University of Cincinnati

Date: 12/20/2010

I, Randall King, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
WARPED Redesigned: An API and Implementation for Discrete Event Simulation Analysis and Application Development

Student's name: Randall King

This work and its defense approved by:

Committee chair: Philip Wilsey, PhD

Committee member: Fred Beyette, PhD

Committee member: Carla Purdy, PhD
WARPED Redesigned: An API and Implementation for Discrete Event Simulation Analysis and Application Development

A thesis submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
in the Department of
Electrical and Computer Engineering
of the College of Engineering

December 20, 2010

by

Randall King
BSEE, University of Cincinnati, 2010

Thesis Advisor and Committee Chair: Dr. Philip A. Wilsey
Abstract

In 1995, researchers at the University of Cincinnati released \textsc{warped} as a publically available discrete event simulation kernel. The goal of the project was to provide a system for research and analysis of the Time Warp distributed simulation synchronization protocol. \textsc{warped} was to be efficient, maintainable, flexible, configurable, and portable. It was written in C++ and used the Message Passing Interface (MPI) standard to accommodate as many parallel platforms as possible. As the software implementation was expanded with additional capabilities and optimizations, several problems with the original design became apparent. The primary problem was that the configuration of various Time Warp optimizations could only be made at compile time. As simulations increased in size and complexity, this compile time became a significant burden. Another problem, related to the first, was that \textsc{warped} could not be used and distributed as a shared library due to the compile time configuration requirement.

This thesis discusses the design and implementation of the Time Warp mechanism in a new version of \textsc{warped}, now called the \textsc{warped v2.x} series (the initial series is now called the \textsc{warped v1.x} series). The primary goal of \textsc{warped v2.x} is to provide run time configuration of the system. The goals of the previous version carry over to the new version. Several simulation models are also included in the initial release of the \textsc{warped v2.0} distribution for use in analyzing the system. In this initial version of \textsc{warped v2.x}, the system includes sequential and parallel simulation kernels that can be configured at run time for use with any of the simulation models compliant with the \textsc{warped} API. The parallel simulation kernel uses the Time Warp distributed synchronization mechanism and includes several Time Warp optimizations, including: various cancellation strategies, fossil collection algorithms, GVT estimation algorithms, state saving algorithms, event list structures, scheduling algorithms, and support for multiple communication protocols (all organized to support run time configuration/selection). This thesis presents the issues and difficulties of implementing
the optimizations along with the solutions used. The optimizations are analyzed using performance data and system profiling. With the available simulations and extensible design, WARPED v2.0 can be used to explore new optimizations for the Time Warp mechanism.
Acknowledgments

I wish to thank my advisor Dr. Philip Wilsey for his guidance and support. He has provided advice and research material throughout the process of completing this thesis. I thank Ryan Miller for finding problems in my implementations and for the help fixing them. Parts of the kernel would not be working without his help. I would like to thank my parents for supporting and accommodating me during the writing of this thesis.
# Contents

1 Introduction ............................................ 1  
   1.1 Motivation ............................................ 2  
   1.2 Thesis Overview ........................................ 3  

2 Background ........................................... 5  
   2.1 Introduction ........................................... 5  
   2.2 Parallel Discrete Event Simulation ......................... 5  
      2.2.1 Time Warp Mechanism ................................ 7  
   2.3 WARPED v1.x .......................................... 9  
   2.4 Other Time Warp Simulation Implementations ......................... 12  
   2.5 Summary ............................................ 13  

3 WARPED Simulation Interface ................................. 14  
   3.1 Introduction ........................................... 14  
   3.2 WARPED Simulation Interface ................................. 14  
      3.2.1 Application ....................................... 15  
      3.2.2 Event .......................................... 15  
      3.2.3 SimulationObject .................................... 16  
      3.2.4 State .......................................... 19  
      3.2.5 VTime ......................................... 19  
   3.3 Example: A PingPong Simulation Model ......................... 20  
   3.4 Configuration .......................................... 23
4 WARPED Kernel Interface ........................................... 24
  4.1 Introduction .................................................. 24
  4.2 Sequential Simulation ........................................... 26
  4.3 Time Warp Simulation ........................................... 28
  4.4 Threaded Time Warp Simulation .................................. 36

5 Time Warp Optimizations Implemented .................................. 40
  5.1 Introduction .................................................. 40
  5.2 Output Events ................................................... 40
    5.2.1 Lazy Cancellation ........................................ 41
    5.2.2 Dynamic Cancellation ...................................... 44
    5.2.3 One Anti-Message .......................................... 47
  5.3 State Saving .................................................. 50
    5.3.1 Adaptive Period ........................................... 51
  5.4 GVT Estimation ................................................... 52
    5.4.1 Mattern ................................................... 53
  5.5 Fossil Collection .................................................. 57
    5.5.1 Optimistic Fossil Collection .................................. 57
  5.6 Summary .................................................. 65

6 Analysis .................................................. 66
  6.1 Introduction .................................................. 66
  6.2 Simulations .................................................. 66
    6.2.1 PHOLD ................................................... 67
    6.2.2 RAID ................................................... 68
    6.2.3 SMMP ................................................... 69
  6.3 Testing Setup .................................................. 71
  6.4 Verification .................................................. 72
List of Figures

2.1 The logical processes of a physical system. ........................................ 6
2.2 Straggler arriving at an LP. ............................................................. 8
2.3 The LP after recovering from a rollback. ....................................... 8
2.4 LPs in a distributed WARPED simulation. ................................... 10

3.1 The model for the default PingPong simulation. ............................... 21
3.2 Implementation of important PingObject functions. ......................... 22
3.3 Implementation of the copyState function. .................................. 23

4.1 Execution flow of a simulation. .................................................... 25
4.2 Structure of a sequential simulation. ............................................ 27
4.3 Execution flow of a sequential simulation. .................................... 28
4.4 Structure of a parallel Time Warp simulation. ............................... 29
4.5 Execution flow of the Time Warp simulate function. ....................... 31
4.6 Processing of events with incorrect state saving. ........................... 33
4.7 Processing of events after the state has been restored after a rollback. .... 33
4.8 The rollback process. .................................................................. 34
4.9 The manager thread process. ....................................................... 38
4.10 The worker thread process. ......................................................... 39

5.1 An example of lazy cancellation in action. .................................... 41
5.2 The lazy cancellation process. ...................................................... 43
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>The dynamic cancellation process.</td>
<td>46</td>
</tr>
<tr>
<td>5.4</td>
<td>Sending all anti-messages.</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>One anti-message optimization.</td>
<td>48</td>
</tr>
<tr>
<td>5.6</td>
<td>Periodic state saving and coast forward phase.</td>
<td>50</td>
</tr>
<tr>
<td>5.7</td>
<td>Periodic state saving and overhead costs.</td>
<td>51</td>
</tr>
<tr>
<td>5.8</td>
<td>Mattern’s GVT Algorithm</td>
<td>53</td>
</tr>
<tr>
<td>5.9</td>
<td>Mattern GVT in Warped, Master.</td>
<td>55</td>
</tr>
<tr>
<td>5.10</td>
<td>Mattern GVT in Warped, Slave.</td>
<td>56</td>
</tr>
<tr>
<td>5.11</td>
<td>Fossil collection using GVT.</td>
<td>59</td>
</tr>
<tr>
<td>5.12</td>
<td>Optimistic fossil collection.</td>
<td>59</td>
</tr>
<tr>
<td>5.13</td>
<td>Recovery mechanism for master simulation manager.</td>
<td>62</td>
</tr>
<tr>
<td>5.14</td>
<td>Recovery mechanism for slave simulation manager.</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>The default small PHOLD configuration.</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>The default small RAID configuration.</td>
<td>68</td>
</tr>
<tr>
<td>6.3</td>
<td>A 4 processor SMMP configuration.</td>
<td>70</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance of output manager configurations.</td>
<td>74</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance of state manager configurations.</td>
<td>77</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance of GVT manager and fossil collection manager configurations.</td>
<td>79</td>
</tr>
<tr>
<td>6.7</td>
<td>Performance of event set manager configurations.</td>
<td>80</td>
</tr>
<tr>
<td>6.8</td>
<td>Performance of communication manager configurations.</td>
<td>82</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Number of threads and processors for each simulation manager. .......................... 26

6.1 Platform specification. ............................................................................................... 71
6.2 Output manager performance for the four different configurations using the PHOLD simulation. .............................................................. 74
6.3 Output manager performance for the four different configurations using the RAID simulation. .............................................................. 75
6.4 Output manager performance for the four different configurations using the SMMP simulation. .............................................................. 75
6.5 State manager performance for five different configurations using the PHOLD simulation. .............................................................. 76
6.6 State manager performance for five different configurations using the RAID simulation. .............................................................. 77
6.7 State manager performance for five different configurations using the SMMP simulation. .............................................................. 77
6.8 GVT manager and fossil collection manger performance for four different configurations using the PHOLD simulation. .............................................................. 78
6.9 GVT manager and fossil collection manger performance for four different configurations using the RAID simulation. .............................................................. 79
6.10 GVT manager and fossil collection manager performance for four different configurations using the SMMP simulation. .............................................................. 79
6.11 Event set manger performance for two different configurations using the PHOLD simulation. .............................................................. 81
6.12 Event set manger performance for two different configurations using the RAID simulation. .............................................................. 81
6.13 Event set manger performance for two different configurations using the SMMP simulation. .............................................................. 81
6.14 Communication manager performance for two different configurations using the PHOLD simulation. .............................................................. 81
6.15 Communication manager performance for two different configurations using the RAID simulation. ............................................. 82

6.16 Communication manager performance for two different configurations using the SMMP simulation. ........................................... 82

6.17 Percentage of total program execution time spent in selected functions. ........................................... 83
Chapter 1

Introduction

Computer simulations model the behavior of real world processes in many fields such as natural sciences, economics, engineering, and military [1]. The need for simulation modeling arises because accessing or building the actual physical systems is potentially difficult, and because many systems are simply too complex and dynamic for mathematical analysis. Experimenting on an actual system can become very expensive as it may need to be rebuilt numerous times. In many cases, the modeled systems are easily represented by a collection of processes that model the physical systems at discrete time increments. These types of simulations are called *discrete event simulations*. As these simulation models (and the physical systems they represent) grow in size, they require more time to complete when executed on a sequential computing platform. This can lead to unacceptably long simulation run times or to simulations with unsupportably large memory requirements. A common example given of this phenomenon is a weather forecast simulation for the next day that takes two days to complete. By the time it is complete, it is irrelevant. Reducing the execution time can produce results in a more timely manner and also allow a larger sample size as more simulations can be run. Parallel discrete event simulation (PDES) attempts to increase the performance of simulations by executing portions of simulations concurrently on parallel or distributed computing platforms [2]. Researchers have attempted different methods for designing PDES solutions [3]. The most promising PDES approaches have distributed synchronization mechanisms and these solutions generally fall into one of two categories: *conservative* or *optimistic*. The Time Warp mechanism is one optimistic synchronization protocol that has been widely discussed in PDES literature [4].
Some of the initial attempts at the University of Cincinnati parallelized simulations by embedding the Time Warp protocol directly into the simulations. These early researchers developed VHDL analyzers using the Time Warp mechanism to run on distributed memory platforms [5] and on shared memory platforms [6]. However, as other researchers introduced new techniques and optimizations to the Time Warp mechanism, modifying the VHDL analyzing tools became increasingly difficult. The Time Warp code was tightly intertwined with the VHDL simulation code. A need existed to separate the VHDL simulation code from the Time Warp implementation.

The first effort by University of Cincinnati researchers to separate the code base resulted in a discrete event simulation kernel named WARPED [7] and a VHDL simulation kernel called TyVIS [8]. The goal of the project was to provide an easy to extend discrete event simulator for researching the Time Warp mechanism. To accomplish this, the kernel was written in C++ and used object-oriented techniques. Simulation developers could create simulations by using an application programming interface (API) provided by WARPED. The simulation kernel hid the actual implementation of the Time Warp mechanisms from the simulation model developer. In addition, researchers could easily modify the kernel to add new optimizations. Some of the optimizations studied by subsequent researchers included event cancellation management [9, 10], global virtual time estimation [11, 12], state management [13–16], and fossil collection [17]. The WARPED kernel performed inter-processor communication through MPI [18] to provide support for different platforms including shared memory multiprocessors and networks of workstations. The kernel also included several test simulations to allow easy and consistent performance analysis.

1.1 Motivation

As the WARPED kernel and the simulation models grew in size, several problems also became more significant. The primary issue was that the setting of many of the various Time Warp configurations of WARPED occurred at compile time. When users wanted to change a single configuration option for testing and analysis, they had to first recompile the entire kernel and simulation model. If the simulation model was very large, this recompilation required considerable time. In addition, because it would no longer be configurable, the WARPED kernel could not be effectively distributed as a shared library. Another problem was that new developers had an increasingly difficult time reading and understanding the kernel as preprocessor macros
used for the compile time configuration were scattered throughout large portions of the code.

To overcome these problems, WARPED has been completely reworked and named WARPED v2.x. The goal is to continue to provide a Time Warp simulation kernel for application development and research but with more flexible, runtime enabled, configuration options. Ideally, the goal is to support configuration down to the level of the basic simulation engine (parallel or sequential). As a side benefit, the WARPED kernel API now provides only the base services absolutely needed by the simulation and no more.

A sequential run-time kernel for the WARPED system is fully functional and has been well-tested using a full regression test suite. However, the Time Warp run-time kernel existed only as a collection of code stubs and incompletely tested method definitions. This thesis discusses the implementation and testing of a fully functional Time Warp mechanism within the WARPED v2.0 system. The initial efforts focused on developing a fully functional Time Warp simulation kernel. However, basic functionality is not very useful however if it does not perform well. Thus, in addition to optimizing the actual implementation, various Time Warp optimizations were added to improve performance. Simulations were developed for performance analysis and for profiling of the kernel. This thesis documents the efforts made to have a fully functional Time Warp simulation kernel for the WARPED v2.x API and provides a preliminary analysis of many of the key optimizations to the Time Warp protocol that have been included with this initial implementation.

1.2 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 provides some background information on parallel discrete event simulation with a focus on the Time Warp mechanism. More details about the WARPED v1.x series are also presented. Finally, several other Time Warp simulators are also reviewed.

Chapter 3 presents the WARPED v2.x kernel API that simulation developers use to create simulations. A simple example simulation named PingPong illustrates how the API is used to create a simulation model.

In this first version (2.0) of the WARPED v2.x series, the parallel version can be configured with two different organizations. One instantiates as a collection of concurrently executing heavyweight processes each with a single execution thread. The second instantiates the collection of heavyweight processes each supporting threaded execution. Chapter 4 discusses the three different modes of execution (sequential, parallel
single thread, parallel multi-threaded). The general architecture of each is presented. The development and implementation of the Time Warp mechanism is discussed in detail including the challenges encountered for this implementation. The focus of this thesis is the non-threaded implementation as the threaded version is the subject of another thesis [19].

Chapter 5 discusses several Time Warp optimizations added to the non-threaded parallel version. These include optimizations for event cancellation, state management, global virtual time estimation, and fossil collection. These optimizations have not been fully implemented or tested for use with the threaded version, and therefore, no performance analysis is done on configurations with the threaded version.

Chapter 6 provides information about how the various optimizations perform as well as profiling data of the kernel. The simulations used for the analysis also are presented.

Finally, Chapter 7 presents an overview of what was accomplished and any future work that can improve the \textsc{warped v2.x} series.
Chapter 2

Background and Related Work

2.1 Introduction

WARPED is a discrete event simulation API and implementation that supports simulation modeling using either sequential or parallel simulation engines. The next section presents information about parallel discrete event simulation with a focus on the Time Warp mechanism [4, 20]. The design of the WARPED v1.x series is discussed in Section 2.3. Finally, other work with parallel discrete event simulators is discussed in Section 2.4.

2.2 Parallel Discrete Event Simulation

A Discrete Event Simulation (DES) models a system that changes states at discrete points in simulated time. The change in states occurs in response to time-stamped events. The main components of a sequential discrete event simulation are as follows:

- Event list: contains all events that have not been processed.
- State variables: represent the various states of the system.
- Global clock: tracks the simulation time.

A common strategy in DES is to model a system as physical processes with each physical process having a state [21]. Logical processes then represent these physical processes in the simulation. Each logical
CHAPTER 2. BACKGROUND

2.2. PARALLEL DISCRETE EVENT SIMULATION

process (LP) has a state and references the global clock. Any interaction between LPs occurs through time-stamped events. An example of the process based methodology is a simulation involving logic gates. Figure 2.1 shows the conversion of the physical system to the logical model used in the simulation. The three AND gates are the physical processes of the system and would correspond to LPs. The current input signals and the resulting output signal make up the state of an LP. The events in the simulation are the changes in the value of signals into and out of the gates. An LP would only send events to the proper LP as indicated by the arrows.

Parallel discrete event simulation involves executing (some or all of the) LPs concurrently. In asynchronous simulations, the time-stamps of events are irregular with only a few events occurring at any given time. This causes synchronization of parallel simulations through a global clock or central event dispatch unit to have poor performance [20]. Allowing the LPs to process events independently through a distributed synchronization mechanism might help alleviate performance problem of a central dispatch mechanism. However, the distributed synchronization mechanism must be able to identify sufficient events for concurrent execution to ultimately improve performance.

The synchronization challenges of PDES are better understood by explaining the event processing steps. Each LP in a simulation processes events from its event list in non-decreasing order. Without this requirement, an LP might process an event that does not have the lowest time-stamp in the event list before an event.
with the minimum time-stamp and modify a present state variable. This means that a simulation would be modeling a system where the future changes the past. This type of error is called a causality error. No causality errors will occur in a simulation as long the local causality constraint is followed. This constraint requires that all LPs process events in non-decreasing time-stamp order [22].

Following the local causality constraint for PDES is not simple however. Consider a parallel simulation that is executing two LPs concurrently. Each LP will process events from its event list in non-decreasing time-stamp order. The only events that each LP knows about are the events in its own event list. Each LP will continuously execute the events in its list because all of the known events are in safe non-decreasing time-stamp order. When an LP attempts to communicate with another LP by sending an event, it is entirely possible that this new event has a time-stamp that is earlier than the receiving LP’s current time progress (because the receiving LP has no knowledge of the other LP). This will trigger a causality error and is the primary challenge of PDES: how to sequence events in non-decreasing time-stamp order. Many techniques attempt to overcome this problem, and they can generally be classified as either conservative or optimistic techniques. Conservative techniques attempt to prevent causality errors from ever occurring. Chandy, Misra, and Bryant developed some of the earliest attempts at PDES using conservative techniques [23, 24]. Optimistic techniques permit causality errors to occur, but focus on detecting and then correcting the errors [20]. This thesis addresses the implementation of a Time Warp synchronized optimistic PDES kernel for the WARPED system. Therefore, additional details on conservative mechanisms are not presented further.

2.2.1 Time Warp Mechanism

One widely used optimistic synchronization protocol is called the Time Warp Mechanism. In Time Warp, an LP processes events in its local input queue without considerations to other LPs about the processing of an event. When an LP receives a new input event that is supposed to be processed at a time earlier than the local simulation time (LVT), a causality error has occurred (such an event is called a straggler event). The arrival of a straggler event triggers the LP to rollback to an earlier time to reprocess the events in their proper order. The rollback affects the state of the LP and the events it has (prematurely) sent. The LP must undo any changes made to the state of the LP by the premature event processing. The LP also has to correct any events that were prematurely sent. In general the LP will save states as it processes events so that it
can restore a previous state upon rollback. Likewise, the LP will maintain a set of output anti-messages that can be used to cancel events that the LP sent prematurely. When an anti-message arrives at an LP and the receive time-stamp of that anti-message is lower than the LP’s LVT, then the LP will rollback. This results in a transmission of anti-messages until all prematurely sent events have been cancelled.

A Time Warp synchronized LP requires several additional data structures to handle rollbacks. While a sequential simulation LP requires an input queue to store unprocessed events, in a Time Warp simulator the input queue must also store processed events. An LP retains the processed events because they may need to be processed again after a rollback occurs. An output queue that saves all output events that an LP sends must also be maintained so that anti-messages can be sent after a rollback. In addition, the LP must maintain a collection of previously used states in a state queue that can be reused for a rollback. Finally, each LP
contains a Local Virtual Clock (LVT) that only changes value when switching to the receive time of the next event available for processing.

To illustrate the rollback recovery process, Figure 2.2 displays the arrival of a straggler with a receive time of 56 when the LVT is 62. The LP has just saved the state at time 60. The straggler triggers the LP to remove the output event with a send time of 60 from the output queue and send an anti-message. Figure 2.3 shows the state of the queues after the rollback process is complete and event processing resumes. The nextEvent pointer points to the straggler and the LVT has been set to 56. The LP has restored its state to a value of B, which was its value at time 55.

By saving processed events, output events, and states, an LP can quickly use large amounts of memory. At some point, the simulation has to reclaim the saved events and states so the system does not run out of memory. Performing irrecoverable operations such as I/O while rollbacks are possible is also a challenge. Time Warp uses an estimated global virtual time (GVT) [4] to solve this problem. The GVT is equal to the lowest time-stamp of all unprocessed events in the simulation. Since the GVT is the lowest time of any unprocessed event, no rollbacks can occur to any time less than the GVT. This means that an LP can safely remove and reclaim the memory of any saved events and states with time-stamps less than the GVT. The LP can also commit any irrecoverable operations with a time-stamp less than the GVT. This process of memory reclamation and irrecoverable operation commitment is called fossil collection.

Many algorithms have been developed to optimize various aspects of the Time Warp mechanism such as GVT estimation, output event management, state saving, and fossil collection. These optimizations and their implementation in WARPED v2.0 are described in detail in Chapter 5.

2.3 WARPED v1.x

The original motivation for developing a Time Warp simulation kernel came from the QUEST [5] and VAST [6] projects. These attempted to build a parallel VHDL analyzer with the Time Warp mechanism implemented in the VHDL kernel. However, this made implementing new Time Warp optimizations and performing analysis on those optimizations difficult. To eliminate these problems, researchers at the University of Cincinnati developed a discrete event simulation kernel that was separate from but supporting the VHDL simulation kernel. They released the discrete event simulation kernel in 1995 and named it
WARPED [7]. The other portions of the VHDL project were a VHDL analyzer and code generator named SAVANT and a VHDL simulation kernel named TyVIS [8].

Another motivating factor for the development of the Time Warp simulator was to provide a base for comparing different optimization algorithms. Researchers were developing different optimizations to solve the same problems (such as GVT estimation), but performed the analysis using different simulators, making comparison to other optimizations difficult. WARPED was to be easy to extend and modify so that researchers could easily compare various optimizations.

There were also several other objectives for WARPED. Efficiency was desired in terms of time and space as well as in communication and scalability. The kernel was written in C++ to provide an object oriented and modular design. Documentation and unit testing were included to provide greater maintainability. WARPED was designed to be flexible enough to allow for implementation of most current and future optimizations. Many options were available to select various optimizations. The Message Passing Interface (MPI) [18] was used to support many different parallel platforms.

The WARPED kernel presents an API to the simulation developer for implementing simulation models.
To create a simulation model in WARPED, the developer must define the following objects using the WARPED API: simulation objects, states, and events. A simulation object represents a physical process, and contains a state queue and an output queue, but not an input queue. The kernel groups the simulation objects together into LPs. Each LP maintains a single input queue for all simulation objects in the LP. This input queue structure allows for efficient processing of events in lowest time stamp first (LTSF) order [25]. Figure 2.4 shows the basic design of a distributed simulation. There are several LPs, each containing several simulation objects and each running on separate computing nodes connected by a network. Simulation objects in the same LP do not use the messaging system to communicate. Instead, the kernel sends events directly from an object into the input queue for the LP. While an LP can have many simulation objects, the kernel does not require them to be synchronized. Researchers could develop any type of scheduler to determine which simulation object to process if LTSF ordering was not desired.

The kernel uses the object oriented features of C++ to provide the base simulation object, event, and state classes that users derive when creating simulation models. This allows the kernel to use the base classes when performing tasks such as event list management and state saving. The kernel does not have specific knowledge about anything in a specific simulation model. Likewise, the simulation model developer only has to be concerned with the structure of the simulation model. The implementation of the underlying Time Warp mechanism is completely hidden from the model developer. Simulation developers must only derive several functions from base classes used by the simulation kernel, such as a function that defines the event processing step. To allow the system to run on heterogeneous platforms, WARPED also uses serialization and deserialization functions for the events and states in the kernel when communicating between different processors.

Many Time Warp optimization options are available in the WARPED v1.x series. Configuration options include: list management, scheduling, memory management, fossil collection, and event cancellation. Unfortunately, as previously described, these options can only be configured at compile time. Developers believed this decision would be sufficient during the initial design [26]. However, this limit proved problematic. The main problem is that compile times have grown very large. As simulations (such as VHDL simulations) become larger and more complex, the recompile time required for optimization comparisons becomes a significant burden. In addition, management of the configurations in the kernel is difficult. The
configuration options consist of preprocessor macros in makefiles and header files, making it difficult to have several options available. Some sections of the kernel code are often difficult to read because of the number of `#ifdef` macros for the different optimization options. Using these macros does allow for the size of the \textsc{warped} library to be smaller. For example, if there are three different GVT estimation algorithms, only the code for the one selected will be compiled. The other two would not be included in the build at all. Due to the configuration of optimization options at compile time, \textsc{warped}$\text{v1.x}$ cannot be distributed as a shared library.

### 2.4 Other Time Warp Simulation Implementations

There are several other projects that have involved the development of parallel discrete event simulators using the Time Warp mechanism. Some of the more important of these are briefly reviewed below.

The Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) is a general discrete event simulator developed by the Jet Propulsion Laboratory [27]. It is written in C++, uses an object oriented design, and can run on several different platforms. SPEEDES can be configured to use conservative and optimistic synchronization mechanisms. It targets distributed memory parallel computing platforms and has extensive libraries to support mixed-simulator integration over (primarily) the HLA backplane [28].

The Georgia Tech Time Warp (GTW) system is a parallel discrete event simulator based on the Time Warp mechanism [29]. It is written in C and is designed to run on shared memory multiprocessor machines and to work well with small granularity simulations. It supports a number of sub-algorithms to optimize Time Warp synchronized simulations in a shared memory environment.

The Rensselaer’s Optimistic Simulation System (ROSS) is a modular Time Warp system written in C [30]. The goal of ROSS is to improve upon GTW by providing extreme performance capability and low memory utilization. It has been used on both shared memory machines and distributed systems. Very large simulations have been ported to execute on the large IBM Blue Gene machines [31].

Researcher at UCLA have developed a simulation environment named Parsec (parallel simulation environment for complex systems) [32]. The goal of the project is to provide an environment in which models can be easily implemented into parallel simulations. Simulations can be developed using either a C-based
language name Parsec, a C++ library named Compose, or a GUI named Pave.

### 2.5 Summary

This section provides information about discrete event simulation and the challenges of parallel discrete event simulation. The Time Warp mechanism is a common synchronization protocol used for PDES. Several discrete event simulators have been developed that use the Time Warp mechanism including an earlier version of WARPED. This earlier version of WARPED contained several shortcomings, particularly the compile time configuration. In the next chapter, the redesigned WARPED v2.x API will be described and an example of its use is presented. Thereafter, in Chapter 4, the implementation of the parallel simulation kernel for WARPED v2.0 is presented.
Chapter 3

WARPED Simulation Interface

3.1 Introduction

The WARPED simulation kernel presents an application programming interface (API) for developing simulation models. The creation of WARPED simulation models is a process of implementing and defining classes and functions that are used by the simulation kernel. This chapter describes the simulation interface API of WARPED v2.x and provides an example of an implemented simulation model named PingPong.

A simulation model built compliant to the WARPED API consists of three main components, namely: simulation objects, states, and events. Each simulation object represents a single physical process. It has a single state, and interacts with other simulation objects by consuming and generating time-stamped events. A simulation object processes events and will often modify its state based on the event processed. The primary task of a simulation model is to define how simulation objects process events, and how those events modify simulation object states. The simulation model also involves the construction and initialization of the simulation objects and their states. The following sections provide details about what must be defined in a simulation model.

3.2 WARPED Simulation Interface

To create a simulation model, the developer must derive C++ class definitions from several abstract base classes. The necessary functions in the base classes are pure virtual functions, so compile time errors
CHAPTER 3. WARPED SIMULATION INTERFACE  3.2. WARPED SIMULATION INTERFACE

will result if the developer does not define them. Descriptions of the base classes and the most important functions that the developer must define are presented in the next few sections.

3.2.1 Application

The WARPED kernel uses the Application class to perform any setup tasks before the simulation begins processing events and any cleanup tasks at the end. Some of the important functions include:

- **initialize** This function allows the simulation model to make any checks necessary. Generally, this function is where simulation models check command line parameters specific to the simulation model.

- **getPartitionInfo** This function is the most important function in the class. Here, the developer must create the simulation objects and assign them to different processors. The function takes the number of processors used in the simulation as a parameter. The partitioning of simulation objects is left to the developer because optimal partitioning varies from simulation model to simulation model. The function returns the partitioning information.

- **finalize** The function allows the simulation model to perform any needed tasks after the simulation has finished executing.

3.2.2 Event

Events represent DES events and they also include the infrastructure to support communication of event information between simulation objects. A simulation model must contain at least one definition of the Event class and can actually have several. A simulation model that has simulation objects that communicate small amounts of information and similar information will usually only need one event definition. An example of a simulation model that would only require one event definition would be a heat transfer simulation. Each simulation object would represent a cross section, so all simulation objects would be of the same type. The event would be the new temperatures of the neighboring simulation objects, so the events would be communicating little information. However, simulation models that contain simulation objects that pass different information in different situations may want to use several event definitions. While it is possible
to combine the information of different event types into one event class, the event would contain extraneous information that needlessly increases the size of the event object. The increase in size can cause a decrease in communication performance and present logical confusion as to the purpose of the event. By using more than one event definition, simulation objects are able to communicate only necessary information.

All event class definitions must contain a send and receive time, a sender and receiver simulation object, and a unique event identifier. The send time is the time at which a simulation object sends an event. The receive time is the time at which the receiving simulation object processes the event. The event identifier uniquely identifies events sent from the same simulation object. Accessor functions for these values are pure virtual functions in the Event class. However, the WARPED kernel provides a base implementation class named DefaultEvent that defines these accessor functions for convenience. Therefore, when implementing the Event class for the simulation model, the developer should use the DefaultEvent class as the base class. The primary functions that must be defined by the user include:

- **eventCompare** This function returns true when the event passed as a parameter is equivalent to the event that called the function. The developer must derive this function to account for simulation model specific data members of the Event class. The event identifier should not be included in this comparison due to possible event regeneration when executing parallel simulations.

- **serialize** The kernel calls this function when the event needs to be sent over a network. All data required to create a new event that is identical to the serialized event must be serialized in this function.

- **deserialize** The kernel calls this function after receiving an event over a network. The function returns a new event constructed using the information in the serialized network message.

### 3.2.3 SimulationObject

Because the WARPED system is planned for experiments with exploratory implementations of parallel and distributed simulation kernels, some special operations for event processing are necessary. In particular, the expectations of experiments with the Time Warp mechanisms force the design to export some features to facilitate rollback and premature computations. Chief among these features is the explicit encapsulation and
CHAPTER 3. WARPED SIMULATION INTERFACE

3.2. WARPED SIMULATION INTERFACE

definition of state as an object. Furthermore, committable operations such as file I/O will be constrained
by specific simulation kernel supported operations. These limits are most visible to the simulation model
developer in the SimulationObject class definitions.

Instances of the SimulationObject class represent physical processes and are the primary com-
ponents of a discrete event simulation. Each different type of physical process in the system must have a
 corresponding derivation of the SimulationObject class. Each SimulationObject has a unique
identifier, a state, and a local simulation time. An example of a simulation model with several derivations of
the SimulationObject class is a logic gate circuit. If the circuit contains AND and OR gates only, then
there should be two classes, one to represent each type of gate.

There are several functions in the SimulationObject class that the developer does not have to
derive. These functions provide convenient operations to use within the derived SimulationObject
class functions. Some of the important functions are:

- **receiveEvent** This function sends an event to a simulation object. It takes an event as a parameter
  and sends that event to the simulation object instance that calls the function.

- **haveMoreEvents** This function determines if there are more events for the simulation object to
  execute at the current local simulation object time. It returns true when there are more unprocessed
  events at the current time.

- **getEvent** This function returns the next event at the current local simulation time. If there is no
  such event, then the function returns NULL.

- **getState** This function returns the current state for the simulation object.

The following are the pure virtual functions that the developer must derive:

- **initialize** The kernel calls this function on every simulation object before the simulation begins
  executing. This allows the object to perform any setup required (such as opening input and output
  files). At least one of the simulation objects must generate an event within this function, otherwise
  the simulation will complete without performing any work because there are no events to process.
• finalize The kernel calls this function on every object after the simulation ends. This allows the object to perform any necessary clean up (such as closing files or outputting results or final state). Any call to the receiveEvent function within this function will do nothing because the simulation is already complete.

• executeObject The actual processing of events occurs in this function. The haveMoreEvents function must be called in a loop so that the simulation object processes all events at the current local simulation time. This is required for orderly state saving within the kernel for parallel simulation. The getEvent function is used to obtain the next event to process for the simulation object. The processing of an event modifies the simulation object state by first calling getState to access the state and then modifying the returned state.

• getName This function returns the name of the instance of the SimulationObject. Each instance of the SimulationObject class must have a unique name. In most cases, the name is initialized in the constructor.

• reclaimEvent The kernel calls this function when an event processed by the object is no longer needed. This function could simply delete the event or insert it into a pool of events for memory reuse. Reusing the events could reduce memory allocation overhead.

• allocateState The kernel calls this function whenever it needs to allocate a new state for a simulation object. The kernel calls this function before the simulation starts to initialize the simulation object’s state. This function can simply create a new state using the C++ new operator, or possibly reuse memory from an old state.

• deallocateState The kernel uses this function when it no longer needs a state belonging to a simulation object. The simulation object could simply delete the state or possibly insert it into a pool of states to be used when a new state is needed in allocateState to reduce memory allocation overhead.

Some simulation models may require the use of file input and output operations. These operations are provided by the predefined methods: getIFStream, getOFStream, and getIOFStream. These
CHAPTER 3. WARPED SIMULATION INTERFACE

3.2. WARPED SIMULATION INTERFACE

methods correspond to input streams, output streams, and input/output streams. For the various streams, the `readLine` function returns a line from the corresponding file and the `insert` function outputs data to the file. A file is opened by passing the file name to one of the `SimulationManager` stream functions. The function will return a new stream that can be used for file operations.

File operations must occur through the kernel to ensure support for rollback and proper commitment as the parallel simulation optimistically processes event. While this limits file operations to simple reading and writing, it provides basic input/output functionality that should support most simulation models destined for execution in large Beowulf clusters. While conventional C++ I/O operations can still be used, all writes/reads including those done prematurely would be directly issued to the file system. Thus on rollbacks, premature reads and writes would not be undone. While the user could potentially attempt a manual correction (such as define a post-simulation rebuild of output files) of these file operations, no support for notification the model of rollbacks is provided in the WARPED v2.x system.

3.2.4 State

Most implementations of `SimulationObject` will require a state to be defined. Events received by the simulation object typically modify this state. Because the simulation objects are simulation specific, the states of these objects are also simulation specific and must be defined. The primary function to derive is the `copyState` function. In most cases, the data members of a derived state can be set by accessing the copied state’s data members. However when a data member is a pointer, the pointer cannot simply be copied. The function must copy the data pointed to by the pointer. Essentially, the function must make a deep copy of the state.

3.2.5 VTime

For many simulation models, a simple integer time model is sufficient. In other cases, the time model must be more complex. For example, the time model for digital systems characterized in the hardware description language VHDL [33] requires a two-tuple time representation. As previously mentioned, one of the ongoing uses for the WARPED kernel is simulation of VHDL models. Hence, WARPED’s API provides some mechanism for complex time models in the simulation model and runtime environment.
achieves this by providing a default integer-based virtual time model that can be extended by using a derived class.

WARPED uses an abstract base class named \texttt{VTime} to represent virtual time. The default integer-based time class in WARPED is called \texttt{IntVTime} and is derived from the \texttt{VTime} class. To derive a more complex time model, the \texttt{VTime} class must be derived. The kernel must be able to perform comparisons on \texttt{VTime} values to be able to determine order for events and other objects with time stamps. When implementing \texttt{VTime}, the simulation model developer must define the C++ comparison operators (equal to, not equal to, less than, greater than, less than or equal to, and greater than or equal to). These following functions must also be derived:

- \texttt{getPositiveInfinity} This function returns the time of positive infinity. This is generally the largest time value that the derived time class supports.
- \texttt{getZero} This function returns the time of zero. The kernel uses this function to set a start time for the simulation.
- \texttt{getApproximateIntTime} This function returns the approximate integer time. The kernel needs this function because it needs a single time value in some cases, such as when dealing with time windows or containers that hold objects of a single time value.

### 3.3 Example: A PingPong Simulation Model

PingPong is an example of a simulation model with fine-grained granularity. Events are sent from one \texttt{SimulationObject} to the next with no real processing. Figure 3.1 shows the default simulation model configuration. In this model, there are five \texttt{SimulationObjects} that represent players who pass a ball around in a ring pattern. An event represents the passing of the ball from one player to the next. One player is designated a master before the simulation starts and serves a ball that is passed around from one player to the next. When the ball gets all the way around to the master, the master starts a new ball. This process continues until the master has started and subsequently received a specified number of balls.

The derivation for the \texttt{SimulationObject} class in PingPong is named \texttt{PingObject}. In addition to the required \texttt{initialize} and \texttt{executeProcess} methods, the class includes two helper methods.
CHAPTER 3. WARPED SIMULATION INTERFACE

EXAMPLE: A PINGPONG SIMULATION MODEL

Figure 3.1: The model for the default PingPong simulation.

named startBall and sendEvent. These methods provide a convenient way to send new events. Figure 3.2 shows the implementation of these functions as well as the initialize and executeProcess functions required by the kernel. The PingObject class contains a data member named isMaster to indicate if a PingObject will start new balls to send. Depending on the configuration, PingPong can have several masters. For all simulation objects other than those with isMaster set to true, the executeProcess simply sends any received event to the next object. When a PingObject with isMaster set to true receives an event and the event was started by the PingObject, it starts a new ball. The executeProcess function uses the getEvent function to obtain the next event to process. The haveMoreEvents function is checked to ensure that all events with a receive time equal to the local simulation object time have been processed. PingPong uses the default IntVTime class to represent time for the simulation. This choice can be seen in the sendEvent function where recvTime is initialized.

The PingObjectState class derives the State class for PingPong. Figure 3.3 shows the implementation of the copyState function. The state for PingPong simply keeps track of how many balls a player sent, received, and started. The copyState function simply copies these three data members.
void PingObject::initialize()
{
    sendTo = getObjectHandle(myDestObjectName);
    if( isMaster )
    {
        startBall();
    }
}

void PingObject::startBall()
{
    PingObjectState* myState = static_cast<PingObjectState*>(getState());
    myState->ballStarted();
    sendEvent( getName() );
}

void PingObject::executeProcess()
{
    PingObjectState* myState = static_cast<PingObjectState*>(getState());
    ASSERT(myState != 0);
    // If we're a master and we've started less balls than we're supposed to
    // have we'll send a new one on.
    if( isMaster )
    {
        if( myState->getNumStarted() < numBalls )
        {
            startBall();
        }
        // Else we're finished.
    }

    while(haveMoreEvents() == true) // There is at least one event to process.
    {
        const PingEvent *eventReceived = dynamic_cast<const PingEvent*>(getEvent());
        ASSERT( eventReceived != 0 );
        myState->ballReceived();

        // If this is a master object (someone capable of starting balls)
        if( isMaster )
        {
            // and this is NOT a ball we started already?
            if( eventReceived->getOwner() != getName() )
            {
                // Then we'll send it on.
                sendEvent( eventReceived->getOwner() );
            }
            // If we are not a master we could not have started the ball so send it on.
        } else
        {
            sendEvent( eventReceived->getOwner() );
        }
    }
}

void PingObject::sendEvent( const string &owner )
{
    IntVT ime recvTime = dynamic_cast<const IntVT ime*>(&getSimulationTime()) + 1;
    Event *newEvent = new PingEvent( getSimulationTime(),
                                      recvTime,
                                      this,
                                      sendTo,
                                      owner );
    sendTo->receiveEvent( newEvent );
    PingObjectState* myState = static_cast<PingObjectState*>(getState());
    myState->ballSent();
}

Figure 3.2: Implementation of important PingObject functions.
3.4 Configuration

To set run time configuration options, users make changes to a configuration file. The details of these options are left to a later chapter. In most cases, the name of the option is self-explanatory. The primary configuration option is the type of simulation manager to use. A simulation manager is responsible for coordinating all activity in the kernel such as scheduling of events and handling communication. The SimulationManager class implements the simulation manager. More detail about the kernel and different types of simulation manager implementations is presented in Chapter 5. There are currently three simulation managers, namely: a sequential simulation manager, a Time Warp simulation manager, and a Threaded Time Warp simulation manager. Appendix A contains example configuration files for each of the three simulation types.

3.5 Summary

This chapter discussed the requirements for creating simulation models and how simulation models interact with the kernel. The WARPED kernel provides an interface to create a simulation model that can then be executed with many different configurations. An example simulation model named PingPong shows how the API can be used to develop simulations. The next chapter will provide more details about how the kernel actually works.
Chapter 4

WARPED Kernel Interface

4.1 Introduction

The development of a simulation model using the WARPED API consists of defining simulation objects, events, and states. These definitions merely provide the definition for the simulation; the WARPED kernel implements the actual execution control structures of the simulation. The primary component in the kernel is the simulation manager and is represented in the kernel by the abstract class named SimulationManager. This class performs several functions including: (i) causing the simulation objects to process events, (ii) organizes and manages internal event lists, and (iii) transparently handles the transmission of event and other information between the system components. When the simulation kernel initializes, a few tasks must occur before the simulation manager takes over. Figure 4.1 shows the basic flow for the execution of a simulation. The kernel reads the configuration file along with any command line parameters. Based on the configuration, the kernel then allocates, configures, and initializes the SimulationManager class. The configuration file determines the type of simulation manager to allocate. From there, the simulation manager takes control of the simulation. Thus, the WARPED v2.x simulation kernel is structured into two distinct parts: (i) a set of common operations for all simulations and (ii) a set of specific simulation managers that deploy unique execution engines for the simulation model. In this thesis, the first part will be called the WARPED simulation core and each instance of the second part will be called the WARPED simulation manager. The simulation core is common across all simulation managers. The remainder of this chapter focuses on the description of
the simulation managers and in particular on the details of the Time Warp synchronized parallel simulation manager.

The various WARPED simulation managers can execute the simulation model in unique ways. However, they must all provide basic functions that the simulation core will use to create, destroy, and initiate execution of a simulation model. Within a simulation manager, the most important function to define is the `simulate` function. The simulation will spend almost all of its time within this function once it is called. As can be seen in Figure 4.1, `simulate` is only called once. The differences between a sequential and parallel simulation originate in `simulate`.

In addition to providing the `simulate` method to control the execution of a simulation model, a simulation manager must also provide several services for a running simulation. For example, one critical service that must be provided is the transmission of events between simulation objects (in this case transmission may be simply the insertion of the event into a list; alternatively it may require the packaging of

![Figure 4.1: Execution flow of a simulation.](image)
the event into a message and its transmission to a remote compute node in a Beowulf cluster). When a simulation object sends an event to another simulation object, it calls the `receiveEvent` function from the `SimulationObject` class. This function then calls the `SimulationManager::handleEvent` function. From there, the simulation manager ensures that the proper simulation object receives the event. The `SimulationManager::getEvent` function provides simulation objects with the next event to be processed. When a simulation object needs an event to process, it calls this function. This is necessary because the event list management occurs in the simulation manager and not within the simulation object.

The **WARPED** v2.0 system has three distinct simulation managers each implementing a derivation of the `SimulationManager` class. These simulation managers are:

- **SequentialSimulationManager**: Executes a sequential simulation as a single thread executing on one processor.
- **TimeWarpSimulationManager**: Executes a parallel simulation using the Time Warp synchronization protocol as a collection of single threaded processes on multiple processors.
- **ThreadedTimeWarpSimulationManager**: Executes a parallel simulation using the Time Warp synchronization protocol as a collection of multi-threaded processes on multiple processors.

Table 4.1 summarizes the number of processors and threads for the various simulation managers. The remainder of this chapter discusses the architecture of each of these simulation managers.

### 4.2 Sequential Simulation

The primary functionality of the *sequential simulation manager* is to process events in non-decreasing timestamp order. When running a sequential simulation, only one instance of the sequential simulation manager...
CHAPTER 4. WARPED KERNEL INTERFACE

4.2. SEQUENTIAL SIMULATION

Figure 4.2: Structure of a sequential simulation.

will exist. All simulation objects are located on this simulation manager. Figure 4.2 shows the structure of a simulation with the sequential simulation manager.

The EventSet class represents the event list within the simulation manager and is the main data structure. An efficient implementation of the EventSet class is critical for the performance of the sequential simulation manager. The EventSet must handle event insertion and removal as efficiently as possible. There are currently two EventSet implementations for the sequential simulation manager: SingleLinkedList and SplayTree. Both data structures maintain the events in non-decreasing time-stamp order. Removal from the data structures is a simple operation when the data structure is already sorted. However, insertion requires the event to be placed in the correct position of the data structure.

Within the simulate function, the EventSet determines the next event to process by looking at the top event. The executeProcess function is called for the simulation object that processes that event. The simulation object will then remove the event from EventSet within executeProcess by the getEvent function that is provided by the SimulationManager. This continues until there are no more events to process or until the time-stamp of the next event to process is greater than a designated termination time. Figure 4.3 shows the flow of the simulate function. Communication between simulation objects occurs through the handleEvent function. For the sequential simulation manager, this function simply inserts the event into the event set manager.

The sequential simulation manager was developed by other researchers at UC and is fully functional. However, the Time Warp simulation manager discussed next required significant work to become fully functional. Its design and performance analysis is the principle focus of this thesis.
CHAPTER 4. WARPED KERNEL INTERFACE

4.3 TIME WARP SIMULATION

Figure 4.3: Execution flow of a sequential simulation.

4.3 Time Warp Simulation

Before explaining the internals of the Time Warp simulation manager, it helps to understand how a parallel simulation will run using WARPED. To execute simulation objects in parallel, they have to be distributed to different processors. Similar to WARPED v1.x, each processor used in the simulation can contain multiple simulation objects clustered together. Each processor contains one or more Time Warp simulation kernels that coordinate the simulation objects on that processor. Because the Time Warp simulation manager implements the Time Warp mechanism, it is much more complex than the sequential simulation manager. Figure 4.4 illustrates the structure of a parallel simulation using the Time Warp simulation managers. There are four processors in the example, each with one Time Warp simulation manager that contains multiple simulation objects and several support functions for managing a Time Warp simulation. In the remainder of this section, a description of the function and structure of the code for each of these support functions is presented.

The event list manager provides support for storing, sorting, and accessing all of the events for the simulation objects contained in the simulation manager. A base TimeWarpEventSet class allows for different data structures to be used for managing events. When a simulation object experiences a rollback, the event list manager performs the necessary event management on the input queue. Unlike the event list manager
used for the sequential simulation manager, the Time Warp simulation manager requires two functions for obtaining events from the event list. The first function returns the lowest time-stamped event for a specific object. The second function returns the lowest time-stamped event for all objects. To accomplish this in the current implementations of the event list managers in the kernel, the event list manager maintains separate containers for each simulation object within the general event set manager class. The need for obtaining the next event for a specific simulation object exists because the simulation manager does not require simulation object to be synchronized. The scheduling manager component exists so that the simulation manager can process simulation objects in any desired order. The default scheduling manager is lowest time-stamp first order. However, the base SchedulingManager class can be derived to perform any type of scheduling desired. The name of the original event set manager implemented is DefaultTimeWarpEventSet. It was created to get a working event list and was not intended to be an optimal solution.

Figure 4.4: Structure of a parallel Time Warp simulation.
The output manager provides output queues for each simulation object. Output queues are needed for the Time Warp simulation to send anti-messages to correct causality errors that occur throughout the simulation. Note that the simulation objects do not contain the output queues. Instead the simulation manager controls all of the output queues in a single instance of the OutputManager class. Thus, when a rollback occurs, the output manager handles the sending of anti-messages. An abstract base class defines the output manager interface to allow for different event cancellation techniques to be used. In the initial implementation of the Time Warp simulation manager only aggressive cancellation was supported by an output queue type named AggressiveOutputManager. As the implementation evolved additional cancellation mechanisms were implemented. These implementations are described later in this thesis.

The state manager performs state saving and restoration operations for all simulation objects contained by a simulation manager. As with the event set manager and output manager, the state manager manages all state queues (rather than the simulation objects). State queues are needed to save and restore the state of simulation objects to support rollback. Thus, when a causality error occurs, the state manager can restore the simulation object to a previous state. The saveState function for the StateManager class is called every time the simulation manager processes an event. This organization hides the state management activities from the simulation model developer and allows the definition of new derived state manager classes that can provide other state management solutions (e.g., periodic state savings [34, 35], incremental state savings [36, 37], and so on). For example, a PeriodicStateManager class has been developed that performs periodic state savings.

The GVT manager is another helper object that supports the implementation of global virtual time (GVT) algorithms for a simulation. By deriving the base GVTManager class, researchers can implement many different GVT estimation algorithms [11, 12, 38, 39]. The simulation manager calls the GVT manager any time it sends an event on the network so that the GVT manager can keep track of all network messages. Some GVT algorithms also help determine termination [40]. For WARPED however, the termination manager is used to determine when a simulation is complete. In the current implementation, both the GVT and the termination manager are required for termination to function properly. The issues dealing with the implementation of the MatternGVTManager class that uses Mattern’s GVT algorithm are described in the next chapter. The TokenPassingTerminationManager class uses a basic token passing algorithm to de-
CHAPTER 4. WARPED KERNEL INTERFACE 4.3. TIME WARP SIMULATION

terminate if a simulation is complete. The termination manager uses the *sticky flags* concept from Mattern’s termination algorithm to check if a simulation manager was idle since the last token round [40]. When all simulation managers are idle for two consecutive token rounds, the simulation is complete.

The *communication manager* is used to send events and other messages between simulation managers. Information about the other processors, such as the IP address and port number, is maintained here. The communication manager has a base class named `CommunicationManager` to allow the kernel to use different underlying communication protocols while maintaining an easy to use interface for communication. In the current system both MPI [18] and TCP/IP communication layers are supported. The communication manager hides the complexity of sending and receiving messages through sockets so that when the sim-

![Figure 4.5: Execution flow of the Time Warp simulate function.](image-url)
ulation manager sends an event, only one function has to be called from the communication manager. It also handles other types of messages such as GVT and start up messages. When the simulation manager receives messages, it routes them to the proper manager to handle the message. For example, when the communication manager receives a GVT message, it routes the message to the GVT manager.

Now that the data structures have been explained, the actual processing of events can be presented. Figure 4.5 shows the flow of the simulate function. As long as the termination manager does not indicate that the simulation is complete, the simulation manager attempts to execute any available events and checks the communication manager for incoming messages. The simulation manager processes events by calling the executeObjects function. If there are no events to process, then the simulation manager notifies the termination manager that it is idle and continues checking for any incoming messages from the communication manager. If there are events to process, then the simulation manager updates the time of the simulation object that it will process next, calls on the state manager to save the state of the simulation object, and then calls on the simulation object to process the event.

The state saving and event processing contain some subtle but important characteristics. When the simulation manager calls the executeProcess function for the simulation object, the simulation object may have several events to process at its current simulation time. In this case, the executeProcess function will execute all of those events before exiting. This is necessary to ensure that when the state manager saves a state for the object, it saves the state before the simulation object processes any events at that time. The state manager never saves state between events processed on the same simulation object with the same time-stamp. When a rollback occurs for a simulation object, it will process events with time-stamps greater than or equal to the rollback time again. If the state manager saved the restored state in between events processed with the same time-stamp, then the simulation object will process events again even though they have already previously modified the state.

As an example of the challenge of state saving and the processing of events at the same simulation time, consider the processing of events in Figure 4.6. In this example, the state simply keeps track of how many events a simulation object has processed. The state manager saves the state in the middle of processing four events at time 51. In Figure 4.7, a rollback to time 53 occurs and the state manager restores the state to time 51. The simulation object then processes all of the events at 51 again, but the state starts as if the simulation
object has already processed two events. The end result is that the state is incorrect.

After the simulation object processes all events at its current simulation time, the simulation manager performs some coordination duties. It calls the GVT manager to start GVT estimation if necessary. The simulation manager then checks to see if any communication messages from other simulation managers have arrived. If so, then it handles these messages and continues to check for more events to process.

Another important aspect of the Time Warp simulation manager is the sending and receiving of events. Whereas the `handleEvent` function in the sequential simulation manager simply inserts the event in the event set manager, the Time Warp simulation manager must check to see if the event has a time-stamp less than or equal to the last event processed for the receiving simulation object. If so, then the simulation manager triggers the rollback process. Figure 4.8 shows the flow of the `rollback` function. The first task in the process is to call on the state manager to determine which state to restore. Due to periodic state saving, the simulation manager must pass the time of the restored state to the event list manager. The rollback process rolls back the event list to the restored state time because that is where the simulation object

![Diagram of processed events](image)

**Figure 4.6:** Processing of events with incorrect state saving.

![Diagram of events processed after rollback](image)

**Figure 4.7:** Processing of events after the state has been restored after a rollback.
will reprocess events. The simulation manager then uses the output manager to rollback the output events using the rollback time, not the restored state time. This is when the output manager sends the anti-messages. The next phase is called the *coast forward* phase and is necessary with periodic state saving. Because the output manager only sends anti-messages for events sent after the rollback time, any events sent from the restored state time up to the rollback time will already exist. Sending them again will cause duplicates to exist. The simulation manager suppresses event sending during the coast forward phase for the simulation object. After the coast forward phase ends, the simulation resumes normal execution.

There was a problem with rollbacks at one point in the development process. The event list manager was sorting events based on receive time only. When a simulation object had a rollback, the event list manager transferred events from the processed list to the unprocessed list and sorted the events. However, the order was sometimes different for events with the same receive time. This caused simulation states to be incorrect. The solution involved uniquely identifying events by the simulation object *sender ID* and the *event ID*. Each simulation object in the simulation has a unique identification using a tuple of the simulation manager ID and the simulation object ID. When a simulation object creates and sends an event, it generates an event ID that is unique only for that simulation object. Instead of sorting just by receive time, the event list manager sorts by receive time first, then event ID, and finally by simulation object ID.

The handling of anti-messages required extensive work to get a parallel simulation to run at all. Fur-
CHAPTER 4. WARPED KERNEL INTERFACE

4.3. TIME WARP SIMULATION

ther complicating matters, the simulation manager was not properly handling anti-messages after the early implementation attempts. When the output manager sent out anti-messages, it grouped them together even though the simulation manager needed to send them to different remote simulation managers. The simulation manager only examined the first anti-message to determine which simulation manager received all of the anti-messages. To fix this problem, the simulation manager reorganized the anti-messages so that it sent the proper anti-messages to the proper simulation managers. The other problem was the way the simulation manager generated anti-messages. The only information sent as an anti-message was the event ID of the corresponding positive event. There were several problems with this implementation. The first and critical issue was that an event ID was only unique to events generated from one simulation manager, not the entire simulation. It was entirely possible for two events in an event list manager to have the same event ID. This meant that the event ID was not sufficient to be able to determine which event to cancel. The other problem was that even if event IDs were unique across the entire simulation, determining which simulation object contained the event corresponding to that event ID would be very inefficient. The event list manager would have to examine every simulation objects’ event list. The original intent was to include the minimal amount of information, but not enough was included. One solution would be to send the entire event as the anti-message. However, if the user defines many data members in the event class, then the anti-messages become unnecessarily large. The only information needed to distinguish events is the sender simulation object ID and the event ID. These were included in the NegativeEvent class along with two additional convenience fields. A NegativeEvent contains the receiver simulation object ID to allow the event list manager to quickly find the positive event. The simulation manager uses the receive time included in a NegativeEvent to determine if the simulation object has already processed the positive event. If it has, then the simulation object performs a rollback before cancelling the event.

Another complication with the initial Time Warp implementation was GVT estimation and fossil collection. The default method of fossil collection involves fossil collecting after the GVT has been updated. The initial implementation did not result in functional fossil collection when there were multiple simulation objects on a simulation manager. The problem involved instances where simulation objects sent events to other simulation objects on the same simulation manager. These events are called local events. The order of fossil collection was originally: states, output events, and then input events. To minimize memory usage, the
event list manager and output manager do not duplicate local events. Instead, the event list manager creates a second pointer to a local event. A pointer in the output manager and a pointer in the event list manager point to the same event. With the original fossil collection method, the output manager fossil collected events, but the pointers to those fossil collected events would still exist in the event list manager. When the event list manager examined those fossil collected events, events that were already reclaimed would be accessed, resulting in errors. To fix this problem, the output manager stores separate lists, one for local events and one for remote events (events sent to other simulation managers). When the output manager performs fossil collection, it simply removes the local events from the list and reclaims the remote events. The event list manager then reclaims all events in its lists.

Another issue with fossil collection involved periodic state saving. There must always be at least one state in the state queue for every simulation object so that the state manager can restore a state after a rollback. Events can cause a rollback to a time equal to the GVT but not to a time less than the GVT. However, with periodic state saving, the most recent state after the GVT may be well after the GVT. Therefore, the state manager must not fossil collect the state saved at the greatest time less than GVT in order to handle rollbacks to the GVT. This caused problems because the output manager and the event list manager were performing fossil collection at the new GVT time. When the state manager restored the state before the GVT, the coast forward events did not exist, causing the simulation to be incorrect. The solution to this problem involves fossil collecting at the restored state time for each simulation object. This is why the order of fossil collection is important. The state fossil collection determines the time at which the output manager and event set manager can fossil collect. The output manager performs fossil collection before the event list manager because it will reclaim any event with a time less than the GVT. The output manager is able to remove the necessary events from the lists without reclaiming them.

4.4 Threaded Time Warp Simulation

The Threaded Time Warp simulation manager is very similar to the Time Warp simulation manager in that all of the components described are also used in the Threaded Time Warp simulation manager. Another researcher [19] performed the work for the Threaded Time Warp simulation manager at the same time as the development of the Time Warp simulation manager. The Threaded Time Warp simulation manager is ac-
tually derived from the Time Warp simulation manager. However, there are several key differences. While the Time Warp simulation manager uses only one thread to handle all event processing as well as global operations such as GVT estimation, I/O, and communication, the Threaded Time Warp simulation manager handles these tasks using multiple threads. One thread performs GVT calculations, handles communication, and manages the other threads. This thread is called the manager thread. All remaining threads are called worker threads and only perform event processing. Figure 4.9 highlights the operation of the manager thread. The first task the manager thread performs is to spawn a designated number of worker threads. The manager thread then performs a loop until the termination manager indicates the simulation is complete. Within the loop, the manager thread receives communication messages and checks the GVT to see if the GVT manager should start GVT estimation. The manager thread then calls the `executeObjects` function to process events. In some cases, the manager thread may actually process events if it is configured to do so. Otherwise, this is where the worker threads process events. The manager thread then transmits any events in the send buffer. At this time, if any of the worker threads or the manager thread have processed events, then the manager thread sets the simulation manager status to active. Figure 4.10 illustrates the operation of the worker thread. When not processing any events, a worker thread increments a counter to let the manager thread know how many threads have processed events.

The Threaded Time Warp simulation manager requires several data structures to handle local communication: the received messages queue, the object queue, and the send buffer. All three are shared memory data structures and perform atomic operations to ensure that only one thread modifies them at a time.

The received messages queue stores all events sent between simulation objects. This includes events sent from other simulation managers as well as the local simulation manager. The events remain in the queue until the manager thread locks the receiver simulation object, preventing other threads from accessing the simulation object. The worker thread then inserts the events into the simulation object’s event list and processes the events.

The object queue holds all simulation objects that the simulation manager contains. When a thread is not performing any work, it examines the object queue for the simulation object containing the lowest available time-stamped event. When the object is found, it is locked and the thread processes the events for that object.

Worker threads use the send buffer queue to store events bound for other simulation managers. The
Figure 4.9: The manager thread process.
manager thread empties the buffer periodically. Only the manager thread accesses the communication manager and sends events to other simulation managers. When sending an event to another simulation object on the same simulation manager, the worker thread inserts the event into the received messages queue, not the send buffer.

These data structures are needed because threads processing simulation objects may access other structures such as the input queue and the communication manager at the same time the manager thread is using them. For example, when an event arrives from another simulation manager, the manager thread could attempt to insert the event into a simulation object’s input queue at the same time that the simulation object in another thread is removing events from the input queue. This could especially present problems when dealing with anti-messages, as one thread could access an event that the other thread is cancelling.
Chapter 5

Time Warp Optimizations Implemented

5.1 Introduction

To improve performance, several Time Warp optimizations were implemented into the Time Warp portion of the WARPED kernel. The primary areas targeted for improvement include: event cancellation, state saving, GVT estimation, fossil collection, and communication. This chapter discusses the implementation of these optimizations in the kernel.

5.2 Output Events

One of the simplest techniques to achieve event cancellation for rollback recovery is aggressive cancellation. With aggressive cancellation, a simulation object sends anti-messages immediately after a rollback occurs. The simulation object removes any event sent with a send time greater than or equal to the rollback time from the output queue and immediately sends a corresponding anti-message.

In WARPED, the output events for simulation objects are managed by the output manager. Within the output manager, there is a container for each simulation object to hold the output events. Initially, a C++ standard library vector stored all of the output events. The vector never needed sorting because a simulation object can only send events in non-decreasing order. With this technique, the output manager examines the container from the back and removes events until it reaches events with times less than the rollback time. This results in a faster rollback process than examining the entire container. The output manager
sends the removed events to the simulation manager through a call to cancelEvents. Here, the simulation manager generates anti-messages (implemented as the NegativeEvent class) and organizes them by receiving simulation objects. The simulation manager then sends the anti-messages through the communication manager to other simulation managers if necessary, or handles them directly by sending them to the event list manager.

### 5.2.1 Lazy Cancellation

In some cases, recomputation of events after rollback results in the retransmission of the same output events that were sent by the premature computations. When this happens, the aggressive cancellation was an unnecessary overhead. In Lazy Cancellation, a simulation object only sends anti-messages after the recomputation shows that the original output event is incorrect [9]. As in aggressive cancellation, the simulation object removes any event with a send time greater than or equal to the rollback time from the output queue. However, the simulation object does not immediately send anti-messages. Instead, when a simulation object generates an output event through the recomputation of an event, it compares the event to those in the output queue. If the event already exists in the output queue, then the simulation object does not send the event. Once a simulation object changes simulation time, it sends anti-messages for any events that it did not regenerate (those with a send time less than the simulation time that were not sent again).

Figure 5.1 shows an example of how lazy cancellation works. A rollback occurs to time 20, so the simulation object will compare all of the original output events with a send time greater than or equal to 20 to any regenerated output events. The simulation object regenerates events sent at times 21, 24, 27, and 29,
so they remain in the output queue, and the simulation object does not transmit them. When the simulation object computes the event at time 26, there is no way the simulation object will regenerate the event at time 25, so it removes the event from the output queue and sends it as an anti-message. The event generated at time 26 is not a regenerated event so the simulation object inserts the event into the output queue and sends it as a regular positive event. The event at time 28 behaves the same as the event with time 26.

The characteristics of a simulation model affect whether it will perform better or worse using lazy cancellation [9, 20, 41]. When a simulation object has many events that affect the generation and sending of other events, then aggressive cancellation will perform better. If a simulation object mostly has events that are independent of each other, then lazy cancellation will perform better than aggressive cancellation. However, in some cases, lazy cancellation is able to perform computation in less time than the critical path execution time [42, 43]. This happens because it is possible that the premature computations actually produce the correct output events.

The WARPED kernel defines the LazyOutputManager from the base output manager class to implement lazy cancellation. The underlying storage of output events remains the same as aggressive cancellation, but the lazy output manager requires several additional data structures. In addition, the lazy cancellation manager requires modification to the underlying output container. The main data structures for the lazy output manager are the lazy queues, one for each simulation object on the simulation manager. During a rollback, the lazy output manager removes events from the output container for a simulation object, and inserts them into the simulation object’s lazy queue. Throughout the entire simulation, any time the simulation manager inserts an event into the output manager, the lazy output manager examines the lazy queue for the simulation object. Figure 5.2 shows the process undertaken. The first task is for the lazy output manager to determine the events that the simulation object has not regenerated. The lazy output manager removes any event with a send time less than the event currently being inserted from the lazy queue and sends it to the cancel events queue. The lazy output manager then compares the event to insert to all events in the lazy queue until a match is found by using the user defined eventCompare method. Event IDs are not used in the comparison because a regenerated event will always contain a different event ID. If there is a match, then the lazy output manager removes the event from the lazy queue and inserts it into the output container. The lazyCancel function return value indicates that the simulation manager should not send the event
and the lazy output manager discards the regenerated event. The lazy output manager then reinserts the original event into the output container in case it may need to send an anti-message for that event, as the event ID needs to match that of the event to cancel. If there is not a match, then the lazy output manager inserts the event and the lazyCancel function return value indicates that the simulation manager should send the event. If any events remain in the cancel events queue after the comparison phase is complete, the lazy output manager sends an anti-message for each of them.

Though the basic implementation was rather straightforward, some challenges had to be resolved to make lazy cancellation functional. The main challenge came when simulations were found to be terminating too early with lazy cancellation. The problem was isolated by removing the termination manager functionality. When this was done, the simulation did not terminate but rather stopped making forward progress. It was determined that when this occurred, at least one lazy queue was not empty. This meant that the lazy output manager was not cancelling events that it should have been cancelling. The only way that

![Diagram of the lazy cancellation process](image-url)

Figure 5.2: The lazy cancellation process.
the lazy output manager could determine that an event was not regenerated was by examining the simulation object’s LVT and the send time of the event. However, if the simulation object never sent any additional events after a rollback, the LVT would never increase and it would never send any anti-messages. To fix this problem, the simulation manager flushes all lazy queues when there are no more events to process on the simulation manager. This prevents the simulation from stalling out as the lazy output manager will send all incorrect events at some point.

5.2.2 Dynamic Cancellation

As stated earlier, lazy cancellation may or may not perform better than aggressive cancellation depending on what percentage of events are regenerated after a rollback. The higher the percentage, the better lazy cancellation performs. Studies have demonstrated that choosing the optimal strategy statically is complicated and requires full knowledge of the simulation model [44]. Additional research has shown that the more optimal cancellation strategy can vary from simulation object to simulation object within a simulation and even within a simulation object throughout the simulation execution time [45, 46].

Because of these problems, Rajan and Wilsey developed an algorithm that attempts to dynamically select the more favorable cancellation strategy for each simulation object [10]. The name of this optimization is called Dynamic Cancellation. By choosing the optimal strategy at runtime, Rajan and Wilsey expect to improve the overall performance of the Time Warp synchronized simulation. In their work, Rajan and Wilsey develop a dynamic solution that monitors each simulation object to see how often their prematurely computed output events are correct. Whenever a simulation object is producing correct premature output events, their algorithm sets the simulation object to use lazy cancellation; whenever the simulation object is not producing correct premature output events, their selection switches to use aggressive cancellation at that simulation object. Each simulation object monitors and selects the cancellation strategy that best suites its own behavior.

To implement Dynamic Cancellation, Rajan and Wilsey added an output event comparison routine to aggressive cancellation that is similar to the comparison performed by lazy cancellation. While this adds additional overhead to simulation objects using aggressive cancellation, it allows Rajan and Wilsey to count how often the simulation object produces correct output events during precomputations. Rajan and Wilsey
then define the output event \textit{Hit Ratio} as the total number correct premature output events against the total number of premature output events. More formally, \textit{Hit Ratio} is defined as:

\[
\text{HitRatio} = \frac{\text{LazyHits}}{\text{TotalComparisons}}
\]

To help ensure that the algorithm responds to the recent behavior of the simulation object, only the most recent history of the output comparisons is measured. Once the \textit{HitRatio} is computed, a thresholding function uses it to determine which cancellation strategy to use. A higher hit ratio favors lazy cancellation and a lower ratio favors aggressive cancellation. Rajan and Wilsey use a thresholding function to trigger switching between the two cancellation strategies. They explored a number of configurations and considered versions with one, two, and even three distinct thresholds. With one threshold, any Hit Ratio less than the threshold will cause the simulation object to use aggressive cancellation and any Hit Ratio greater will cause the simulation object to use lazy cancellation. When there are two threshold values, a dead zone between the two thresholds exists to help prevent thrashing. When the Hit Ratio is below the first threshold, a simulation object uses aggressive cancellation. When the Hit Ratio is above the second threshold, a simulation object prefers lazy cancellation. Any Hit Ratio in the dead zone leaves the cancellation strategy unchanged. The third possibility is to have three thresholds. This operates in the same way as two thresholds except that a third threshold is added that is beneath the lower threshold of the two threshold case. When the Hit Ratio falls below this third (lowest) threshold, a simulation object permanently switches to aggressive cancellation for the remainder of the simulation.

Initially, with Dynamic Cancellation, all simulation objects use aggressive cancellation. When a rollback occurs, the simulation object immediately sends anti-messages. However, it does not reclaim the positive events that generated the anti-messages. Instead, the simulation object adds the events to the lazy queue. Any time a simulation object inserts an event into the output queue it compares the event to the events in the lazy queue and records a lazy aggressive hit or miss. This is done to determine how well the simulation object would have performed if the simulation object used lazy cancellation. When a simulation object is using lazy cancellation, dynamic cancellation performs the same with the only addition being the calculation of the Hit Ratio. The overheads involved with dynamic cancellation are essentially the same as lazy cancellation.

In the \textsc{Warped} kernel, a class named \texttt{DynamicOutputManager} implements dynamic cancellation.
The class derives the lazy output manager because many of the data structures and processes required for dynamic cancellation involve lazy cancellation. Some of the additional data structures include containers for the past comparison results and the current strategy used by each simulation object. The dynamic output manager uses the lazy output manager compare function to determine if a simulation object regenerated an event. Figure 5.3 shows the general outline of the Dynamic Cancellation comparison process.

By default, there are two thresholds that determine the cancellation strategy that simulation objects use. The lower threshold to switch from lazy to aggressive cancellation is set at 0.2 and the threshold for aggressive to lazy cancellation is set at 0.5. The default filter depth is 16. All of these values were selected based on the original researchers’ claim that these provided the best performance. At this time, these are hard coded values, but they could easily be turned into run-time configuration options by adding the necessary functionality into the configuration parser.

One of the uncertainties with implementing dynamic cancellation was recording misses. Lazy aggressive hits are easy to measure as a simulation object has made only one comparison. However, the dynamic output
manager records lazy misses once the simulation object LVT has passed the send time of events in the lazy queue. This can sometimes result in the lazy aggressive misses being clustered together. The effects of this clustering are not entirely known, but the only other alternative would be to record all the misses as one. This however, would bias towards lazy cancellation, as the dynamic output manager would record a large number of misses as only one miss.

Overall, implementing dynamic cancellation was not much of a challenge once lazy cancellation had been implemented. The primary addition to the class involved the switching between lazy and aggressive cancellation. This task was relatively straightforward. When switching aggressive to lazy cancellation, the output manager has already sent the anti-messages, so nothing needs to happen. When switching lazy to aggressive cancellation, the dynamic output manager needs to send all of the events in the lazy queue as anti-messages.

### 5.2.3 One Anti-Message

The typical method to cancel erroneous events is to send an anti-message for each incorrect (or premature) event. As can be seen in an example shown in Figure 5.4, one simulation object can generate several anti-messages that another single simulation object receives. The simulation object sends a total of seven anti-messages. If a simulation object has to send anti-messages over a communication network, each message incurs overheads. Even if a simulation object aggregates the anti-messages together, the larger size of the message will hurt performance. An optimization to reduce the communication overheads involves sending...
only one anti-message for each simulation object receiving anti-messages from a simulation object that was rolled back. This technique is possible because when a simulation object rolls back, it sends anti-messages for every event sent from the simulation object after the rollback time. The lowest time-stamped anti-message bound for another simulation object is the only anti-message needed to cancel any erroneous events. When a simulation object receives the single anti-message, it cancels all events in its input from the simulation object that sent the anti-message with times greater than or equal to the anti-message. Figure 5.5 shows how the one anti-message optimization works with the same example from Figure 5.4. The simulation object only requires two anti-messages with the optimization.

By sending only one anti-message for each simulation object, the simulation object can reduce the communication overhead necessary for cancelling events. If a simulation object was previously aggregating anti-messages, one anti-message will reduce the size of messages transmitted. If a simulation object sent each anti-message individually over a network, the one anti-message optimization will reduce the amount of network traffic.

When a simulation object has a rollback in WARPED, the output manager sends all output events to be cancelled for that simulation object to the simulation manager. The simulation manager is then in charge of generating anti-messages in the form of the NegativeEvent class and responsible for determining to which simulation managers it sends anti-messages. The process for doing this is to go through all of the events the output manager provides and separating them according to the simulation object receiver. The anti-messages for one simulation object are then sent together in one network message to the simulation object.
manager that contains the receiver. If the simulation object is on the local simulation manager, then the simulation manager handles the group of anti-messages without using the communication manager.

On the receiving end, the simulation manager passes anti-messages to the event list manager. If necessary, the receiving simulation object will rollback. The event list manager then finds the corresponding positive event by comparing the sender simulation object IDs and event IDs of the positive event and anti-message. The method for finding the event in the event list manager is implementation specific. Using a contiguous storage container such as an array, a linear search from either the front or back would be required to find each event. This could in the worst case result in a full traversal of the array for each anti-message. Other data structures may only require a logarithmic time to locate the positive event.

Implementing the sending of only one anti-message was straightforward. Instead of sending the entire group of anti-messages, the simulation manager sends the anti-message with the minimum time-stamp. The more involved changes occurred in the event list manager where the anti-message is actually handled. The receiving of anti-messages required more modifications. Once the event list manager finds the positive event that corresponds to the anti-message, the rest of the event list has to be examined. Any time an event is from the same sender, the event list manager removes it from the event list. This process assumes that the event list is already sorted. To implement these changes in the event list implementations, new event list manager classes were derived with the handleAntiMessage function being the only function overridden by the new classes.

This optimization is currently working with aggressive cancellation. However, there were issues when working with lazy (and therefore dynamic) cancellation. The goal of lazy cancellation is for a simulation object to only cancel the incorrect premature events. If using the one anti-message optimization, as soon as the lazy output manager sends a single anti-message, it cancels the remaining events in the lazy queue, even if they are correct and do not need to be cancelled. This negates the reason for using lazy cancellation. The only case where this would be beneficial is when one non-regenerated event leads to a simulation object not regenerating most of previously sent events. However there would still be an issue when a simulation object sends two events with the same send time, same receiver, and different receive times. If the simulation object regenerates the first event, the event will correctly remain in the receiver’s input queue. If the simulation object does not regenerate the second event, then it will send an anti-message and cancel out the event that
it just processed. To avoid this, the lazy output manager would need extra overhead. One possibility would be to process the first event again. Another would involve keeping track of the events on the receiving end to make sure that the event list manager does not cancel the correct one. Both add some overhead that may negate the gains of using the one anti-message optimization.

5.3 State Saving

To recover from causality errors, a simulation object must save its state. The most basic form of state saving is to save the state of a simulation object every time it changes in response to the processing of an event. The rollback process could then restore the state of the simulation object directly before an incorrect computation occurred. However, saving the state so often would result in large memory usage. To reduce the memory required for saved states, a simulation object can save states periodically after it has processed a specified number of events [34]. The number of processed events between state saves is called the *checkpoint interval*. One effect of periodic state saving is that the restored state is not the correct state from which the straggler can start computing. To get the correct state, a simulation object must reprocess events. The process of moving forward from the restored state to the required state at the rollback time is called *coasting forward*. A simulation object does not send any output events generated during the coast forward phase; it only updates its state. Figure 5.6 shows periodic state saving and coasting forward. There are two main costs associated with periodic state saving [16]. The first is the actual saving of the states.

![Periodic state saving and coast forward phase.](image)

Figure 5.6: Periodic state saving and coast forward phase.
As the checkpoint interval increases, the amount of memory required for and the time spent saving states decreases. The second cost involves the coast forward phase. With a larger checkpoint interval, the time required for executing events during the coast forward phase increases.

### 5.3.1 Adaptive Period

The optimal checkpoint interval can vary from simulation object to simulation object within a simulation and can also change for a single simulation object throughout a simulation’s execution [16]. Instead of statically determining a checkpoint interval, a simulation can dynamically adjust it for each simulation object. To determine the optimal checkpoint interval for time performance, the costs of saving states and coasting forward have to be balanced. Algorithms for dynamically adjusting the checkpoint interval have been developed by Lin [13], Palaniswamy [14, 47], and Ronngren [15]. In these algorithms, an analytical model derives the formula used for determining optimal checkpoint intervals. These formulas tend to be fairly complex and can involve a significant amount of overhead. A heuristic algorithm developed by Fleischmann and Wilsey attempts to provide a lightweight yet effective checkpointing solution [16]. In this heuristic, a formula called the cost function is defined as the sum of the cost of coasting forward and the cost of saving states. Figure 5.7 illustrates these costs and their sum; the minimum of the cost function is the optimal checkpoint interval.
The heuristic algorithm works as follows. All simulation objects begin the simulation with a checkpoint interval of one. A simulation object recalculates its checkpoint interval after it has processed an $N$ number of events. The simulation object compares the new value for the cost function to the old value. If the new value is not significantly greater than the previous value, then the simulation object increments the checkpoint interval by one. Otherwise, the simulation object reverses the adaptation direction and decreases the checkpoint interval by one. The interval adjusts in the same way for the reverse direction: the interval decreases as long as the new cost is not much greater than the old cost. This acts as an inexpensive way to keep the cost function near its minimum and the checkpoint interval near its optimal value. The maximum value for the checkpoint interval is 30 based on empirical observations by Fleischmann and Wilsey [16]. A simulation object sets the checkpoint interval to this maximum value if it has not rolled back in the last observation period.

In Warped, the state manager base class is in charge of state saving for all simulation objects on a simulation manager. The primary functions provided by the class are `saveState` and `restoreState`. The simulation manager calls the `saveState` function every time before a simulation object processes events. The `PeriodicStateManager` implements periodic state saving by only saving the state of a simulation object after it has processed a specified number of events. The period is maintained within the `PeriodicStateManager` class.

A class called `AdaptiveStateManagerBase` derives the base state manager class to provide a general class for numerous adaptive checkpoint interval algorithms. The class provides an interface for measuring and recording timings for event execution, the coast forward phase, rollbacks, and state saving. The class named `CostAdaptiveStateManager` implements the heuristic algorithm described earlier.

### 5.4 GVT Estimation

Because Time Warp requires event and state histories to be maintained in case of rollbacks, memory usage can become large. A mechanism must exist that determines when it is safe to reclaim events and states and to commit I/O operations. Jefferson defines the Global Virtual Time (GVT) at real time $r$ as the minimum of: (1) all virtual times in all virtual clocks at time $r$, and (2) of the virtual send times of all messages that have been sent but have not yet been processed at time $r$ [4]. The GVT can be used to determine how
Several algorithms have been developed that attempt to estimate GVT [12, 38–40]. One simple and efficient algorithm has been developed by Mattern [11].

5.4.1 Mattern

Mattern’s GVT estimation algorithm is based on a snapshot algorithm for determining consistent global state of a distributed system and uses the concept of cuts. A cut is a line dividing a time diagram into two sets: the past and the future. The GVT algorithm uses two cuts C and C’. The basics of the algorithm as described by Mattern can be seen in Figure 5.8. These two cuts represent two control rounds in a ring topology. The goal is to estimate the GVT at C’. To do this, a mechanism must determine the minimum of the local virtual times and the minimum of all messages that cut across C’. By moving C’ to the right so that all messages sent before C arrive before C’, only the messages sent after C need to be examined. From this, the minimum timestamp of messages in transit at C’ can be found.

To begin the algorithm, a control mechanism colors all processes (entities that exchange messages) white. Each process counts the number of white messages it sends and receives. When a process moves past cut C, the control mechanism colors it red and any messages that the process sends are red. The control mechanism maintains the minimum timestamp of all red messages sent by a process. The number of white messages is known for cut C and if none are in transit, then the minimum of all local virtual times at C is a valid GVT estimate. Otherwise, the control mechanism initiates a second round. The minimum timestamp of the red messages can be found along with the minimum local virtual time to get a valid GVT estimate.

In WARPED, GVT estimation occurs through the base class named GVTManager. A class named
MatternGVTManager derives the base class to implement Mattern’s GVT algorithm. The GVT manager keeps track of the white message count for a simulation manager and modifies the value in several ways. Any time a simulation manager sends an event message or negative event message to another simulation manager the GVT manager increments the white message count. If the simulation manager is red, then the GVT manager checks the timestamp of the message in order to determine the minimum timestamp of red messages. Every time a simulation manager receives a white message, the GVT manager decrements the white message count. The white message count is positive if the simulation manager has sent more messages than it has received. The white message count is negative when the simulation manager has received more messages than it has sent. When all simulation managers have received all of the sent messages (there are no messages in transit), the sum of the white message counts of all simulation managers will equal zero.

A simulation manager initiates the algorithm periodically with the period being set in the configuration file. Only one master simulation manager can start the GVT estimation process. The master sends a token around to all simulation managers in a ring topology. The token contains three values: the minimum LVT of the simulation objects, the minimum time-stamp of red messages, and sum of the white message counts. The GVT manager initially sets the minimum time-stamp of the red messages to positive infinity. After sending the token, the GVT manager sets the white message count to zero. When a simulation manager receives the token the first time, it changes from white to red. The GVT manager updates the minimum LVT on the token and adds the white message count of the simulation manager to the white message count on the token. After the GVT manager has added the white message count, it sets the value back to zero. Once the token gets back to the master, the GVT manager adds the white message count of the simulation manager to that of the token. If the sum is zero, then the GVT is the minimum LVT on the token. The GVT manager then sends this GVT to the other simulation managers. The GVT manager then colors the simulation managers white. If the white message count is not zero, then the master GVT manager initiates another round. In this round, the GVT managers will once again update the white message counts and minimum LVTs. In addition, they will update the minimum time-stamp of the red messages. When the token reaches the master, the GVT manager checks the white message count. This process repeats until the count is zero. When this happens, the GVT manager determines the GVT by finding the minimum of the LVTs and the red message time-stamps. The GVT manager then transmits this value to the other simulation managers.
Several problems prevented the Mattern GVT manager from initially working. One major issue was that the only simulation manager keeping track of white messages was the master simulation manager. This meant that some messages were in transit across cut C but were ignored because the master simulation manager did not send or receive them. In some cases, this resulted in the GVT manager using the LVT minimum even though it was not a valid GVT. When this happened, simulation objects consistently encountered rollbacks to times less than the GVT. This resulted in errors because the necessary events and states no longer existed. The fix was to simply have all the simulation managers keep track of white messages.

There was another issue with GVT managers counting white messages. The GVT manager was only counting event messages in the white message count and in determining the minimum red message timestamp. The problem was that anti-messages also affect the LVT of simulation objects and could trigger

![Diagram](image.png)
rollbacks. The GVT manager was not taking these into account. The fix added anti-messages to the white message count and to the minimum red message timestamp calculation.

Another issue involved messages that the lazy output manager could potentially send. When simulation objects do not regenerate events, they send anti-messages. These anti-messages will have a send time that is less than the current LVT of the simulation object and can trigger a rollback at the receiving simulation object. With the original implementation of Mattern's algorithm, the GVT manager was not tracking these anti-messages, as it only counted sent messages and the current LVTs. Once again, simulation objects were experiencing rollbacks to times less than the GVT. To fix this problem, when the GVT manager determines the lowest LVT on the simulation manager, it also determines the lowest timestamp of all events in the lazy

Figure 5.10: Mattern GVT in Warped, Slave.
cancel queues. The GVT manager then uses the minimum between these two times as the minimum LVT. This takes into account the minimum timestamp of any message that may be sent in addition to the ones that are sent.

5.5 Fossil Collection

To enable recovery from causality errors in Time Warp, simulation objects must retain states and events. This can cause simulations to use large amounts of memory and if that memory is never released, memory will be exhausted. To prevent this from happening, Time Warp uses a process called fossil collection. During fossil collection, a simulation object reclaim any states and events that it no longer needs. This typically occurs after the simulation updates GVT because simulation objects do not need states and events with time-stamps less than the GVT.

In WARPED, fossil collection occurs immediately after a simulation manager updates the GVT. The GVT manager updates GVT and then calls on the simulation manager to perform fossil collection. The simulation manager calls on the state manager to fossil collect states first to find the time to use for fossil collecting the events. The next step in the fossil collection process involves the simulation manager committing I/O operations for all local simulation objects. The simulation manager cannot commit these until the fossil collection process because no rollbacks can possibly occur to any time less than the GVT. The output manager than performs fossil collection using the lowest state time not fossil collected. Finally, the event list manager fossil collects the input events.

5.5.1 Optimistic Fossil Collection

The form of fossil collection described is conservative because it must ensure that any events or states that a simulation object needs after a rollback exist. Two downsides to this technique are that GVT estimation algorithms can involve many network messages and incur considerable lag from the true GVT. The GVT can also be affected by a single high use simulation object that triggers frequent GVT estimation (adding unnecessary overhead). The other simulation objects have little or no memory that can be reclaimed but still have to go through the entire fossil collection process.

In order to improve performance and eliminate the need for GVT estimation to perform fossil collection,
Young et al developed a distributed technique called Optimistic Fossil Collection (OFC) [17, 48, 49]. With this technique, each simulation object identifies and collects fossils asynchronously and without requiring a GVT estimate. Instead of using GVT as a boundary for fossil identification, each simulation object identifies fossils based on a probability that the memory actually is a fossil. Since it identifies fossils based on a probability model, the possibility exists that the simulation object might reclaim an active state or event. In that case, a future rollback would cause a catastrophic failure and the simulation would have to be restarted or restored to a time where a complete snapshot is captured [50]. The details of OFC are described more fully below.

OFC allows fossil collection to match the runtime behavior of each simulation object. Each simulation object develops a statistical model of its rollback length. From the model, the simulation object determines the boundary between fossilized and active events and states. The simulation object can then fossil collect events and states that are outside the boundary. Users define a risk factor to determine how aggressively the simulation object will identify fossils. A higher risk factor means that simulation objects will have a smaller bound and reclaim more memory. However, the possibility of a simulation object reclaiming memory that it needs later is also higher. When a rollback occurs beyond the bound for a simulation object, it is called a catastrophic rollback. To recover from catastrophic rollbacks, the simulation must be completely restarted (probably with a lower risk factor) or have secondary snapshots of the entire simulation that can be restored. In warped, a simulation snapshot is infrequently taken (on the order of 10-40 wall clock minutes) in case of catastrophic rollback. Catastrophic rollbacks can lower performance if they occur often and require significant overhead. Lowering the risk factor lowers the number of catastrophic rollbacks.

Optimistic fossil collection can result in a simulation using more or less memory depending on the situation. The case where a simulation uses less memory could result when a simulation object almost never rolls back or only has short rollback lengths. The simulation object requires less state and event histories because it can reclaim them safely within a short time after their use. The distributed nature of optimistic fossil collection allows the simulation object to fossil collect on its own. Because optimistic fossil collection is distributed, each simulation object can fossil collect based on how it is actually behaving. Figure 5.11 shows an example of how simulation objects can have different LVTs and memory usage based on where the GVT is estimated. Figure 5.12 shows the same example using optimistic fossil collection. Some of the
simulation objects might have less memory usage because they have fossil collected at times greater than the GVT.

There are many different ways to implement the various features of optimistic fossil collection. The first implementation decision is how to determine the bound on rollback lengths. Young [48] presents techniques that assume that rollbacks and LVT updates have an underlying stationary distribution. One approach is to simply declare a set number of events or states to be active. When a simulation object needs more, it reclaims the oldest. Other approaches involve sampling the rollback lengths and then determining the probability of a rollback distance. The probability can be determined by modeling the rollback lengths as a geometric distribution. A bound on rollback lengths could also be determined by using the Chebyshev inequality.

The other issue involves implementing the checkpointing and restoration functionality to deal with catastrophic rollbacks. The simplest method of dealing with catastrophic rollbacks is to restart the simulation.
from the beginning. This is obviously not very efficient. Young [48] discusses an efficient algorithm involving checkpointing at set virtual times. The benefits of the algorithm are that no coordination is necessary between simulation objects and the size of the simulation state saved is relatively small.

A simulation saves a consistent simulation state by checkpointing each simulation object as its LVT passes a predetermined virtual time. This time can be set before the simulation begins so that the simulation objects require no coordination. The checkpoint process involves saving the last state of the simulation object before the checkpoint time and all events sent by the simulation object with a send time less than the checkpoint time and a receive time greater than or equal to the checkpoint time.

When a catastrophic rollback occurs, the simulation must determine the checkpoint to restore. The simulation accomplishes this by having the simulation object causing the fault to send a suggested restore time to a designated master. The master then initiates a token passing round to alert the other simulation objects that a catastrophic rollback occurred. When each simulation object receives the initial message, they empty their input and output queues. The simulation objects drain all communication channels of any remaining messages. The master starts a second round to determine the greatest checkpoint time that all simulation objects have passed. A final third round informs all simulation objects about the checkpoint time to use. Once the simulation objects establish the checkpoint time, each simulation object restores its state and output queue and then sends all events in the output queue.

More than one attempt was made at implementing optimistic fossil collection into the new version of WARPED. The primary differences between the implementations were how and when events and states were fossil collected. First, the elements of the implementations that did not change will be discussed.

A new class named OptimisticFossilCollectionManager was created to handle the functionality of optimistic fossil collection. Whenever a rollback occurs, the simulation manager sends the time of the restored state for the simulation object to the optimistic fossil collection manager (OFCM) so that it can sample the restored time. It uses the restored time instead of the rollback time because the restored time is what the OFCM actually needs to know about the active states and events. The OFCM keeps separate samplings for each simulation object. After it has collected a user specified minimum number of samples for a simulation object, the OFCM calculates the bound on the active history length. It then recalculates the bound every time a rollback occurs until a user specified maximum number of rollbacks have occurred.
for the simulation object. From that point forward, the OFCM does not sample rollbacks to determine the active history length bound. The OFCM determines the bound using the Chebyshev inequality as described earlier. Fossil collection managers can override the function called to sample rollbacks to implement other ways of determining the active history bound.

The checkpointing and recovery techniques implemented are similar to those described earlier. The configuration file specifies the checkpoint interval for a simulation. When a single simulation object passes the checkpoint time, the OFCM checkpoints all of the simulation objects on that simulation manager. This is done because the only current scheduler implementation synchronizes all of the simulation objects on the simulation manager so that when one passes a virtual time, that means all others have as well. The OFCM serializes events and writes them to a file. However, it cannot write states to files because there is no serialize function for the State class. If the user defined state class contains pointers, then there would be no way to restore the state after writing it to a file. Instead, the OFCM saves states in memory. This does lead to a larger memory usage but should not present a large problem because the OFCM only saves one state per simulation object for each checkpoint. The only other choice would be to require user defined states to have a serialize and deserialize function similar to events. However, this would mean all existing simulations would have to be modified. The OFCM checkpoints file I/O by storing the current file position for input files and writing all buffered output file data to the checkpoint file that contains the saved events. It also writes the current position of the output file. The OFCM does not write the buffered output file data to the actual files so that the normal rollback recovery method can be used for file I/O. When a simulation object has a rollback and passes the checkpoint time again, the OFCM only updates that simulation object’s state. However, it does write all event and file data to a new output file. This is not very efficient as it repeats work but was used because it was simple to get working.

The recovery process is the same as described earlier. To restore any file I/O, the OFCM resets current positions for input and output files to the saved values. For output files, the OFCM erases all data from the restored current position to the end of the file. It then restores the saved buffer data to the normal file buffers. Once again, this is not the most efficient method as the OFCM will negate file writes but was the simplest way to get a working solution. A problem encountered for the recovery process involved simulation managers receiving the send events sent to restore the event lists after a catastrophic rollback. Once the
recovery process begins, the only communication messages the communication manager receives are those related to the recovery token passing process. The communication manager ignores all other messages. This acts to drain the communication channels of any remaining messages. The problem is that when the master sends out the final round of recovery messages, another simulation manager might start sending events before all simulation managers get the recovery messages. To work around this problem, a field was added to all communication messages indicating the number of catastrophic rollbacks the simulation manager had encountered. The communication manager uses this number to determine if a simulation manager sent a message before or after the last catastrophic rollback. Once the recovery process starts, the communication manager only sends recovery messages, so no event messages from before the catastrophic rollback can possibly be sent.

In addition to the minimum and maximum number of samples, the configuration file specifies the checkpoint interval, the risk factor, and the default active history length of all simulation objects. This is needed so that each simulation object has a bound at the beginning of the simulation until a more accurate one can
The prime difficulty was implementing the actual fossil collection process. The first attempt involved the OFCM reclaiming events and states only when new ones were needed. The `new` and `delete` operators were overloaded for the base event and state classes. Whenever simulation objects required new events and states, they would call on the OFCM to provide the memory for the new event or state. Any time `delete` was called on an event or state, the OFCM would add its memory to a free pool of memory. The OFCM would then use this pool first when called on to provide more memory for states and events. The OFCM examined the oldest event or state and if the time-stamp was outside the active history bound, the OFCM called the destructor and returned its memory for use by the new event or state. If the OFCM could not reclaim any memory, then it allocated more memory. In this way, the OFCM avoided memory allocation overheads and reclaimed events and states only when more memory was actually needed. The operators were overloaded in the base event and state classes so that any existing user simulation models would not
have to be modified. In addition, users would not have to know anything about the underlying Time Warp implementation.

Anytime the OFCM allocated an event or state, it added the new event or state to a list of in use memory. The OFCM did this so that finding the oldest event or state involved simply examining the front of the list. However, this technique produced several problems. The first problem was that simulation managers stored twice as many pointers to events. An alternative would be to have the event list manager return the oldest event. However, this involved searching the processed events for every simulation object on the simulation manager, which was time consuming. The bigger problem was that the oldest event may not be the right size (it might be another event type). The OFCM would have to find the oldest event of the right type instead of just the oldest event. This resulted in even more overhead. The OFCM could not examine a specific simulation object because the constructor of the event had not been called yet, only the memory allocation process had been called. Another issue was that when the OFCM had to reclaim a local event, it had to access another simulation object’s output events or event list to remove the event. This required coordination between simulation objects. These overheads resulted in significant performance degradation, especially as the simulation grew in size.

The current OFCM implementation uses a much simpler method of fossil collection. The new and delete are not overloaded and the OFCM does not need any extra tracking of oldest events. Instead, every time a simulation object processes an event, the simulation manager calls the fossilCollect function for the OFCM. Within this function, fossil collection occurs for the simulation object that just processed the event. The OFCM reclaims any event or state earlier than the calculated active history length. A period counter has been added within this function so that fossil collection does not occur for every processed event but rather only occurs periodically. This value is currently hardcoded but will be added as a configuration option later. The GVT manager still calculates GVT and file I/O commits occur in coordination with GVT updates at this time. In the future, the simulation manager could commit I/O at the same time as the OFCM fossil collection. This second implementation provides performance that was similar to fossil collection using GVT. The analysis chapter has more details.
5.6 Summary

Several Time Warp optimizations have been implemented into WARPED. All of the optimizations are capable of running simulations using the Time Warp mechanism. However, some of them are not performing up to the level expected and could use further enhancements. The next chapter discusses the details of these optimizations’ performance and attempts to explain potential problems.
Chapter 6

Analysis

6.1 Introduction

This chapter details the analysis performed on the Time Warp simulation manager of the WARPED kernel. Because only the sequential simulation manager has been previously well tested, the correctness of the newly implemented Time Warp simulation manager had to be verified. A section in this chapter outlines the steps taken for this verification. In addition, the correctness of the implemented optimizations had to be verified. These optimizations were then analyzed by comparing relevant configurations. Another section in the chapter compares the performance of the various configurations. Most of the verification and performance testing used simulations that model simple processes so that the behavior of the simulation is understood.

6.2 Simulations

The previous version of WARPED was distributed with several simulation models that could be used to perform analysis on the kernel. However, because of the new simulation API, users could not directly use the same test simulation models. Some of the main changes involved the names of the base event and state classes as well as the initialization interfaces. The names of the functions required for interfacing with the simulation manager were different, such as the function needed for sending events to other simulation objects. In addition, the Application class was a new addition to WARPED v2.x, so each simulation
model had to implement the entire class. As described earlier, the Application class initializes and partitions simulation objects. Three of the more important simulations were ported to the WARPED v2.x API: PHOLD, RAID, and SMMP.

### 6.2.1 PHOLD

The PHOLD model is a variant of the Hold model used to test queue implementations [51]. A PHOLD simulation model consists of a set number of processes (implemented as simulation objects in WARPED) and events. The number of events in the simulation model at any given time is constant. Each process starts by sending the same number of events. A process selects the destination of the event and the time-stamp increment by random variables. The user can configure almost every aspect of the simulation model including: process state size, event computational grain, the event process destinations, and the random probability distribution. Figure 6.1 shows an example of a small PHOLD configuration containing 4 processes that all connect to each other. The rollback behavior of a PHOLD simulation model is dependent upon the configuration created. As the degree of connections between processes increases, the number of rollbacks will increase.

![PHOLD Diagram](image-url)

Figure 6.1: The default small PHOLD configuration.
6.2.2 RAID

The RAID simulation models a Redundant Array of Inexpensive Disks level 5. These disks are able to provide increased storage capacity as well as increased I/O performance. In addition, some configurations provide increased safety. The simulation models the action of processes writing and reading data from the disks. A RAID controller distributes a memory request to different disks. To model this system, a unique simulation object represents each disk, process, and RAID controller. The memory requests are the events. Figure 6.2 shows an example of a small RAID simulation model. There are two source processes that send memory requests to two controllers. After receiving a request from a source process, the controller divides up the request and sends the parts to different disks. When a disk receives a request, a delay occurs based on configurable disk parameters that determine the disk seek time. After the delay, the disk sends a response to the source process that sent the request. Once a source process has received responses from all of the individual requests made to the disks, it generates another request. There is a high level of simulation object interconnection as the disks receive requests from multiple controllers, controllers receive requests from multiple source processes, and source processes receive responses from multiple disks. The high level of interconnection produces a high number of rollbacks.

Figure 6.2: The default small RAID configuration.
6.2.3 SMMP

The SMMP simulation models a shared memory multiprocessor. Each processor has a local cache and shares a single main memory with the other processors. The model is not entirely realistic as several requests to main memory can occur at once because requests are not serialized. The source makes a memory request that can be satisfied in the cache or main memory based on a configurable cache hit ratio. The number of time units for a request satisfied in the cache or main memory is also configurable. The default values are 10 time units for the cache and 100 time units for main memory. One processor consists of six components in the simulation: a source, join, queue, fork, server, and another join object. To better approximate the physical system, all six processor components will reside on a single simulation manager. The simulated processors will be distributed among the simulation managers. The main memory consists of four components: a join, queue, server, and router object. These will all reside on the same simulation manager. Figure 6.3 shows the layout of the simulation objects for an example configuration containing 4 processors.

The SMMP model produces infrequent rollbacks as each processor can satisfy requests made to the cache without using the network communication system. Only when processors make requests to the main memory system does the simulation use the communication system. As a result, the simulation manager that contains the main memory system will experience the most rollbacks as it receives requests from all of the other simulation managers.
Figure 6.3: A 4 processor SMMP configuration.
6.3 Testing Setup

The platform used for testing was a Beowulf cluster of 40 nodes. The details of these of these nodes are given in Table 6.1. The three simulations described (PHOLD, RAID, and SMMP) were used for the performance analysis. All three simulations were run on 16 nodes of the Beowulf cluster.

The PHOLD simulation consisted of 200 processes that each started 20 events. An exponential distribution was used for random variable generation. Each process had a state size of 1024 bytes, performed 100 floating point divisions for the event computational grain, and was connected to 20 other processes. The simulation terminated after the GVT passed 2000. A round robin partitioning scheme was used to distribute simulation objects. The RAID simulation consisted of 32 disks, 8 controllers, and 96 source processes. Each source process generated 2000 requests. The SMMP simulation model consisted of 64 processors, resulting in a total of 388 simulation objects being used. Each processor generated 75000 requests, with 85% of them being satisfied in the cache. The cache satisfied requests in 10 time units while the main memory satisfied requests in 100 time units.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Xeon CPU E5410 2.33GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>Core 2 Duo P8600</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>4</td>
</tr>
<tr>
<td>Number of Threads</td>
<td>8</td>
</tr>
<tr>
<td>RAM</td>
<td>4 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Fedora 2.6.23.1-21.fc7</td>
</tr>
<tr>
<td>MPI Implementation</td>
<td>MPICH 1.2.7p1</td>
</tr>
</tbody>
</table>

Table 6.1: Platform specification.
6.4 Verification

After porting the simulation models to the \texttt{WARPED v2.x} API, their correctness was verified. A full regression test suite currently tests the sequential simulation manager thoroughly. Therefore, the simulations were initially run using the sequential simulation manager. Once their expected end states matched the actual states, they could be used to test the correctness of the Time Warp simulation manager. The RAID simulation model was particularly useful in identifying problems. RAID has more complex state variables that quickly cause simulations to complete with an incorrect state if any problem occurs. The simulation model was able to bring out various problems including incorrect state saving and restoration, anti-message problems, global checkpointing and recovery errors, and lazy cancellation problems. \texttt{WARPED v2.x} is distributed with unit tests that test individual classes in the kernel. These tests were used to verify lazy and dynamic cancellation as well as adaptive state saving optimizations. The unit tests also ensured that later changes did not break these optimizations.

6.5 Performance

In addition to the simulations completing with the correct state using various optimizations, the simulations had to perform as expected with those optimizations. This section compares configuration options within certain categories to measure the performance in terms of execution time. In all of the configurations, the default settings were used with only one configuration varying at a time. Time Warp optimizations for event cancellation, state saving, GVT estimation, and fossil collection are analyzed. The event list structures and communication methods are also examined. Although they are not Time Warp specific optimizations, they are helpful for understanding potential enhancements of the \texttt{WARPED} kernel. The same default configuration results are shown in multiple tables and figures only for convenience in comparing results. The following is the default baseline configuration:

- Event List Manager: Default
- Scheduling Manager: Default
- State Manager: Periodic, Period = 10
For all configurations, the PHOLD, RAID, and SMMP simulation models provide the performance measurements. To collect these measurements, an acceptable number of samples had to be taken. This acceptable value can be determined using statistical methods. An initial round of testing was performed to collect sample means and standard deviations for all of the configurations. The number of samples required to have 95% confidence that the error in the mean was less than 1% mostly ranged from 2 to 8 samples, with two configurations requiring more than 15 samples. For simplicity and to balance the length of time required to run all tests, the sample size was set to 10.

### 6.5.1 Event Cancellation Performance

The output manager implements the various event cancellation optimizations. There are four possible configurations for event cancellation: aggressive, aggressive using only one anti-message, lazy, and dynamic. Aggressive cancellation is the default configuration. Figure 6.4 shows the performance of the simulations when the event cancellation technique is varied. The one anti-message optimization performs with a less than 1% difference than aggressive cancellation for PHOLD because event destinations are distributed at random and are unlikely to be clustered. As a result, a single anti-message will not cancel a large number of additional messages. However, the source objects in the RAID simulation send events to only one fork object, so a single anti-message is likely to cancel many additional positive messages from the same object. Therefore, the optimization performed 4% better compared to aggressive cancellation for RAID. For the same reason, the one anti-message optimization also outperformed aggressive cancellation by 1-2% using SMMP.

Lazy cancellation outperformed aggressive cancellation for PHOLD and RAID. The largest gain in performance of 28% came from the RAID simulation. This is due to the disk objects in the RAID model regenerating a large percentage of their events after a rollback. The gain in performance was not as large for PHOLD because event destinations and receive time increments are determined randomly, so regenerated
events are likely to be different after a rollback from a positive event. In addition, the total number of rollbacks is only 10% of the total for RAID simulations. All simulation objects beside the one containing the main memory objects for SMMP encounter almost no rollbacks for the entire simulation. Even that one simulation manager does not rollback as much as the RAID simulation, so there are no performance gains. These results indicate that lazy cancellation has been implemented properly.

The dynamic output manager performed about the same as the lazy output manager with differences in execution time varying by less than 1%. These results correspond with earlier findings by Rajan [10]. One difference is that Rajan reported a 1-2% increase in performance for dynamic cancellation using the
CHAPTER 6. ANALYSIS

6.5. PERFORMANCE

<table>
<thead>
<tr>
<th>Output Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggressive</td>
<td>58.71</td>
<td>(58.26, 58.86)</td>
<td>1444712</td>
</tr>
<tr>
<td>OneAntiMsg</td>
<td>56.38</td>
<td>(56.08, 56.68)</td>
<td>1314104</td>
</tr>
<tr>
<td>Lazy</td>
<td>42.06</td>
<td>(41.81, 42.31)</td>
<td>1984468</td>
</tr>
<tr>
<td>Dynamic</td>
<td>41.96</td>
<td>(41.83, 42.08)</td>
<td>1962851</td>
</tr>
</tbody>
</table>

Table 6.3: Output manager performance for the four different configurations using the RAID simulation.

<table>
<thead>
<tr>
<th>Output Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggressive</td>
<td>61.22</td>
<td>(60.58, 61.85)</td>
<td>221124</td>
</tr>
<tr>
<td>OneAntiMsg</td>
<td>60.25</td>
<td>(60.00, 60.51)</td>
<td>221108</td>
</tr>
<tr>
<td>Lazy</td>
<td>61.46</td>
<td>(61.23, 61.69)</td>
<td>208536</td>
</tr>
<tr>
<td>Dynamic</td>
<td>62.10</td>
<td>(61.70, 62.51)</td>
<td>208393</td>
</tr>
</tbody>
</table>

Table 6.4: Output manager performance for the four different configurations using the SMMP simulation.

RAID simulation. With this implementation, RAID did perform slightly better with the dynamic output manager, but the improvement was not statistically significant (less than 0.5%). This is mostly likely due to the lower ratio of fork objects to disk object used for the RAID simulation in these tests. The fork objects favor aggressive cancellation while the disk objects favor lazy cancellation, so dynamic cancellation should perform better than either of the static configurations.

6.5.2 State Management Performance

The state manager implements static and dynamic periodic state saving. The default static period for the state manager is 10. In addition, periods of 1, 3, and 30 were tested along with a dynamic period implemented in the cost adaptive state manager. The value 30 was chosen as the upper limit because that is the greatest value possible for the cost adaptive state manager. The tests attempt to find out how well the cost adaptive state manager performs.

The results of the test are shown in Figure 6.5. Though the results appear to show that the cost adaptive state manager is incorrect, it is actually working properly. The state period for simulation objects that do not rollback correctly approaches and finishes at 30. Likewise, the state period goes to 1 for simulation objects that rollback very often. However unexpected behavior occurred with SMMP. Because almost all of the simulation objects in the simulation experience no rollbacks, the period value of 30 should produce the
best performance. Instead, this value produces the worst performance. A profile of the SMMP simulation with the period value of 30 showed that SMMP simulation object `executeProcess` code was a larger percentage of the execution time than the simulation with a period value of 1. This was despite the larger period value simulation having fewer rollbacks and fewer anti-messages sent. The debug output showed that only three simulation objects ever rolled back, and these simulation objects adapted their period to 1.

Several attempts at finding the additional cost were not successful at locating the cause of the additional execution time. One possibility is that with the larger period, the simulation objects spend more time in the coast forward phase. However, measurements showed that the total execution time spent in the coast forward phase comprised much less than 1% of the increase in total execution time. Another possible source of the increase in execution time is from the events that are re-executed after the coast forward phase is complete. With a larger period, the rollback length will increase, as will the number of events re-executed. Measuring the time spent executing events more than once is difficult, so the number of re-executed events was measured instead. The results showed that a period of 1 produced more re-executed events than the adaptive period, but yet performed better.

The RAID simulation also provided unexpected results. Because it produces a large number of rollbacks, it should perform better with a lower state saving period. Instead, it performs best with the largest value tested. Both the RAID and SMMP problems seem to point to something in WARPED that influences state saving cost outside of coast forward costs and state save costs. The most probable source is the re-execution of events outside the coast forward phase. However, initial data has not supported this. As of now, the problem and source of the problem remain unknown. More work is required in this area.

<table>
<thead>
<tr>
<th>State Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic-1</td>
<td>50.12</td>
<td>(49.74, 50.50)</td>
<td>200138</td>
</tr>
<tr>
<td>Periodic-3</td>
<td>49.82</td>
<td>(49.55, 50.09)</td>
<td>165983</td>
</tr>
<tr>
<td>Periodic-10</td>
<td>50.29</td>
<td>(59.98, 50.60)</td>
<td>151615</td>
</tr>
<tr>
<td>Periodic-30</td>
<td>53.63</td>
<td>(53.32, 53.95)</td>
<td>158140</td>
</tr>
<tr>
<td>Adaptive</td>
<td>50.11</td>
<td>(49.91, 50.31)</td>
<td>195843</td>
</tr>
</tbody>
</table>

Table 6.5: State manager performance for five different configurations using the PHOLD simulation.
Figure 6.5: Performance of state manager configurations.

Table 6.6: State manager performance for five different configurations using the RAID simulation.

<table>
<thead>
<tr>
<th>State Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic-1</td>
<td>58.07</td>
<td>(57.89, 58.25)</td>
<td>3363377</td>
</tr>
<tr>
<td>Periodic-3</td>
<td>58.74</td>
<td>(58.66, 58.82)</td>
<td>2587201</td>
</tr>
<tr>
<td>Periodic-10</td>
<td>58.71</td>
<td>(58.56, 58.86)</td>
<td>1444712</td>
</tr>
<tr>
<td>Periodic-30</td>
<td>57.16</td>
<td>(56.97, 57.35)</td>
<td>780504</td>
</tr>
<tr>
<td>Adaptive</td>
<td>57.87</td>
<td>(57.69, 58.06)</td>
<td>3590107</td>
</tr>
</tbody>
</table>

Table 6.7: State manager performance for five different configurations using the SMMP simulation.

<table>
<thead>
<tr>
<th>State Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic-1</td>
<td>60.61</td>
<td>(60.37, 60.85)</td>
<td>295147</td>
</tr>
<tr>
<td>Periodic-3</td>
<td>60.32</td>
<td>(59.92, 60.73)</td>
<td>249653</td>
</tr>
<tr>
<td>Periodic-10</td>
<td>61.22</td>
<td>(60.58, 61.85)</td>
<td>221124</td>
</tr>
<tr>
<td>Periodic-30</td>
<td>67.65</td>
<td>(67.22, 68.08)</td>
<td>210338</td>
</tr>
<tr>
<td>Adaptive</td>
<td>64.25</td>
<td>(64.07, 64.43)</td>
<td>355430</td>
</tr>
</tbody>
</table>
6.5.3 Fossil Collection Performance

Fossil collection in WARPED occurs either through the optimistic fossil collection manager (OFCM) of after the Mattern GVT manager updates the GVT. The tests in this section show the response of the simulations as the GVT update period is varied. As the update period decreases, performance deteriorates due to increased communication overheads. As the update period increases, performance also deteriorates because portions of the active history may be placed in slower levels of the memory hierarchy. Figure 6.8 shows that this is the case as simulation execution time increases as the update period varies from 100 to 1000.

The performance of the OFCM is compared to the three results using the Mattern GVT manager. The configuration for the OFCM was different for each simulation tested. The default active history length was the most important parameter modified. This parameter prevents excessive catastrophic faults from occurring before the OFCM has enough time to sample rollbacks. The variance from simulation to simulation is due to the increase in receive timestamps from send timestamps for events. The OFCM for RAID and PHOLD performs worse than the Mattern GVT manager for the two lower update period values but better than the update period value of 10000. The decrease in performance is less than 10% and for both simulations. With SMMP however, the OFCM performs better than the update periods of 100 and 10000 with the Mattern GVT manager. This improvement mostly results from the default active history length parameter because most simulation objects in the simulation never rollback and use the default length for the entire simulation.

<table>
<thead>
<tr>
<th>GVT/Fossil Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mattern-100</td>
<td>48.39</td>
<td>(48.23, 48.55)</td>
<td>149399</td>
</tr>
<tr>
<td>Mattern-1000</td>
<td>50.29</td>
<td>(50.60, 58.56)</td>
<td>151615</td>
</tr>
<tr>
<td>Mattern-10000</td>
<td>61.31</td>
<td>(59.85, 62.77)</td>
<td>140184</td>
</tr>
<tr>
<td>OFC</td>
<td>50.83</td>
<td>(50.71, 50.95)</td>
<td>132353</td>
</tr>
</tbody>
</table>

Table 6.8: GVT manager and fossil collection manger performance for four different configurations using the PHOLD simulation.
Figure 6.6: Performance of GVT manager and fossil collection manager configurations.

Table 6.9: GVT manager and fossil collection manager performance for four different configurations using the RAID simulation.

<table>
<thead>
<tr>
<th>GVT/Fossil Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mattern-100</td>
<td>58.95</td>
<td>(58.68, 59.22)</td>
<td>1572543</td>
</tr>
<tr>
<td>Mattern-1000</td>
<td>58.71</td>
<td>(58.56, 58.86)</td>
<td>1444712</td>
</tr>
<tr>
<td>Mattern-10000</td>
<td>77.98</td>
<td>(77.58, 78.38)</td>
<td>856554</td>
</tr>
<tr>
<td>OFC</td>
<td>64.73</td>
<td>(63.50, 65.95)</td>
<td>1538881</td>
</tr>
</tbody>
</table>

Table 6.10: GVT manager and fossil collection manager performance for four different configurations using the SMMP simulation.

<table>
<thead>
<tr>
<th>GVT/Fossil Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mattern-100</td>
<td>65.11</td>
<td>(64.65, 65.56)</td>
<td>218683</td>
</tr>
<tr>
<td>Mattern-1000</td>
<td>61.22</td>
<td>(60.58, 61.85)</td>
<td>221124</td>
</tr>
<tr>
<td>Mattern-10000</td>
<td>65.92</td>
<td>(65.74, 66.11)</td>
<td>219204</td>
</tr>
<tr>
<td>OFC</td>
<td>64.96</td>
<td>(64.31, 65.60)</td>
<td>223904</td>
</tr>
</tbody>
</table>
6.5.4 Event Set Performance

The DefaultTimeWarpEventSet and the TimeWarpMultiSet are the two current event list managers in WARPED. The purpose of the Time Warp multi-set is to improve performance of the event list by decreasing sorting costs. The results in Figure 6.7 show that this attempt was only successful in improving performance for the SMMP simulation model. There was no statistically significant difference between the two implementations for the other simulation models. As will be shown in a later section, this is due to the communication overheads dominating the simulation time. The cost of maintaining sorted event lists is small compared to the communication costs. However, for SMMP, the communication costs are not as high because most events do not need to be sent over a network. As a result, SMMP performs 2-3% better using the TimeWarpMultiSet. This result shows that the TimeWarpMultiSet does improve event list and scheduling performance.
6.5.5 Communication

The two possible communication manager options use TCP/IP and MPI for the actual communication needed between processors on a distributed platform. This section presents results from these two different communication implementations. Figure 6.8 shows that TCP/IP significantly outperformed MPI on the platform tested. These results have less to do with WARPED and more to do with the platform used for testing and the MPI implementation used. However, they are included here to show that overheads introduced from MPI may be significant depending upon the implementation and platform used.

<table>
<thead>
<tr>
<th>Event Set Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>50.29</td>
<td>(49.98, 50.60)</td>
<td>151615</td>
</tr>
<tr>
<td>MultiSet</td>
<td>50.43</td>
<td>(50.27, 50.60)</td>
<td>142340</td>
</tr>
</tbody>
</table>

Table 6.11: Event set manager performance for two different configurations using the PHOLD simulation.

<table>
<thead>
<tr>
<th>Event Set Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>58.71</td>
<td>(58.56, 58.86)</td>
<td>1444712</td>
</tr>
<tr>
<td>MultiSet</td>
<td>59.45</td>
<td>(59.32, 59.59)</td>
<td>1892643</td>
</tr>
</tbody>
</table>

Table 6.12: Event set manager performance for two different configurations using the RAID simulation.

<table>
<thead>
<tr>
<th>Event Set Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>61.22</td>
<td>(60.58, 61.85)</td>
<td>221124</td>
</tr>
<tr>
<td>MultiSet</td>
<td>59.46</td>
<td>(59.00, 59.92)</td>
<td>221392</td>
</tr>
</tbody>
</table>

Table 6.13: Event set manager performance for two different configurations using the SMMP simulation.

<table>
<thead>
<tr>
<th>Communication Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>50.29</td>
<td>(49.98, 50.60)</td>
<td>151615</td>
</tr>
<tr>
<td>MPI</td>
<td>90.98</td>
<td>(90.05, 91.91)</td>
<td>177377</td>
</tr>
</tbody>
</table>

Table 6.14: Communication manager performance for two different configurations using the PHOLD simulation.
CHAPTER 6. ANALYSIS

6.5. PERFORMANCE

![Communication Manager Configurations](image)

Figure 6.8: Performance of communication manager configurations.

<table>
<thead>
<tr>
<th>Communication Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>58.71</td>
<td>(58.56, 58.86)</td>
<td>1444712</td>
</tr>
<tr>
<td>MPI</td>
<td>125.86</td>
<td>(124.45, 126.28)</td>
<td>1892643</td>
</tr>
</tbody>
</table>

Table 6.15: Communication manager performance for two different configurations using the RAID simulation.

<table>
<thead>
<tr>
<th>Communication Manager</th>
<th>Avg. Execution Time (s)</th>
<th>95% C.I. Execution Time (s)</th>
<th>Avg. Rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>61.22</td>
<td>(60.58, 61.85)</td>
<td>221124</td>
</tr>
<tr>
<td>MPI</td>
<td>134.90</td>
<td>(133.98, 135.81)</td>
<td>195007</td>
</tr>
</tbody>
</table>

Table 6.16: Communication manager performance for two different configurations using the SMMP simulation.
6.6 General Performance and Profiling

While the performance of the optimizations is important, efficiency in the general WARPED kernel is also important. The system was profiled using Valgrind. The PHOLD simulation can provide a balanced workload among all simulation managers, so it is preferred to RAID and SMMP for profiling the kernel. The default configuration is used because the main focus is on the parts of the kernel that are not specific Time Warp optimizations. Table 6.17 shows the percentage of time the simulation spends in some of the major functions. This table is not a list of all the functions, just the ones of note. The table shows that the simulation spent over 51% of its execution time deserializing messages from other simulation managers. The deserialization is the primary cost in a simulation where the workload is balanced. Another important function to note is the cancelEvents function. The simulation spent more than 25% of its time just removing erroneous events. Overall, the profiling of the PHOLD simulation showed that the communication costs of running a simulation on a network of workstations can be substantial. The partitioning of simulation objects can improve performance but is application specific. As a result, WARPED does not attempt to optimize any partitioning of simulation objects.

In contrast to PHOLD, the SMMP simulation does not perform a large amount of communication over a network. Communication between simulation objects stays within a single simulation manager. A profile

<table>
<thead>
<tr>
<th>Function</th>
<th>% Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeWarpSimulationManager::executeObjects</td>
<td>99.99</td>
</tr>
<tr>
<td>CommunicationManagerImplementationBase::checkPhysicalLayerForMessages</td>
<td>97.57</td>
</tr>
<tr>
<td>SerializedInstance::deserialize</td>
<td>51.74</td>
</tr>
<tr>
<td>SerializedInstance::getSerializable</td>
<td>41.89</td>
</tr>
<tr>
<td>TimeWarpSimulationManager::cancelLocalEvents</td>
<td>25.80</td>
</tr>
<tr>
<td>operator new</td>
<td>20.81</td>
</tr>
<tr>
<td>operator delete</td>
<td>16.05</td>
</tr>
<tr>
<td>MatternGVTManager::updateEventRecord</td>
<td>8.22</td>
</tr>
<tr>
<td>dynamic_cast</td>
<td>6.14</td>
</tr>
<tr>
<td>CommunicationManagerImplementationBase::retrieveMessageFromPhysicalLayer</td>
<td>5.92</td>
</tr>
<tr>
<td>DefaultTimeWarpEventSet::handleAntiMessage</td>
<td>2.73</td>
</tr>
<tr>
<td>DefaultSchedulingManager::peekNextEvent</td>
<td>1.49</td>
</tr>
<tr>
<td>TimeWarpSimulationManager::fossilCollect</td>
<td>1.17</td>
</tr>
<tr>
<td>Process::executeProcess</td>
<td>&lt;1.00</td>
</tr>
</tbody>
</table>

Table 6.17: Percentage of total program execution time spent in selected functions.
of SMMP showed that a simulation manager spent over 72% of execution time in the scheduling manager looking for the next event to execute. This shows that improving the event list management for the `TimeWarpSimulationManager` will improve performance for low communication simulations.

### 6.7 Conclusion

The results in the chapter show that the `WARPED` kernel is performing as expected for most configuration options. However, there are still some issues. The main concern is the performance of the different state manager options. There appears to be some costs associated with state saving in the `WARPED` kernel that are not being taken into account. The problem has yet to be resolved.
Chapter 7

Conclusions and Suggestions for Future Research

Conclusions

WARPED v2.x is a publically available discrete event simulation kernel written in C++ and uses the MPI standard message passing interface. It uses run time configuration of simulation parameters and algorithms. The run time configuration eliminates the time burden of recompilation needed in WARPED v1.x every time a new configuration is needed. WARPED v2.x can now be distributed as a shared library because different compilations are not needed for different configurations. The WARPED v2.x kernel presents a single API for simulations. Once a simulation has been compiled, many different configurations can be used by setting parameters in a configuration file. The configuration options include Time Warp optimizations for output event cancellation, state saving, GVT estimation, and fossil collection.

Several simulation models are included in the WARPED v2.x distribution. These simulation models can be used to quickly perform analysis whenever new algorithms are integrated into WARPED. Unit tests are also included to enhance the maintainability of WARPED. The simulation models include: PingPong, PHOLD, RAID, and SMMP. These simulation models can be used for testing new optimizations by enhancing and extending the WARPED kernel.
Suggestions for Future Work

While WARPED is now working in many different configurations, more work could be performed on adding more features and optimizing for greater performance gains. Some configuration options could contain more optimization choices. Parts of these additional options have been started but are not completely working. Others are not seen as necessary at this time.

One potential option is to include a UDP option. The basic class structure and code already exists in the WARPED v2.x distribution but is not currently working. If the need for such a communication system is found to exist at a later time, it could be completed. The primary need would seem to be for decreased communication costs, though an unreliable communication system is probably not desirable for simulations.

A communication optimization called message aggregation [52] is also partially implemented. Some of the difficulty in implementing this optimization is putting together and taking apart the aggregated messages. The overheads in sending messages separately may also not be greater than any delays or causality errors that may occur while the messages are being aggregated.

Optimistic fossil collection currently contains only one option for deciding the length of time acceptable for fossil collection. Other options such as a geometric decision function could be added. This would primarily be for comparison purposes. Optimistic fossil collection does not work with the Threaded Time Warp simulation manager at this time. The main problem involves the checkpointing functionality. One object may pass the checkpoint while others have not. In the Time Warp simulation manager, the optimistic fossil collection manager (OFCM) checkpoints all objects when one of the objects passes the checkpoint time. For the Threaded Time Warp manager, the checkpoint function needs to be modified so that the OFCM checkpoints objects separately. Changes will have to be made with the restoration process as well.

Another aspect of optimistic fossil collection that could be improved is the storing of the saved data. At this time, the necessary events and file operations are written to files. These files can grow to enormous sizes when file I/O is used in simulations as a file operations have to be saved. An attempt could be made to reduce the number of file operations saved to reduce checkpointing overhead.

The procedure and commands used to start a simulation can be generalized so that a greater variety of platforms can be supported. For example, WARPED currently starts parallel simulations by having one simulation manager spawn the other simulation manager processes. This startup process could be changed
so that each simulation manager process starts independently, followed by a phase to coordinate communication and start of the simulation. The benefit here would be that the processes could be distributed and started when virtual machines boot.

The stream classes for input and output could be extended to include greater functionality. At this time, there are only two functions, one for reading lines and one for writing lines. Some of the other standard file operations such as get or put could be implemented. The C++ input and output operators could possibly be overloaded to provide a simpler interface. There are difficult issues to overcome with both of these however. Any operation that occurs has to be buffered so that rollbacks can be handled. In addition, a large number of operations may have to be written out to files when saving checkpoints in optimistic fossil collection.
Bibliography


Appendix A

WARPED Configuration File

A.1 Sequential Simulation Configuration File

```
# FileName: sequential.config
# Simulation Type
Simulation: Sequential
#
#
# EventList
# Type Options:
# Type: SingleLinkedList, SplayTree
EventList { 
  Type: SingleLinkedList
}
```

A.2 Time Warp Simulation Configuration File

```
ParallelDebug { 
  # Put every CPU into a loop that could only break out in gdb.
  # To use: set true and
  # Kick off a simulation, all sim managers will be sitting there spinning.
  # Attach on every CPU with gdb, and then break them out and you could watch
  # every CPU in gdb simultaneously. Use: "set x=0" in gdb to break the while loop
  SpinBeforeSimulationStart: False
}
```

```
# Simulation Options:
# TimeWarp, (Single thread TimeWarp simulation)
# ThreadedWarp (TimeWarp with multiple threads executing objects)
# Sequential, (Single thread sequential simulation)
Simulation: TimeWarp
```

```
# TimeWarp Scope
TimeWarp {
```
### W ARPED CONFIGURATION FILE

APPENDIX A.

#### A.2. TIME WARP SIMULATION CONFIGURATION FILE

```plaintext
###
# ObjectQueue and ThreadControl are only used for ThreadedWarp Simulation types
###
# ObjectQueue:
# Type Options:
# CalendarQueue (Not Currently Working)
# LockedQueue
# NumberOfBuckets : For CalendarQueue specify an integer number of buckets
# BucketWidth : For CalendarQueue specify an integer width for each bucket
ObjectQueue {
  Type : LockedQueue
  NumberOfBuckets : 5
  BucketWidth : 2
}
# ThreadControl:
# WorkerThreadCount : Integer value for the number of worker threads
# If left undefined /proc/cpuinfo is used
ThreadControl {
  WorkerThreadCount : 2
}

# Type Options:
# Default, MultiSet (must use if using MultiSet event list)
Scheduler {
  Type : Default
}

# EventList:
# Type Options:
# Default, MultiSet, Threading
EventList {
  Type : Default
}

# CommunicationManager:
# PhysicalLayer Options:
# MPI requires (--enable-timewarp)
# TCPSelect requires (--enable-timewarp)
# UDPSelect requires (--enable-timewarp), not implemented yet
# Type Options:
# Default
# MessageAggregating – Not fully implemented yet.
# Nodes Format:
# List of nodes to simulate on IN ADDITION TO the starting node
# At least one node besides the current must be specified or
# do not specify the Nodes option to run 1 simulation manager
# The names can be repeated however for threadedwarp
# it would be more efficient to increase the number of threads
CommunicationManager {
  PhysicalLayer : MPI
  Type : Default
  Nodes : n001, n002, n003, n004
}

# StateManager:
```
APPENDIX A. WARPED CONFIGURATION FILE

# Type Options:
#   Periodic, Adaptive
# Period Format:
#   Must be an integer value

StateManager {
    Type: Periodic
    Period: 10
}

# OutputManager:
# Type Options:
#   Lazy, Dynamic, Aggressive
# AntiMessages Options:
#   Default
#   One

OutputManager {
    Type: Aggressive
    AntiMessages: Default
}

# GVTManager
# Type Options:
#   Mattern, AtomicMattern
# Period format:
#   Must be an integer value

GVTManager {
    Type: Mattern
    Period: 1000
}

# Optimistic Fossil Collection Manager
# Type Options:
#   None, Cheby

OptFossilCollManager {
    Type: Cheby
    CheckpointTime: 1000
    MinimumSamples: 64
    MaximumSamples: 100
    DefaultLength: 2000
    AcceptableRisk: 0.99
}

}