this recursion: (1) the SVCs between neighboring nodes at level-0 are permanent virtual circuits that are set up at boot time, and (2) the formation of the hierarchy up to level i is necessary and sufficient for the establishment of SVCs between PGLs at level $i+1$. The first fact is easily modeled by proper initialization of node parameters. The second is used during the hierarchy formation. We have addressed the problem of modeling this recursion using a combination of two solutions.

First, the format of the SVC setup requests was generalized by not requiring that the destination of an SVC setup request be a complete ATM address. When the destination address is specified as a partial address, it is instead required that it correspond to a peer group address which is then interpreted to be the RPPN corresponding to the PGL of that peer group. (A full destination address trivially corresponds to the level-0 PGL of the destination peer group, which is the same as the RPPN address of the destination node.) This generalized specification of SVC destination enabled us to develop a single model for routing services which services both user-initiated requests (which specify full destination addresses) as well as internally generated requests (which specify partial destination addresses for RCCs between PGLs) for SVC setup.

Second, the recursive interaction between topology maintenance architecture was modeled with the generalized routing service architecture. This was done using TeD's *internal channel* feature that allows, among other things, for an *inherited* architecture's processes to send events on an internal channel to be processed by the processes of its *inheriting* architecture. Thus, an internal channel, `RCCREQ`, is used for such communication between `TopologyMaintenance` architecture and `RouteService` architecture. Whenever a PGL at level $i+1$ is elected in a group at level i, the current topology database at the RPPN corresponding to the PGL contains sufficient information to discover the (partial) addresses of the new PGL's neighbors. SVCs must be setup between this PGL and its neighbors. For this, SVC setup requests are forwarded by the `TopologyMaintenance` architecture down the `RCCREQ` internal channel, which are received and serviced by the `RouteService` architecture. For its part, the routing service makes use of the current topology database maintained by the `TopologyMaintenance` architecture (the database is directly accessible because of inheritance). As mentioned previously, the database contains just sufficient information at that point in time to service the current SVC requests to the destination PGLs. Thus, dependency of the `TopologyMaintenance` architecture on the routing service is resolved using an internal channel, whereas the dependency of the `RouteService` architecture on the availability of the appropriate topology database information is resolved by means of access to such information through inheritance.

Complex Model Behavior The complexity of PNNI protocols appeared to demand richness and power from the modeling language, such as arbitrarily nested synchronization points, and timer services. In the simulation community, such expressive power is generally recognized as corresponding to *process-oriented* simulations.

Furthermore, it is generally accepted that *event-oriented* simulations, which hold lesser expressive potential, can be more efficiently implemented than process-oriented simulations. TeD (as defined in the current language specification) attempts to strike a balance between the expressive power and efficiency of execution associated respectively to the two simulation approaches by supporting *quasi processes*. These are processes in which simulation-time can be advanced only in the "main body" of the process. In fact, in the version of TeD used for PNNI modeling, the restriction was more stringent in that *wait* statements are not allowed to appear under conditional and looping statements, but rather appear only as the "top-level" statements in the processes. The TeD compiler is capable of translating the quasi-process descriptions into event-oriented simulations with zero stack overheads, thus achieving the associated efficiency.

The challenge was to specify the PNNI protocols under the quasi-process model supported by TeD. Surprisingly, this turned out to be quite natural, and the protocols were modeled in an elegant manner despite the restrictions. This instills confidence that tradeoffs between expressive power and efficiency can be settled well in many applications. The PNNI model suite serves as an example model which does not demand the full expressive power of true process-oriented views. However, for other more demanding models, TeD provides a newer, more powerful, process implemenation (with support for wait statements inside conditional, looping and sub-routine constructs). The efficient implementation of full-blown processes is achieved using the stack reconstruction approach as described in chapter 3.

Large and Complex State Each PNNI node contains a large amount of state information — Topology Database for potentially hundreds of nodes, SVC mapping information for (potentially thousands of) SVC's, link state information for (potentially dozens of) ATM links, and so on. As an example, consider the 200 node network depicted in figure 32. In this network, each node maintains a topology database of 100 entries, in which each entry is 4 kilo bytes in size, giving a total of 4 mega bytes of state for topology maintenance alone. Moreover, each entry is a composite of several nested **C**$^{++}$ classes (many classes, in fact, being defined as template instances).

In response to this need, the Transparent Incremental State Saving (TISS) facilities of TED were developed for managing such large state in a transparent manner. As was witnessed, this makes it possible to develop TED models without any knowledge of the underlying TISS method or GTW simulation engine. In fact, one of the authors that contributed to parts of the PNNI models was completely ignorant of the intricacies of the underlying incremental state saving techniques, and also of the actual optimistic simulation techniques in general!

In essence, object-oriented class definitions were used that perform simulation-specific functions, but in a transparent manner. For example, `NodeAddress` is a class that provides the functionality to manage node addresses. Internally, it is modeled as an array of `INT`, each element of which is a transparent implementation of an incrementally state-saved object. Due to the overloaded assignment operators predefined by TED over `INT` types, no additional simulation-specific primitives appear in the model code, resulting in transparent model code that is oblivious to implementation details.

Transparent Incremental State Saving (TISS) The approach of using overloaded operators for TISS, as originally proposed in [59], included an important restriction. In their solutions, classes whose operators were overloaded for TISS were not meant to be used to instantiate variables on the stack. This was "justified" by their assumption that classes used for state variables (which are typically allocated on the heap) will be different from those used for temporary variables (those allocated on the stack). However, in the PNNI models, the same class definitions had to be used in many situations – for state variables, event variables, as well as for local variables inside functions, and as function argument types. For example, the `NodeAddress` class is used extensively throughout the model code in all contexts.

This creates the following problem: When an overloaded assignment operator is invoked on an object during the simulation, the object needs to decide whether it must (incrementally) save its state before this impending modification to its member variables. If the object is a temporary stack-based variable (either a local variable, or an argument to a function, or a variable automatically introduced by the compiler for its various intermediate purposes), then the state should not be saved. Otherwise, the object's state must be saved. The problem thus reduces to finding ways in which the object can detect at runtime whether it truly belongs to the state, or if it is only a temporary object, modifications to which can be safely ignored.

Several solutions are possible to address this problem. One approach is to include a boolean `flag` variable in each overloaded class using which the class remembers whether it was instantiated as part of an LP's state. This is easily implemented by setting a global variable to true during simulation initialization phase, and turning it off outside that phase. The constructor for the overloaded class can be written to check this global variable, and remember the boolean value in its boolean `flag` member variable. Every time a modification is detected, this `flag` variable is consulted to check whether this is a stack variable, and the state is saved if and only if `flag` is true. There are, however, two drawbacks to this approach. First, the `flag` boolean variable adds at least one byte of memory as overhead per overloaded class. Second, the constructor for *every* instance of the overloaded class will be invoked at runtime. This can impose significant runtime overhead if a large number of instances are created in the simulation. This approach was initially implemented in TeD, and both the drawbacks were quickly encountered in the case of PNNI models. This warranted looking for another efficient alternative, as described in the following second approach.

Since stack variables are by definition allocated on the stack, it is easy to detect whether a variable belongs to the stack or not by checking if its address falls within the bounds of the runtime stack! If the address does fall within the range of stack addresses, then state saving can be skipped for those variables. If not, then there exist two possibilities. Either the variable belongs to the heap, or it is a global variable. In either case, it needs to be state saved. This approach proved to be a sufficiently efficient and adequate solution, which was incorporated into TeD and used for the PNNI models.

Processor Mapping Since, in general, automatic load balancing is very hard, TeD provides facilities for user-defined mappings. Three different schemes were tried for mapping PNNI nodes to processors: a simple round-robin scheme, a second based on balancing node degrees, and a third which partitioned the PNNI hierarchy with few cross-edges. As expected, the third scheme performed the best. The first two schemes suffered considerable roll-backs with consequent poor processor utilization.

With the default round-robin assignment of entities to processor by TeD runtime system, the number of rollbacks was significant. Though round-robin assignment works well for symmetric networks, the PNNI networks are asymmetric (nodes communicate with neighbors within the same peer group more often than with other neighbors), and thus explained the poor efficiency of such a mapping. A more sophisticated mapping algorithm was implemented which used the peer-grouping information of the PNNI network to allocate nodes, so that nodes within the same peer group map to the same processor, thus considerably reducing the inter-processor communication. However, this was not adequate for networks with less number of groups than processors, or with roughly same number of groups as processors but with grossly unmatched group sizes. For these exceptions, another mapping algorithm was implemented that maps the nodes to processors based on a measure of their degree (number of unassigned neighbors). The algorithms are described in figures 24 and 25.

Group-based Heuristic Mapping:

- Find the unassigned group, max_g, with the largest number of nodes.
- Find the processor, min_p, with the least load (total number of nodes assigned to it).
- Assign all the nodes of group max_g to min_p. Mark max_g as assigned. Increment the load of min_p by the size of max_g.
- Repeat until all groups are assigned.

Figure 24: Group-based Node-to-processor Assignment Algorithm for PNNI Network Models

Degree-based Heuristic Mapping: Let $m = \lceil N/P \rceil$, where N is the number of nodes in the network, and P is the number of processors.

- Find the unassigned node, max_v, with maximum residual degree (largest number of *unassigned* neighbors).
- Find the processor, min_p, with the least load (total number of nodes assigned to it).
- Assign max_v to min_p. Mark max_v as assigned. Increment the load of min_p.
- Assign as many unassigned neighbors of max_v as possible to min_p such that the load of min_p does not exceed m. Increment the load of min_p by the number of such assignments. Mark such neighbors as assigned.
- Repeat until all nodes are assigned.

Figure 25: Degree-based Node-to-processor Assignment Algorithm for PNNI Network Models

In general, it was observed that the group-based algorithm is far superior to the degree-based algorithm, when the number of processors does not exceed the number of groups. This is expected since nodes communicate with neighbors that are members of the same peer group more often than with other neighbors. Also, the greater the number of groups, the greater the number of processors that can be used while still sustaining a good level of speedup. This leads one to believe that near-linear speedup is possible on larger PNNI networks with hundreds of groups.

Event Size Reduction. The effect of large size of some infrequent events is mitigated using data-compression techniques. The data of the class variables contained in the large events were compressed before sending the event, and uncompressed into class variables of the event upon receiving the event, just before processing it.

Compression factors between 2 and 3 were observed, thus reducing the maximum event size significantly, thereby improving the performance. It was found that the occasional compression/uncompression operations do not constitute any significant overheads compared to the average event processing granularity.

Unfortunately, the preceding technique does not constitute a completely satisfactory solution for simulating PNNI networks of very large size. This is because of the possibility that the `PTSE` and/or the `SVCSetupRequest` events can grow linearly with the size of the largest peer group, which implies that event data compression may not be sufficient. Other solutions are being explored, such as splitting the event into smaller-sized events. However, the most elegant solution appears to be one in which the underlying simulator supports variable-sized events, such that the large size of one infrequent event does not affect the performance of managing every single event during the simulation.

Problems with Optimistic Computation In optimistic parallel simulations such as Time Warp, it is often possible that an LP encounters a situation that it is impossible to have entered into if purely sequential simulation is performed. Of course, such situations are bound to be rolled back, unless there has been a programming error in the model. However, the possibility of such erroneous conditions increases the complexity of model development simply because it is no longer possible for the model to naively signal an exception whenever it runs into an error condition. This constraint conflicts with the important requirement in large scale model development in which it is almost necessary to perform appropriate sanitary checks on the data and computation, and signal to the programmer whenever an undesirable situation is encountered. This makes it necessary for the LP to act in a special way upon detecting erroneous conditions. This is an important problem because, in general, *the LP has no way of finding out whether an erroneous condition is due to parallel simulator's optimistic computing, or due to a programming error.*

In such cases the model must not halt the simulation on detecting an apparent erroneous condition, but rather do something more clever by which an error report is generated if and only if the error was caused due to a modeling mistake and not due to optimistic processing. Thus, the correctness checking techniques in the modeling systems must be suitably generalized to support simulation-independent specification, while at the same time cover the possibility of optimistic parallel simulation.

To solve the problem of resolving transient and non-transient errors during optimistic parallel simulations, TeD provides a special macro, `POT_ERR()` ("potential error"), that is used in the model to signal a potentially erroneous condition. If the error is a result of optimistic processing, the condition gets rolled back appropriately, and the macro will have null effect. However, if the error was indeed due to a modeling/programming error, an error report is generated by TeD upon the termination of the simulation.

In the process of developing the PNNI models it was found that an integrated correctness-checking facility is absolutely essential both to debug as well as to gain confidence in the correctness of the model execution when simulated *in parallel* using a risky and aggressive style of optimistic simulator (such as GTW).

4.4 Model Reusability

The PNNI node models are easily reused in other TeD applications. This is done by instantiating several PNNI nodes appropriately connected according to the underlying network graph (such a reusable "assembling network entity" is also provided), and by connecting a "user node" entity to each PNNI node.

From the entity interface given in Figure 22, it can be seen that any user node that has an `inout` channel of type `UserChannel` can become a user node to the PNNI node, just by connecting the corresponding channels.

4.5 Simulation Study

Three sample network configurations were used in the study: a 200 node network, a 600 node network and finally a 1000 node network. The 200 node network depicted in in figure 32 was provided by Iraj Sainee of Bellcore. The 600 and 1000 node networks were synthetically generated by Fang Hao of Georgia Tech using the Georgia Tech Internetwork Topology Model generation (GT-ITM) software[13]. Using a few heuristics, all the three networks were pre-processed to automatically partition and generate hierarchical PNNI ATM addresses and other protocol-specific configuration parameters for the network nodes.

The networks were configured to include a single user node attached to each network node. Each user node included a client that initiates SVC connections and a server that accepts SVC connections. Two types of client behavior were used: *uniform* and *localized*. In networks with *uniform* clients, the clients pick a SVC destination uniformly at random from the list of *all* servers. In networks with *localized* clients, the clients choose destinations that are closer to them in the PNNI hierarchy more often than those that are farther in the hierarchy.

In each simulation, a total of over 100,000 SVCs were established by all the clients combined. The number of failed SVC connection requests was negligible. A call connection can fail for two reasons: if the destination server is unreachable (the network is still "booting up"), or if no path with sufficient capacity is available. When a connection succeeds, the SVC has been setup, and data can be sent over that SVC. Each client uses an exponentially distributed intercall time with mean 1.0, and pareto-distributed hold time with mean 1.0 and α value of 1.2. At the end of a call hold time, the SVC is disconnected, which tears down the SVC across the network. (All constants described above can be customized easily on a per client/server basis.) Each successful SVC connection requires p instances of Dijkstra's shortest path computation, corresponding to the best-next-hop calculation at each intermediate node along a path, where p is the path length of the connection. This computation is in addition to other protocol processing.

The simulations were performed on an SGI Origin shared-memory multiprocessor system at Georgia Tech, containing R10000 processors and 4 Giga bytes of memory. For this application, the simulation on one processor achieved the same performance as that of a sequential simulation, since the overheads of parallel simulation on one processor were kept low.

The simulation speedup as the number of processors is increased is shown in figures 26, 27 and 28. The elapsed time for completing the execution of each simulation configuration is depicted in figures 29, 30 and 31. Satisfactory speedup is observed across all network configurations. For example, as figure 26 demonstrates, a speedup

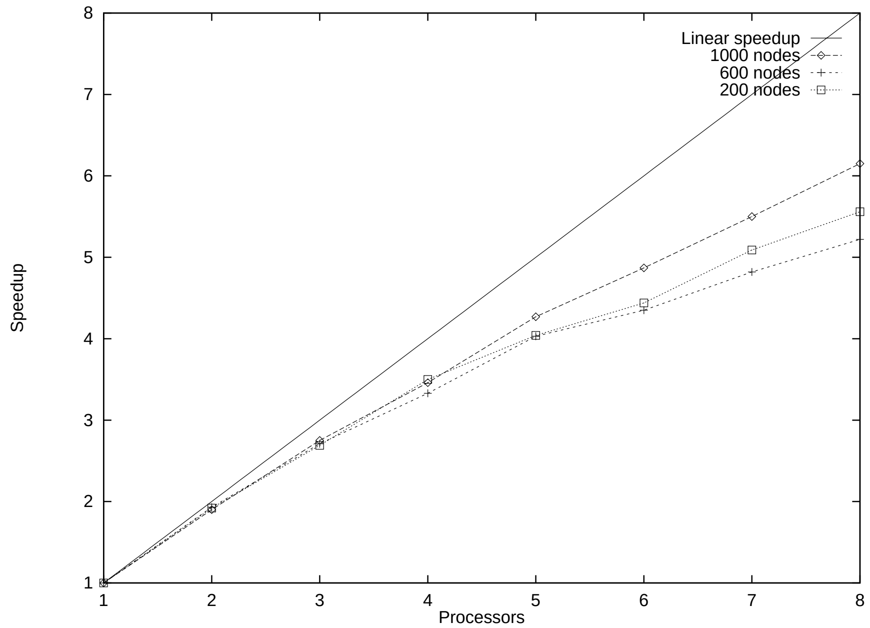


Figure 26: Simulation Speedup for PNNI Network Configurations with *Localized Communication* Clients

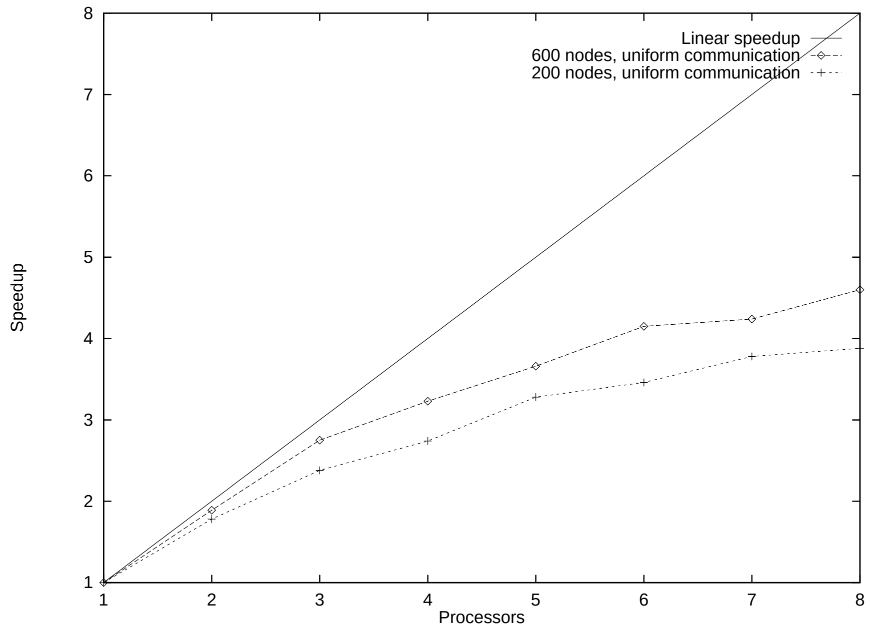


Figure 27: Simulation Speedup for PNNI Network Configurations with *Uniform Communication* Clients

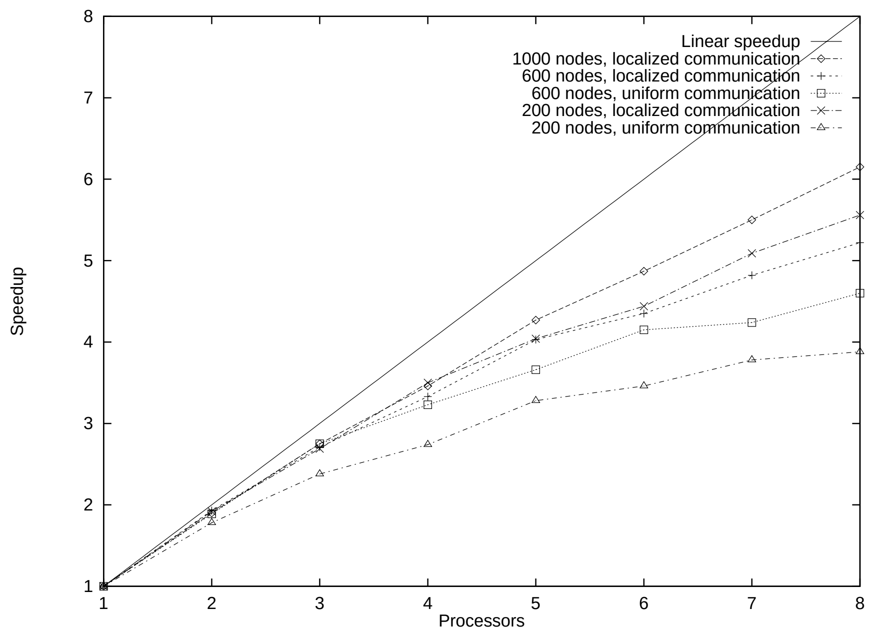


Figure 28: Simulation Speedup for all PNNI Network Configurations

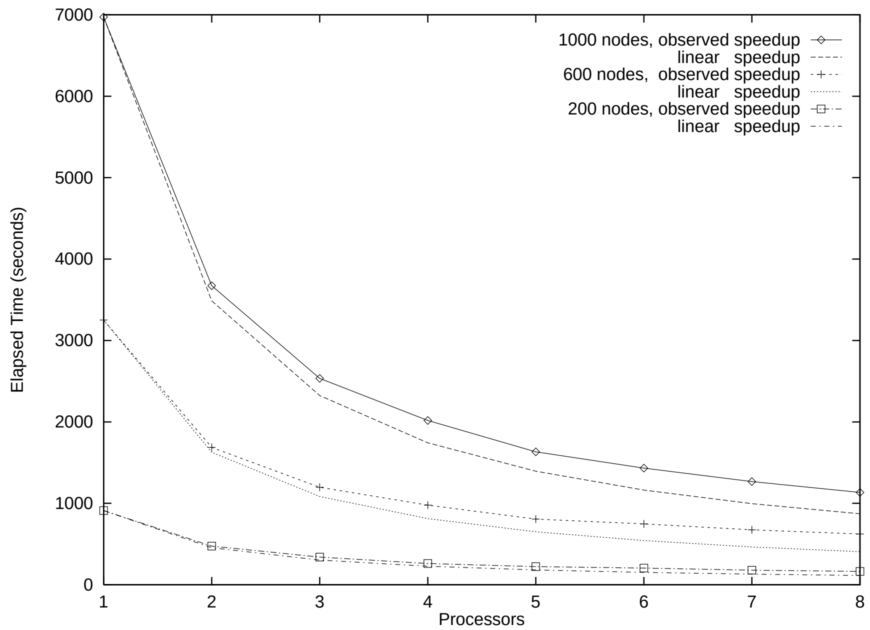


Figure 29: Elapsed Time for PNNI Network Configurations with *Localized Communication* Clients

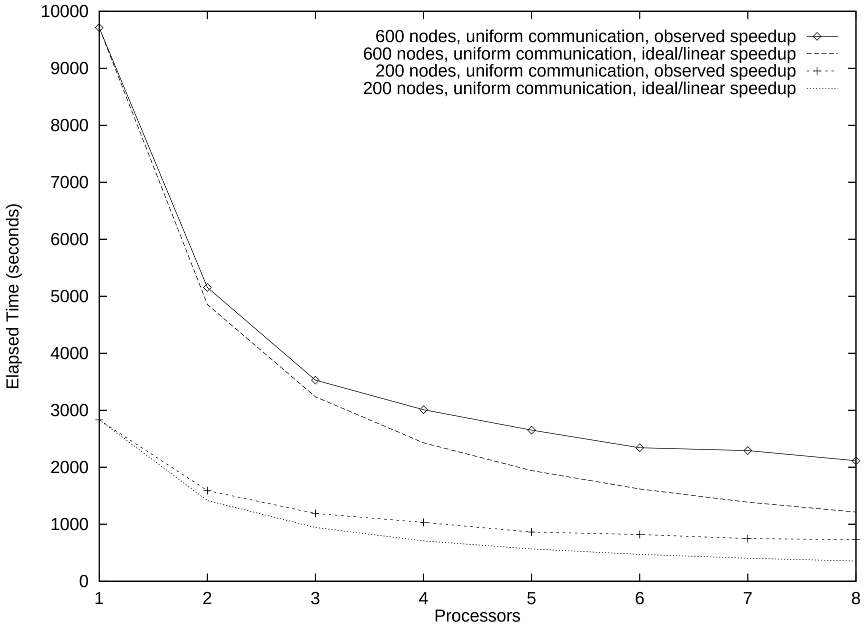


Figure 30: Elapsed Time for PNNI Network Configurations with *Uniform Communication* Clients

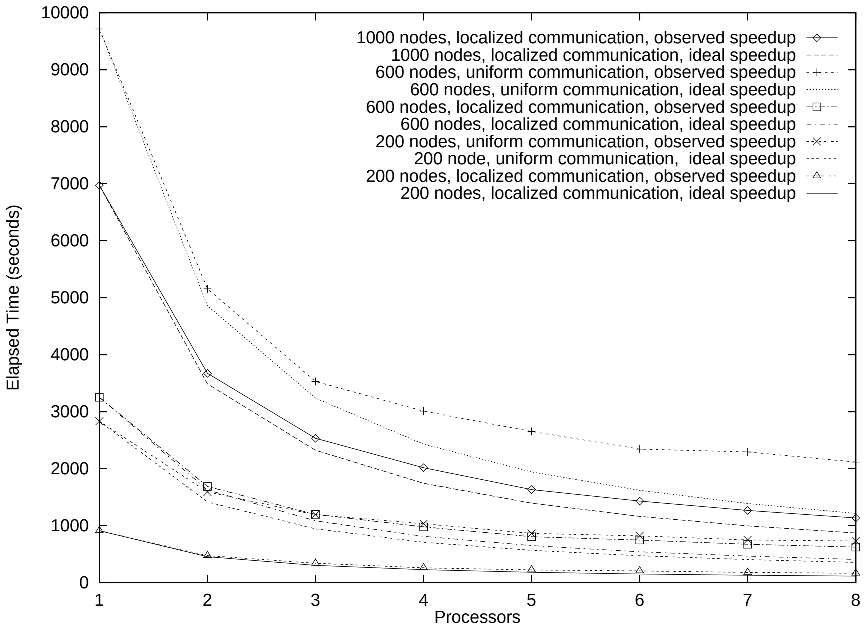


Figure 31: Elapsed Time for all PNNI Network Configurations

of 6.15 is obtained when the 1000-node network is simulated on 8 processors.

At the model-level, among the metrics of interest is the total number of SVC connection setups simulated. Figure 29 demonstrates that approximately 1,000,000 SVC connection setups can be simulated in approximately 3 hours of wall-clock time for a large (1000-node) PNNI network, and in less than 0.5 hours for smaller (200-node) networks.

As the number of processors increases, a rapid increase in the number of rollbacks is observed. It is also seen that, as the number of processors increases, the speedup is linear at first, and then flattens out. This is caused by the fact that when the number of PNNI nodes per processor goes below a threshold, rollbacks become frequent. This behavior was observed with smaller networks, when the performance flattened out at 3 processors, indicating that very large networks can be simulated with linear improvements with moderate numbers of processors.

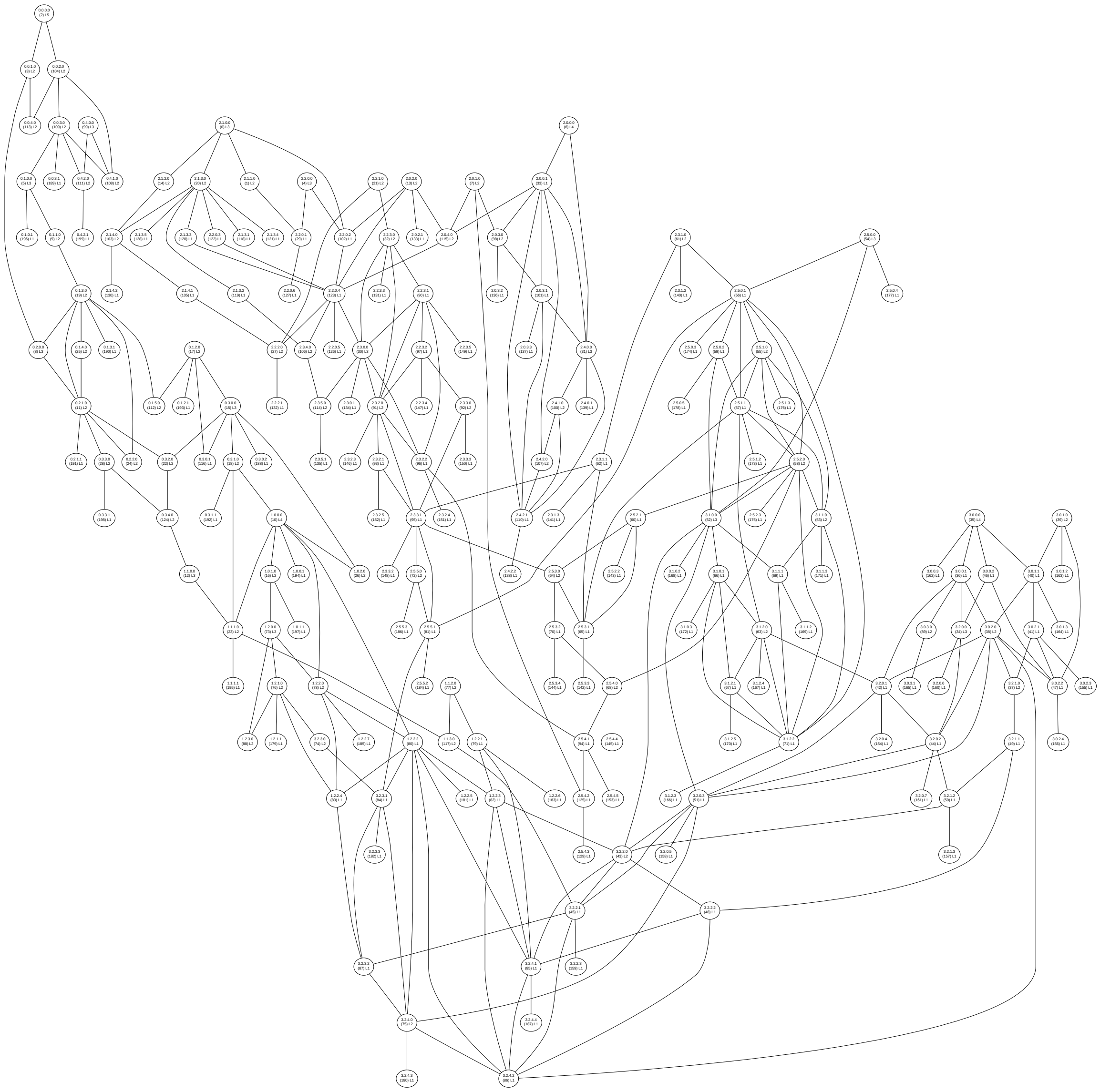


Figure 32: A 200 Node ATM PNNI Network used in Performance Study

Chapter 5

CONCLUSIONS AND FUTURE WORK

5.1 Summary

This thesis has been concerned with furthering the state-of-the-art in parallel simulation in general, and parallel network simulation in particular. System-level issues arising from the simulation of important real-life problems were addressed, and solutions to the issues are applied and tested in the context of large-scale telecommunication network models. Primary focus was on optimistic approaches to parallel simulation, although most of the techniques are applicable to other simulation approaches as well. In particular, the following research issues were addressed:

- Efficient optimistic parallel simulation of large *fine-grained* models.
- Efficient optimistic parallel simulation of large and complex *process-oriented* models.
- Feasibility demonstration of parallel simulation of complex, large-scale telecommunication network protocol models.

Previously, it was generally believed that optimistic approaches are not suitable for fine-grained applications, because of the high cost of traditional state saving approaches. The reverse computation work presented here served to show an efficient alternative to state saving, thereby opening up optimistic approaches to fine-grained models.

Similarly, with few alternatives to traditional stack copying techniques, parallel simulation researchers encountered difficulties in simulating process-oriented views optimistically with acceptable performance levels. In this thesis, it was shown that stack reconstruction can be used to optimistically simulate process-oriented models significantly faster than stack copying approaches.

5.1.1 Reverse Computation

It was shown that reverse computation can be used in place of state saving techniques to run fine-grained simulations significantly faster using significantly less memory. Contributions included the following:

- **Reverse Computation Performance:** This thesis work is the first to empirically compare the performance of reverse computation with that of state saving techniques, and demonstrate that reverse computation is not only a feasible approach but also an excellent match for high-performance simulations.

  In applications with predominantly constructive operations, the memory footprint of reverse computation can be significantly less than that for state saving, resulting in reduced memory and computation overheads, and more efficient use of the memory hierarchy.

- **Reverse Computation for Queueing Network Simulation:** This work is also the first to identify that reverse computation is more suitable than state saving for optimistic parallel simulation of queueing network models. This is due to the significant presence of constructive operations in queueing network models, making the reverse computation approach applicable to a wide range of application areas. Example areas include air traffic control and manufacturing simulations, all of which contain models of networks of queues.

- **Automation via Reverse Compiler:** To address the usability of the reverse computation approach, algorithms were designed to automatically generate the intermediate code necessary for supporting reverse computation. A reverse compiler was developed for the C language, and the compiler was applied to the applications. The automated version was shown to perform almost as efficiently as the hand-written reverse code.

  It was demonstrated that it is indeed possible to use reverse computation for complex models automatically without any significant performance penalty. This was achieved by actually building a compiler for reversing complex models, and by observing that the performance of the compiler-generated reverse code is close to that of efficient, hand-written reverse code, Important simulation-specific optimizations for reverse computation were identified and applied. The optimizations were essential for minimizing the overheads relative to a straight-forward automation.

- **Reverse Computation Implementation and Case Studies:** Support for reverse computation was incorporated into the Georgia Tech Time Warp (GTW) optimistic simulation kernel for shared memory multiprocessors, and its performance was evaluated on two telecommunication network applications.

  - **Case Study I – ATM Multiplexor Cascade:** A performance study on a fine-grained ATM Multiplexer model was completed. It was shown that reverse computation improved the parallel simulation performance by as much as four-fold as compared to state saving, with significantly reduced state memory utilization.
  - **Case Study II – PCS Network:** Similarly, a performance study on a medium-grained PCS (Personal Communications Services) network model was completed. Reverse computation was shown to improve the parallel simulation performance by as much as six-fold as compared to state saving, with significantly reduced state memory utilization.

- **Comparison to State Saving:** The performance of the reverse computation method was compared with that of state saving approaches. The performance study included a comparison of several variants of state saving, such as copy, periodic and incremental state saving. The performance study confirmed the significant benefits of reverse computation over *all* the above variants of state saving, on multiprocessor machines with cache-coherent non-uniform memory access (CC-NUMA) architecture. Reverse computation was observed to perform significantly better than state saving. Detailed analysis of the memory subsystem behavior confirmed that the performance gains are due to the beneficial effects of smaller footprint of reverse computation as compared to that of state saving.

5.1.2 Stack Reconstruction

The stack reconstruction approach was implemented and tested in the TeD language, and a performance study was completed which confirmed its low overhead in several network models. In fact, in some of the models, the stack reconstruction-based processes were observed to achieve almost the same high performance as that of equivalent event-oriented models.

In optimistic process-oriented simulations, state saving mechanisms are required for rollback, as well as for process context switching. The stack reconstruction approach avoids duplication of the state saving mechanisms, and integrates the two mechanisms into one, An added advantage of the stack reconstruction approach is that it supports the efficient runtime migration of processes across heterogeneous platforms.

Contributions included the following:

- **Process-orientation Performance:** The work presented here is the one of the first to empirically demonstrate the feasibility of efficiently executing process-oriented views in high-performance simulations. In applications with relatively small active stack sizes, the incremental nature of stack context switching results in significantly lower overheads compared to stack copying approaches.

- **Large-scale Process-oriented Views:** Since the memory requirement of stack reconstruction is less than that of the stack copying method, it is possible to simulate larger models with stack reconstruction than with the stack copying method. Due to this benefit, the ability to simulate a PCS network model containing over a million simulation processes was demonstrated.

- **Event-Orientation Performance for Simple Processes:** It was also demonstrated that certain simpler types of simulation processes (those that make limited use of the generality of process-oriented views) can be simulated with an efficiency that is close to that of equivalent event-oriented views. This is desirable because event-oriented views, in general, can be more efficiently executed than process-oriented views, but process-oriented views express model semantics better. The automatic realization of event-oriented performace coupled with support for full process-orientation makes it possible to use process-oriented models even in the place of event-oriented models, since the stack reconstruction approach extracts the same high efficiency from the simplified process-oriented models.

5.1.3 PNNI Models

Using the TeD language and simulation system, models of the ATM (Asynchronous Transfer Mode) Forum's standard protocol suite called the Private Network to Network Interface (PNNI) were developed. In addition to automatic and scalable topology discovery and routing protocols, the PNNI protocol suite includes the most sophisticated signaling protocols devised to date, with the goal of supporting quality-of-service (QoS)-based routing for global-scale ATM networks. In order to accommodate the high complexity of the models, it was required to develop and incorporate several support features in the TeD language and simulation backend. Using the modeling and simulation process of the ATM PNNI network protocols, important characteristics of complex, realistic telecommunication network models were identified.

- **Feasibility Demonstration and Reference Effort:** The design, development and verification of ATM PNNI models in the TeD language was completed. During the process, several important modeling and simulation issues were addressed and a variety of techniques was used. Finally, it was demonstrated that despite their complexity, such models can indeed be simulated in parallel, and with significant speedup, demonstrating the feasibility of parallel simulation on such models. Based on the solutions incorporated into the TeD language and simulation backend, other researchers have also since developed a range of other network models in the TeD language and demonstrated significant speedup via parallel simulation.

- **Size and Fidelity:** The TeD models of the ATM PNNI network protocol suite represent some of the largest, most complex and realistic telecommunication network models that have been successfully simulated in parallel to date, delivering significant speedup using optimistic parallel simulation. This is in contrast to previous parallel simulation work that focused on the parallel simulation of relatively simplistic queueing network models.

5.2 Future Work

There are several open issues with respect to the reverse computation approach and stack reconstruction method, some of which are listed in the following sections.

5.2.1 Automatic Recognition of Reversible Patterns

How can it be determined whether a given function is perfectly reversible, *i.e.*, requires zero state? What are the necessary and sufficient conditions under which a function is perfectly reversible, taking iteration, recursion and multiple-assignment into consideration? Algorithms which automatically identify the naturally reversible patterns in the model code are important in order to prevent reverse computation from degenerating to state-saving. Perhaps a library of forward–inverse pairs of functions can help in this direction.

5.2.2 Choosing between Reverse Computation and State Saving

If a function is *not* perfectly reversible, how can it be determined whether or not reverse computation requires less state than (incremental) state saving? Informally, reverse computation is concerned with "control state" which keeps track of what is happening to the data, whereas conventional state saving is concerned with the actual data that is being manipulated. If an automated approach exists to make the choice between reverse computation and state saving, depending upon the input model, then it can be incorporated into a compiler to make the best choice for any given model. Alternatively, a more general system can switch dynamically between reverse computation and state saving, depending on the varying runtime characteristics of the model.

The relevant theoretical problem is to find whether there exist data types for which the state-saving cost for their operations differs widely when reverse computation is used instead of state-saving. In other words, are there any data types, even under certain constraints or restrictions, for which reverse computation has provably lower memory complexity than state saving (without sacrificing the time complexity)? As an illustration of this, consider a circular shift operation on an array of n elements. This operation requires no state for reverse computation, but it does appear to require $O(n)$ state size using state-saving, if a `for` loop is used for shifting. However, by using a pointer–based implementation for the array, and shifting the "start" and "end" pointers of the array, only the pointers need to be state-saved. This reduces the size of saved state to the size of two pointers. This implies that, for circular shift, the memory requirement for state-saving is only a constant factor away from reverse computation. It is unclear if this is true in general. For example, an interesting sub-problem concerns the `insert` and `delete-min` operations on a priority queue. We are not aware of any theoretical result that proves or disproves that only a constant number of *state modifications* is sufficient for an arbitrary combination of `insert` and `delete-min` operations on the queue, without sacrificing the asymptotic average time complexity of $O(\log n)$ for insertion and deletion. Reverse computation, on the other hand, requires no state history, despite state modifications, because, `insert` can be reversed using `delete`, and vice versa.

5.2.3 Reversing Arbitrary Random Number Generation

Random number generators are very commonly used in simulations. Hence, in order to apply reverse computation, it is important to have inverse functions for every forward random number generator used in the model. Is it possible for a compiler to automatically generate the inverse code when the source code of an arbitrary random number generator is given as input? Perhaps automation is possible for at least certain specific classes of generators, *e.g.* linear congruential generators? Such automated techniques can help relieve the user from the burden of writing the reverse function for his specific generator library.

5.2.4 Floating Point Arithmetic

Since floating point arithmetic is subject to roundoff, arithmetic operations can result in roundoff errors during the reverse execution. Solution approaches exist (for example, emulating a precision that is higher than the highest precision supported by the modeling language), but the performance implications are unclear.

5.2.5 Rollback Dynamics

In general, reverse computation reduces the overhead in the forward computation path, but potentially increases the rollback cost. Additional work is needed to better understand the rollback dynamics of reverse computation on a wider range of applications.

5.2.6 Reversing Data Transfer

Consider the case in which event data are copied onto the state variables of a logical process. This type of operation occurs commonly in telecommunication networks, in which the data of a packet arrival event are copied onto state buffers. If the event is canceled, changes to local state must be undone. Since the data copying operation is a destructive assignment, it degenerates to incremental state saving even if reverse computation is used. Is there a way to prevent such degeneration of reverse computation into incremental state saving? One way to find out is to copy references (or pointers), instead of the data itself. But, such an approach must carefully take model semantics into account – typically, event data and state data are assumed to be distinct from each other, and hence, writes to one must not be reflected in the other.

5.2.7 Extensions to Object-orientation

Also, a limitation of the source-level reversibility for procedural programs, as presented here, is that the techniques need to be carefully extended and/or modified for other paradigms such as object-oriented programming languages.

For example, consider the reversibility of **C**$^{++}$ programs. Techniques for reversing **C** programs are trivially applicable to **C**$^{++}$ programs as well, since all **C**$^{++}$ programs can be translated to equivalent **C** programs. However, efficiency concerns of such an approach must be resolved carefully. For example, the run-time manipulation of the `this` pointer variable, the handling of virtual function tables, and the maintenance of run time type information, must be investigated to minimize their degeneration into destructive assignments.

Another important open issue concerns how to efficiently reverse exception handling mechanisms, such as the `throw`, `try` and `catch` primitives of **C**$^{++}$.

5.2.8 Stack Data and Computation Tradeoff

For optimistic process-orientation, the stack copying approach saves the entire stack, including all control and data information. The stack reconstruction method saves copies of only those control and data items that are actually modified. The distinction between the two approaches is analogous to that of copy state saving and incremental state saving. The tradeoffs between copy and incremental state, taking into account the total size of the data and the amount of data modified per event, have been well-studied previously. Similar tradeoffs remain be explored for stack copy and stack reconstruction approaches.

5.2.9 Combining Reverse Computation with Stack Reconstruction

It is possible to integrate reverse computation into the stack reconstruction method, such that incremental state saving is replaced with reverse computation for operations between process suspension points. However, the compilation techniques and performance implications remain to be explored.

5.2.10 Reversible Threads

The integration of reverse computation with process-orientation is in fact a special case of the more generalized concept of reversible threads. So far, reversibility for threads has only been studied using data saving approaches. It would be interesting to apply reverse computation to general multi-threading systems that support reversibility.

Solutions to all the preceding issues can serve to deliver the finishing touches on the techniques presented in this thesis.

Bibliography

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] ATM Forum PNNI Draft Specification v1.0, ATM Forum 94-0471R12, 1996.

[3] B. Awerbuch, Y. Azar, and S. Plotkin. "Throughput competitive on-line routing." *Proceedings of the* 34^{th} *IEEE Annual Sumposium on Foundations of Computer Science*, November 1993.

[4] R. L. Bagrodia, W. Liao. "Maisie: A Language for the Design of Efficient Discrete-Event Simulations." *IEEE Transactions on Software Engineering*, Vol. 20(4), 1994.

[5] H. Baker. "NReversal of Fortune—The Thermodynamics of Garbage Collection." *Proceedings of the International Workshop on Memory Management*, Springer Verlag Lecture Notes in Computer Science 637, 1992.

[6] S. Bellenot. "State Skipping Performance with the Time Warp Operating System." *Proceedings of the* 6^{th} *Workshop on Parallel and Distributed Simulation (PADS'92)*, pages 53–64, 1992.

[7] Charles Bennett. "Thermodynamics of Computation." *International Journal of Physics*, pages 905–940, volume 21, 1982.

[8] J Bhasker. *A VHDL Primer*. Prentice Hall, 1995.

[9] S. Bhatt, R. M. Fujimoto, A. T. Ogielski, K. S. Perumalla. "Parallel Simulation Techniques for Large-Scale Networks." *IEEE Communications Magazine*, Vol. 36, No. 8, pages 42-47, August 1998.

[10] P. Bishop. "Using Reversible Computing to Achieve Fail-safety." *ISSRE 97, IEEE Computer Society Press*, 1997.

[11] B. Biswas, R. Mall. "Reverse Execution of Programs." *ACM SIGPLAN Notices*, Vol. 34(4), April 1999.

[12] C. J. M. Booth, D. I. Bruce. "Stack-free Process-oriented Simulation." *Proceedings of 11th Workshop on Parallel & Distributed Simulation*, 1996.

[13] K. Calvert, M. Doar, E. Zegura. "Modeling Internet Topology." *IEEE Communications Magazine*, June 1997.

[14] C. D. Carothers, R. M. Fujimoto, and Y-B. Lin. "A Case Study in Simulating PCS Networks Using Time Warp." *Proceedings of the* 9^{th} *Workshop on Parallel and Distributed Simulation (PADS'95*, pages 87–94, June 1995.

[15] C. Carothers, K. S. Perumalla, R. M. Fujimoto. "Efficient Optimistic Parallel Simulations using Reverse Computation." *In Proceedings of* 13^{th} *Workshop on Parallel and Distributed Simulation*, May 1999.

[16] C. D. Carothers, K. S. Perumalla and R. M. Fujimoto. "The Effect of State-saving in Optimistic Simulation on a Cache-coherent Non-uniform Memory Access Architecture." *Proceedings of the 1999 Winter Simulation Conference*, December 1999.

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

[18] W. Chen, J. Udding. "Program Inversion: More than Fun." *Elsevier Science Publishers*, 1990.

[19] CISCO Web pages. "The Next-Generation ATM Switch: From Testbeds to Production Networks," `http://www.cisco.com/warp/public/730/General/ngatm_wp.htm`, 1998.

[20] S. Das, R. M. Fujimoto, K. Panesar, D. Allison and M. Hybinette. "GTW: A Time Warp System for Shared Memory Multiprocessors." *Proceedings of the 1994 Winter Simulation Conference*, pages 1332-1339, December 1994.

[21] M. A. Ellis, B. Stroustrup. "The Annotated **C**$^{++}$ Reference Manual." *Addison Wesley Publishers*, 1990.

[22] C. Vanden Eynden. "Elementary Number Theory." page 141, Random House, New York, 1987.

[23] D. Eppstein. "A Heuristic Approach to Program Inversion." 9^{th} *Int. Joint Conf. Artificial Intelligence*, 1985.

[24] Michael Frank. "The R Programming Language and Compiler." `http://www.ai.mit.edu/~mpf/rc/home.html`

[25] M. Frigo, C. E. Leiserson, K. H. Randall. "The Implementation of the Cilk-5 Multithreaded Language." *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, June 17-19, Montreal, Canada, 1998.

[26] R. M. Fujimoto. "Time Warp on a shared memory multiprocessor." *Proceedings of the 1989 International Conference on Parallel Processing*, volume 3, pages 242–249, August 1989.

[27] R. M. Fujimoto. "Parallel Discrete Event Simulation." *Communications of the ACM*, pages 30–53, vol. 33(10), October 1990.

[28] R. M. Fujimoto, D. M. Nicol. "State of the Art in Parallel Simulation." *Proceedings of the Winter Simulation Conference*, pages 122–127, December 1992.

[29] R. M. Fujimoto. *Parallel and Distributed Simulation Systems.* New York: Wiley-Interscience. January 2000.

[30] R. M. Fujimoto, I. Nikolaidis, and C. A. Cooper. "Parallel Simulation of Statistical Multiplexers." Discrete Event Dynamic Systems: Theory and Applications, pages 115-140, April-July 1995.

[31] R. Gawlick, A. Kamath, S. Plotkin, and K. Ramakrishnan. "Routing and admission control in general topology networks." *Stanford University Technical Report STAN-CS-TR-95-1548*, 1995.

[32] F. Gomes. "Optimizing Incremental State-Saving and Restoration." *Ph.D. thesis*, Dept. of Computer Science, University of Calgary, 1996.

[33] A. Griewant, D. Juedos, H. Mitev, J. Utke, O. Vogel and A. Walther. "ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in **C/C**$^{++}$." *ACM Transactions on Mathematical Software*, pages 131–167, volume 22(2), Jan. 1996

[34] J. Grimm, L. Pottier, N. Rostiang-Schmidt. "Optimal Time and Minimum Space-Time Product for Reversing a Certain Class of Programs." *Research Report, Institut National de Recherche en Informatique et en Automatique (INRIA)*, February 1996.

[35] D. Jefferson. "Virtual Time." *ACM Transactions on Programming Languages and Systems*, 7(3), pages 404-425, July 1985.

[36] F. Hao, I. Nikolaidis, E. Zegura. "Efficient Simulation of ATM Networks with Accurate End-to-End Delay Statistics." *Proceedings of the International Conference on Communications*, 1998.

[37] K. Kant. *Introduction to Computer System Performance Evaluation.* McGraw Hill Computer Science Series, pages 186-192, 1992.

[38] P. J. Koopman, *Stack Computers: The New Wave.* On-line book at `http://www.cs.cmu.edu/~koopman/stack_computers/`.

[39] K. Kumaran, D. Mitra. "Performance and Fluid Simulations of a Novel Shared Buffer Management System." *Proceedings of IEEE INFOCOMM*, March 1998.

[40] G. Leeman. "A Formal Approach to Undo Operations in Programming Languages." *ACM Transactions on Programming Languages and Systems*, pages 50–87, volume 8(1), Jan 1986.

[41] P. L'Ecuyer and T. H. Andres. "A Random Number Generator Based on the Combination of Four LCGs." *Mathematics and Computers in Simulation*, volume 44, pages 99–107, 1997.

[42] E. Mascarenhas, V. Rego. "Ariadne: Architecture of a Portable Threads System Supporting Thread Migration." *Software — Practice and Experience*, Vol. 26(3), 1996.

[43] M. Matsumoto and T. Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator." *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, volume 8, Number 1, pages 3–30, January 1998.

[44] I. Mitrani. *Simulation Techniques for Discrete Event Systems.* Cambridge University Press, 1982.

[45] "The Next Generation Internet Implementation Plan." `http://www.ngi.gov`, February 1998.

[46] D. M. Nicol, *editor*, Special Issue on the TeD. *SIGMETRICS Performance Evaluation Review*, Vol. 25, No. 4, 1998.

[47] K. S. Perumalla, C. A. Cooper, R. M. Fujimoto. "An Efficiency Prediction Method for ATM Multiplexers." *Proceedings of Broadband Communications*, April 1996.

[48] K .S. Perumalla, R. M. Fujimoto, A. T. Ogielski. "The TeD Language Manual." `http://www.cc.gatech.edu/computing/pads/ted.html`, 1996.

[49] K. S. Perumalla, A. T. Ogielski, and R. M. Fujimoto. "MetaTeD: A Meta Language for Modeling Telecommunication Networks." *GIT-CC-96-32, College of Computing, Georgia Institute of Technology*, 1996.

[50] K. S. Perumalla, and R. M. Fujimoto. "A **C**$^{++}$ Instance of TeD." *GIT-CC-96-33, College of Computing, Georgia Institute of Technology*, 1996.

[51] K. S. Perumalla, M. Andrews, S. Bhatt. "A Virtual PNNI Network Testbed." *Proceedings of the 1997 Winter Simulation Conference*, pages 1057–1064, 1997.

[52] K. S. Perumalla, R. M. Fujimoto. "Efficient Large-scale Process-oriented Parallel Simulations." *Proceedings of the 1998 Winter Simulation Conference*, pages 1057–1064, 1998.

[53] K. S. Perumalla and R. M. Fujimoto. "Source Code Transformations for Efficient Reversibility." *Technical Report, GIT-CC-99-21*, College of Computing, Georgia Institute of Technology, August 1999.

[54] B. Press, and I. MacIntyre. "On the Trade-off Between Time and Space in Optimistic Parallel Discrete Event Simulation." *Proceedings of the* 6^{th} *Workshop on Parallel and Distributed Simulation (PADS '92)*, pages 33–42, 1992.

[55] A. Poplawski and D. M. Nicol. "Nops: A Conservative Parallel Simulation Engine for TeD." *In Proceedings of the* 12^{th} *Workshop on Parallel and Distributed Simulation (PADS '98)*, volume 23, pages 180–187, May 1998.

[56] B. Ramkumar, V. Strumpen. "Portable Checkpointing for Heterogeneous Architectures." *The* 27^{th} *International Symposium on Fault-Tolerant Computing (FTCS-27)*, pages 58–67, 1997.

[57] The Reversible Computing Home Page at MIT, `http://www.ai.mit.edu/~cvieri/reversible.html`

[58] F. Reynolds Jr. "A Spectrum of Options for Parallel Simulation." *Proceedings of the 1988 Winter Simulation Conference*, pages 325–332, December 1988.

[59] R. Ronngren, *et al.* "Transparent Incremental State Saving in Time Warp." *Proceedings of* 10^{th} *Workshop on Parallel & Distributed Simulation*, 1996.

[60] S. Schoemaker, *editor*. *Computer Networks and Simulation III*. Elsevier Science Publishers B.V., North Holland, 1986.

[61] M. Sokol, D. P. Briscoe, A. P. Wieland. "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution." *SCS Multiconference on Distributed Simulation*, Vol. 19, pages 34–42. July 1988.

[62] R. Sosic. "History Cache: Hardware Support for Reverse Execution." Computer Architecture News, pages 11–18, volume 22(5), December 1994.

[63] J. S. Steinman. "Incremental state-saving in SPEEDES using **C**$^{++}$." *Proceedings of the 1993 Winter Simulation Conference*, pages 687–696, December 1993.

[64] A. Tanenbaum. *Distributed Operating Systems*. Chapter 4, Prentice Hall, 1995.

[65] K. Umamageswaran, K. Subramani, P. A. Wilsey, P. Alexander. "Formal Verification and Empirical Analysis of Rollback Relaxation." The Elsevier Science *Journal of Systems Architecture*, No. 44, pages 473–495, 1998.

[66] R. A. Whitehurst, J. Brutocao. "Parallel Execution of Process-based Simulation Models." *Proceedings of SCS Multiconference*, 1998.

[67] W. Willinger, et al. "Self-Similarity through High Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level." *Proceedings of ACM SIGCOMM*, 1995.

[68] Z. Xiao, B. Unger, R. Simmonds and J. Cleary. "Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation." *Proceedings of the* 13^{th} *Workshop on Parallel and Distributed Simulation (PADS '99)*, pages 20–28, May 1999.

End of thesis text.

Vita

Kalyan S. Perumalla was born on July 27, 1968, in the current high-tech city of Hyderabad, India. He grew up in the old city portion of Hyderabad, exploring its historical segments by bicycling through its umpteen number of lanes and alleys, and honing bare-handed climbing skills on the city's numerous rocky hills. When it was time for undergraduate study, in 1987, he placed in the top 1 percentile in the state-wide aptitude test to qualify for competitive merit-based funding from the state government for his entire undergraduate engineering education at the Osmania University College of Engineering in Hyderabad. In 1991, he received his Bachelor of Engineering degree in Mechanical Engineering from the Osmania University, studying thermodynamics, control systems and the like. When he was introduced to computer programming during the second year of undergraduate study, his interest in computer science grew to the point that he undertook self-study of computer science in the library, and passed several nation-wide tests conducted by the Computer Society of India. He placed first in the FORTRAN programming test held at the national level in 1989.

To pursue his new-found passion in computer science after his bachelor's degree, Kalyan joined the Master's program in computer science at the University of Central Florida (UCF), Orlando, and obtained his M.S. degree in 1993. His master's project involved the development of a graphical interactive distributed debugging system called DBPVM for the Parallel Virtual Machine (PVM), which was later distributed to many PVM users around the world. Simultaneously with the master's study, he worked at the Institute for Simulation and Training (IST) in Orlando, where he was part of a team that developed computer generated forces for battlefield simulation. In addition to having fun attaching intelligent synthetic behavior to virtual war vehicles, such as producing realistic firing sequences for various types of simulated tanks, he was introduced to problems and solutions in distributed simulation firsthand, and also learned working as a team member. While at UCF, the annual ACM programming contests became his hobby in which participated twice (1991 and 1992). His team placed 8^{th} and 2^{nd} respectively at the southeast regional contests.

Immediately after finishing his master's in 1993, Kalyan joined the Ph.D. program in computer science at the College of Computing, Georgia Institute of Technology. During the initial years into the program, he dabbled with parallel combinatorial optimization and developed a parallel mixed integer program solver called PARINO which incorporated some of the state-of-the-art and novel techniques such as branch-and-cut methods and distributed cut management. In 1995, he met Professor Richard Fujimoto, and zeroed in on the research area that would keep him occupied for the rest of his doctoral study – namely, parallel and distributed simulation, with focus on telecommunication network modeling and simulation. In the summer of 1994, he worked at Schlumberger in Oklahoma and tasted corporate experience for the first time. In the summers of 1995 and 1996, Kalyan interned at Bellcore, New Jersey, interacting with telecommunication network experts to gain insights into the problems in modeling and simulation. In 1996, along with Richard Fujimoto, he proposed the new TeD language for network modeling as part of an NSF project, and developed a compiler to translate TeD language models for automatic parallel simulation. The

TeD language has since been used by several other researchers for many interesting network simulation problems. In 1997, he became a full-time research scientist at the College of Computing at Georgia Tech. More network simulation research work continued while he was a co-principal investigator in a related DARPA project on scalable simulation systems, and the research results were reported in numerous publications in journals and conferences. In May 1999, his co-authored paper on reverse computation received the best paper award at the premier international parallel simulation conference, PADS.

Kalyan continued to pursue his hobby of ACM programming contests by becoming the Georgia Tech team coach. In 1995, his team placed second at the regional contest and placed 11^{th} at the World Finals at Nashville, Tennessee. In 1998, his team placed first and went to the World Finals at Eindhoven, The Netherlands. Again in 1999, the Georgia Tech team placed first at the regional contest and will proceed to the World Finals in Orlando in March 2000.

During his free time, Kalyan enjoys playing racquetball and swimming, and playing with his son. He continues to work as research scientist at the College of Computing, Georgia Tech at the time of his graduation, and plans to pursue a career in the academia or research laboratories in the industry.

○ ○ ○

Techniques for Efficient Parallel Simulation and their Application to Large-scale Telecommunication Network Models

Kalyan S. Perumalla

148 Pages

Directed by Professor Richard M. Fujimoto

It is widely recognized that parallel simulation technology is necessary to address the new simulation requirements of important applications such as large-scale network models. However, current parallel simulation techniques possess limitations in the type and sizes of models that can be efficiently simulated in parallel. In particular, optimistic parallel simulation of fine-grained models have been plagued by large state saving overheads, both in event-oriented and process-oriented views, resulting in unsatisfactory parallel execution speed for many important applications. In the absence of alternative solutions, it was generally believed that optimistic approaches are inapplicable in such applications. This thesis asserts that, by use of appropriately constructed novel techniques, it is indeed possible to perform fast optimistic parallel simulation of fine-grained models, using both event-oriented and process-oriented views. In support of this claim, techniques are presented here that significantly lower the overheads, thereby enabling the capability to efficiently simulate large-scale, fine-grained models in parallel. On sample models, when compared to previously known approaches, the techniques presented here improve the simulation speed by a factor of 3 or more, while simultaneously reducing the memory requirements by almost half.

The first technique addresses the high overheads of state saving mechanisms that are traditionally used in supporting rollback operations in optimistic parallel simulation. An alternative approach called *reverse computation* is identified for realizing

rollback, which is demonstrated to significantly improve the parallel simulation speed while greatly reducing the memory utilization of the simulation.

The next technique concerns the process-oriented worldview, which is extremely useful in many domains, such as telecommunication network protocol modeling. An approach called *stack reconstruction* is developed to address the high execution overheads traditionally associated with process-oriented views, and its effectiveness is demonstrated in achieving a high rate of process-context switching during optimistic simulation.

Additional contributions of this thesis include the identification and solution of other typical problems encountered in the design, development and parallel simulation of models for real-life telecommunication network protocols. This work serves to demonstrate the readiness and feasibility of applying parallel simulation technology to today's large and complex models.

The parallel simulation techniques described in this thesis are applied and analyzed in the context of telecommunication network simulation, using representative network models. The techniques, however, are not restricted to network simulation, but are equally applicable to other domains as well. For example, the parallel simulation of any network of queues can benefit from the reverse computation system presented here. In fact, the reverse computation system is relevant to other application areas as well, such as in database recovery, and debugging environments. Similarly, the stack reconstruction approach is applicable to any multi-threaded system that requires an efficient incremental checkpointing facility for its thread states.