Time Warp Simulation on Multi-core Processors and Clusters

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 School of Electric and Computing Systems
of the College of Engineering and Applied Sciences

August xx, 2015

by

Doug Weber
BSEE, University of Cincinnati, 2014

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

Abstract

Acknowledgments

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

Many systems can be described by events that occur in discrete time intervals such as communication networks, digital logic circuits, transportation systems, or disease outbreak. To gain a better understanding of these systems, researchers develop models of the systems and perform simulations on computing platforms. These *Discrete Event Simulations* (DESs) can take a long time to simulate large, complex systems by sequentially processing events which has led researchers to design parallel algorithms to run simulations on parallel computing platforms. The field of study that deals with parallel algorithms to speed up discrete event simulations is known as Parallel Discrete Event Simulation (PDES).

Parallel algorithms can be written for many different architectures that support parallelism such as shared memory architectures, clusters, or any other type of system that support parallel execution. Furthermore, communication between concurrent workers in the system can be achieved by using shared data structures or by passing explicit messages between workers.

Some time warp systems are designed for only shared memory multiprocessors. These systems minimize communication latencies and allow very fast, simple algorithm because everything can run in a single address space and use shared data structrures with pointers to prevent unnecessary copying. However, they are still limited by the computational power and memory size of the machine. Other time warp systems are completely based on a message passing scheme. These systems can be scaled to any number of machines using interconnection networks as a means of exhchanging messages. However, this approach introduces high communication latencies.

The WARPED2 simulation kernel, which is a reimplementation of the original WARPED simulation kernel, is an entirely different approach which uses shared memory between a set of *worker threads* in a single process to eliminate communication overheads, but uses message passing between processes to allow the system to scale to larger sizes. A dedicated *manager thread* handles all message passing communication between processes. Figure 1.1 illustrates the communication model that is used in WARPED2.

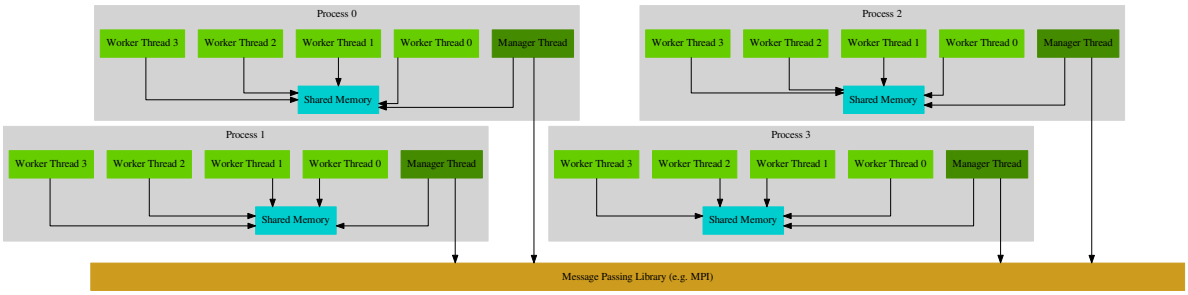


Figure 1.1: Communication Model of WARPED2

This communication model not only allows for high scalability and reduced communication overhead within processes, but allows time warp simulations to follow the critical path better since the worker threads can share a scheduling data structure to process events from. Fewer scheduling data structures creates a bottleneck since multiple worker threads cannot access them simultaneously without race conditions. On the other hand, more scheduling data structures allows more concurrency but spreads out the critical path of execution, allowing more rollbacks.

To further reduce any communication overheads in both message passing and shared memorory communication, partitioning the work between processes will also be explored. Partitioning the work in a parallel discrete event simulation is achieved by simply partitioning the LPs. In WARPED2, a two phase partitioning scheme will be explored. The first phase is for partitiong LPs among processes in order to minimize communication and the second phase is for further partioning the LPs in a process to the event scheduling data structure which should localize LP communication among worker threads and reduce rollbacks.

The combination of shared memory and message passing communication does create some complications in the implementation of a time warp system. GVT and termination detection algorithms are usually designed for either shared memory or message passing but not both. Using a single message passing algo-

rithm between worker threads and processes would mean that some kind of message passing scheme would have to be implemented for worker threads to communicate. This scheme would still use shared memory for message communication but would still have the overheads of explicit message passing. On the contrary, it is possible to have a single shared memory algorithm that extends to distributed memory systems but would require a shared virtual memory system. A shared virtual memory system would still have to transparently use message passing to achieve synchronization. For these reasons, the GVT and termination algorithms developed for WARPED2 include a message passing algorithm and a shared memory algorithm that work in conjunction.

1.1 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 contains some background information on parallel simulation and parallel computing that is used in this thesis.

Chapter 3 reviews several of the prominent parallel simulation kernels that use the Time Warp synchronization protocol. The software architecture and target compute platforms for each is described.

Chapter 4 introduces the software architecture and modeling API for the WARPED2 simulation kernel.

Chapter 5 gives an overview of the studies carried out in the remainder of the text, motivations for the studies, and the hardware architectures and simulation models used in the studies.

Chapter 6 describes the pending event set data structures used within the WARPED2 kernel and provides some preliminary results in finding the best configurations for various models.

Chapter 7 talks about partitioning.

Chapter 8 descibes various GVT and termination detection algorithms that have been used explains the algorithms used in WARPED2. This chapter also provides some preliminary results in determining the best configurations for these algorithms.

Chapter 9 analyzes techniques for managing memory efficiently. This includes state saving techniques, fossil collection, and GVT period. Some preliminary results are also shown for various configurations.

Chapter **??** summarizes the preliminary results from preceding chapters by putting them all together.

Finally, Chapter 11 contains some concluding remarks and suggestions for future research.

Chapter 2

Background

This chapter describes some of the basics of Parallel Discrete Event Simulation (PDES) with a focus on the Time Warp mechanism. Then a brief overview of parallel architectures and parallel communication models and their strengths and weaknesses are compared. Lastly, the big.LITTLE platform is introduced and some different scheduling/switching mechanisms are described.

2.1 Discrete Event Simulation

Discrete Event Simulation (DES) is a method of modeling the execution of a physical system with a sequence of events that occur at discrete time intervals. A Discrete Event Simulation typically contains three main data structures

State variables: A set of variables that describe the current state of the system.

Simulation Clock: A clock to measure the progress of the simulation and determine the order of event processing.

Pending Event Set: A set of future events that are waiting to be procesed.

A *Simulation Model* describes a physical system by a set of *Logical Processes* (LP's). Each LP corresponds to a physical process that is part of the physical system. The LP's interact with timestamped events that

dictate the simulation time that the event should be processed. With each event that occurs, and only when an event occurs, the state of the system is updated.

In a *Sequential* Discrete Event Simulation only one event is processed at a time. All pending events are kept in a single list which is sorted by timestamp. The next event to be processed is always the one with lowest timestamp. Each successive event updates the state of the system, advances the simulation clock, and possibly produces new future events. This is clearly not very efficient for large simulations. This method can be improved by realizing that events for different LP's are independant and will only affect the state for a single LP.

2.2 Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) is a method running a discrete event simulation on a parallel computer which could be a shared-memory multiprocessor, a distributed memory system such as cluster or NUMA system, or a combination of both. In a parallel discrete event simulation the state of the system is usually split among the logical processes so that each one contains a portion of system's state without any sharing of state variables [1]. In addition to each logical process having it's own separate state, the logical processes also have seperate simulation clocks and pending event sets. Event's from different LP's can then be processed concurrently without the need to worry about sharing state variables and the model can be viewed as concurrent processes operating independantly which contribute to the overall progression of the simulation. This has the potential to increase performance significantly; However, it is possible that events at a receiving LP can be received and processed out of order, violating causality. These *causality errors* can occur because of the independant nature of the logical processes and because the LP's can be processing events at different rates. Causality errors can produce incorrect changes in state variables and incorrect events to be sent to other LP's. Parallel Discrete Event Simulation techniques can be categorized in terms of how causality errors are handled. *Conservative* approaches use methods to detect when possible causality errors might occur and prevent them from ever occuring. *Optimistic* approaches, on the other hand, allow causality errors to occur but use methods to detect and recover from the errors. Generally, the simulation models can be developed without the knowledge of the underlying simulation mechanism. The simulation mechanism is usually implemented in a self-contained module which provides an API for the models and is

commonly referred to as the *kernel* or *executive*. For the remainder of this text, only optimistic methods will be discussed, specifically the Time Warp mechanism which is the most widely used optimistic mechanism used in practice.

2.2.1 Time Warp

The Time Warp mechanism is an optimistic method of simulation which is based on the virtual time paradigm [2]. *Virtual Time* provides a method of ordering events in distributed systems which are not described by real time such as a simulation. When used for parallel discrete event simulation, Virtual Time is synonymous with simulation time. The current time of an LP's simulation clock in Time Warp is called the *Local Virtual Time* (LVT).

When an a causality error is detected at an LP (next event to be processed is less than the simulation time) the effects of the incorrectly processed event(s) must be undone. The process of undoing the effects is called a *rollback* and the event that triggers a rollback is called a *straggler event*. When a straggler event is detected at an LP, the first step taken during the rollback is to restore the LP's state back to a previous state before the incorrect event(s) were processed. Then the LP must "unsend" the events that were incorrectly sent by sending *negative events* or *anti-messages*. The negative event, when received by the receiving LP will stop the corresponding positive event from being processed or if the corresponding positive message has already been processed at the receiving LP then that LP must also rollback. This processes recursively occurs until all causality errors are corrected. The negative messages are never processed as normal events but serve only to annhilate an generated event produced by an incorrectly processed event (causality error).

Jefferson [2] describes how to support rollbacks with three main data structures:

1. Input Queue
2. Output Queue
3. State Queue

Every LP will have a seperate input queue, output queue, and state queues. The input queue contains the unprocessed and processed events for the LP that it belongs to. The input queue must be sorted in timestamp order and the LP's must always process from the lowest unprocessed event. The LVT is always the largest

timestamped processed event and is used to detect a straggler event. The output queue contains the events that have been sent by the LP that it belongs to which will allow the LP to send anti-messages during a rollback. The state queue contains previous states of the LP and allows the proper states to be restored during a rollback.

The *Global Virtual Time* (GVT) of the simulation at a given point during the simulation is the minimum of all unprocessed events and the send times of all events that have been sent but not received [2]. There are numerous algorithms for determining the GVT which will be discussed further in chapter8. LP's cannot send events that are less than their LVT value and so the GVT acts as a lower bound on how far a rollback can occur. Because no LP's will ever rollback past the GVT value, it is often used to free memory that is no longer needed for the events in the input and output queues and the states in the state queue that have timestamps less than the GVT as well as committing I/O operations that cannot be undone. This process of freeing memory and committing I/O operations is known as *fossil collection*. Fossil collection does not have to be based on the GVT and several other methods of fossil collection have been developed which will also be discussed more in chapter 9.

The need to save the state of the LP's is one of the fundamental overheads in the Time Warp mechanism in terms of both the amount of time it takes to copy the LP's states to save them and the amount of memory that must be used to store them. Because of this, a number of different approaches have been developed to reduce the overhead of state saving such as periodic state saving, incremental state saving, and reverse computation.

2.3 Parallel Systems Architectures

Systems that support parallel processing come in many forms. They can generally be characterized by how processors and memories are grouped together. Shared memory systems are a single machine which share a common address space which can have a single physical memory unit or multiple memory units. On the other hand, a cluster is a system comprised of multiple machines with seperate address spaces.

2.3.1 Shared Memory Multiprocessor Systems

A Symmetric Multiprocessor (SMP) is a type of shared memory multiprocessor system where each processor has uniform access to a single shared memory through a common bus. SMP systems cannot scale very large due to increasing contention as the number of processors increasing and for this reason, they usually have 8 or fewer processors. To increase available bandwidth in the system, each processor usually has one or two levels of private caches as well as a shared cache which act as a way limit the number of memory accesses. Figure 2.1 illustrates what a typical SMP system looks might look like.

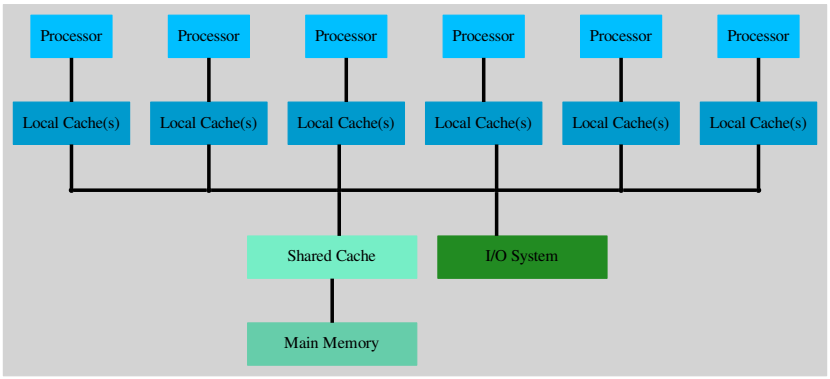


Figure 2.1: SMP System

A Distributed Shared Memory sytem, also known as a Non-Uniform Memory Access (NUMA) system has mulitple memories that are distributed. In these systems, the memories are stilled shared between processors, but access to different memories may take different amounts of time. An interconnection network connects processors and memories together with one or more processors per memory. Because memory access times can vary, software on NUMA systems usually try to keep memory accesses local to a processor. However, NUMA systems can scale much larger than SMP systems because contention to single memory does not necessarily increase with an increasing number of processors. Figure 2.2 illustrates what a typical NUMA system might look like.

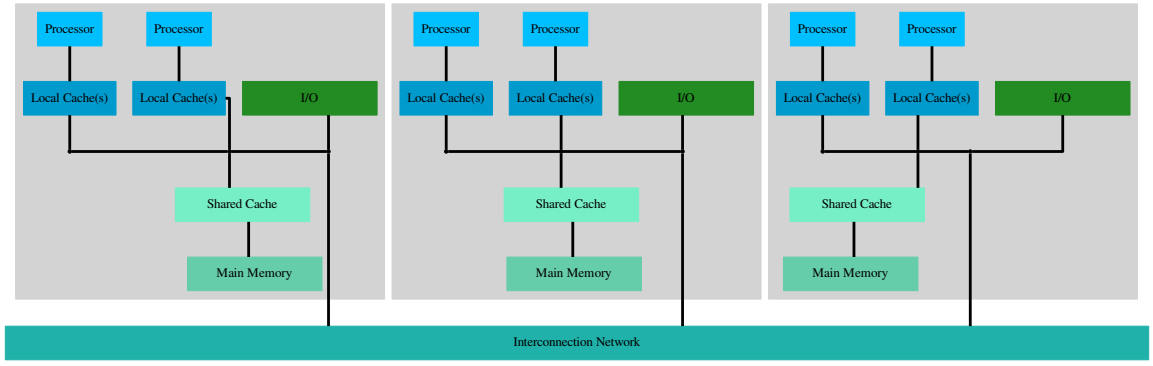


Figure 2.2: NUMA System

2.3.2 Clustered Systems

A Beowulf Cluster is a type of cluster which appears to the user as a single machine but is actually a loosely coupled set of machines connected together over a local network. A single program is executed by all machines concurrently by launching multiple processes on each machine. A program written for a beowulf cluster typically use some type of message passing to communicate among processes and is typically written with parallel communication software such as MPI or PVM. Figure 2.3 illustrates a common realization of a beowulf cluster.

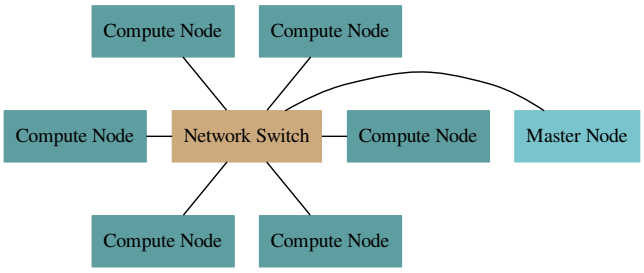


Figure 2.3: Beowulf Cluster

2.4 Parallel Systems Communication

Parallel applications are composed of multiple workers that operate independantly in parallel and may have to exchange information. The workers in a parallel application can be a process, thread, or any other type of execution context and can be within a shared memory system or a cluster or any type of parallel system. Workers can communicate by either sending explicitly passing messages to each other by means of well defined message formats or by using shared data structures that all workers can access. The former method of communication is known as *message passing* and the latter method of communication is known as *shared memory* communication. Both communication methods are fundamentally different and both have strengths and weaknesses.

2.4.1 Message Passing

In a message passing system, workers are completely isolated in different address spaces and communicate only through serialized messages. The formats of the messages must be defined so that the message can be serialized and deserialized by the sender and the reciever, respectively. Message passing can either be synchronous or asynchronous. With synchronous message passing, the send/receive operations must be done in a specific order so that the sender/receiver workers operate together in a synchronized fashion. The send operation at the sender will block until the message is received at the reciever and the receive operation at the receiver will block until the message is fully received. That means the every worker must follow a predictable communication pattern. Workers cannot continue other operations during communication operations and may slow down the whole system. On the other hand, asynchronous communication allows workers to start a send and recieve operations and immediately continue without blocking. To allow this, temporary queues must be used to hold pending operations. The workers do not have to follow a predictable communication pattern in this case because the messages will be queued and can be processed at any time and in any order. The main advantage of message passing is that the number of workers can be scaled to any size as long as the work is partitioned in the right way. Also, he workers can execute in address spaces on different machines and communicate over a local network connection. The biggest disadvantage, however, is an increased communication latency which can be vary large compared to the speed of computation. This makes message passing especially hard for fine-grained parallel applications. An illustration of simple

message passing is shown in figure 2.4.

Message Passing Interface (MPI)

MPI [3] is an extensive message passing API specification for parallel applications and supports both synchronous and asychronous forms of communication. It is a standard specification for developers and MPI users and many current implementations exist. The most widely used implementations used in practice include MPICH and OpenMPI.

2.4.2 Shared Memory

The workers in a parallel application can also share a common address space and communicate through shared data structures. The producer worker will insert data directly into the data structure and the consumer worker will remove the data and use it. This takes much less time to transmit data than with a message passing scheme. However, access to the shared data structures must be protected so that multiple workers do not simultaneously access the same data which could cause unpredictable results. Access to the data structures is usually enforced using lock synchronization mechanisms that protect entire sections of code that are executed by different workers and can end up accessing the same data structures such as mutexes or semaphores. Shared memory data structures may suffer from performance if lots of workers contend for the lock at the same time. For this reason, it is very hard to scale systems that use only shared memory as a means of communication. An illustration of a simple shared memory system is show in figure 2.4.

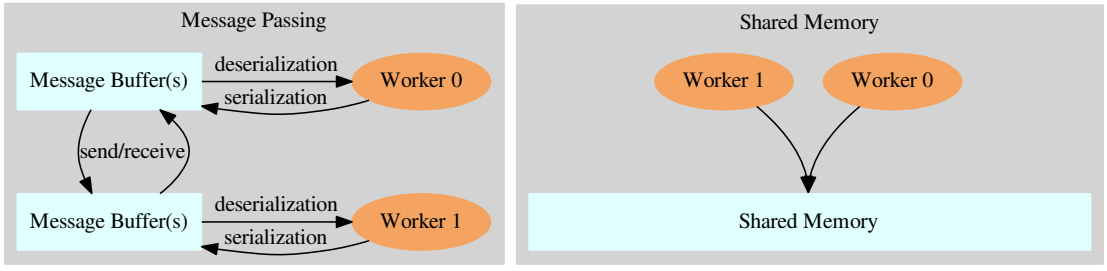


Figure 2.4: Message Passing and Shared Memory Communication

2.5 ARM big.LITTLE

ARM big.LITTLE is a computer architecture design which uses a combination of powerful processor cores that use a lot of power (big) with slower, more power-efficient processor cores (LITTLE). The purpose of this design is to allow good performance when the system load is heavy by running tasks on the big cores while also allowing power savings when the system load is low by running tasks on the LITTLE cores. The operating system CPU scheduler can make use of the big.LITTLE architecture in multiple ways by using different task allocation and switching policies. The three main methods that are currently being used in practice are cluster switching, CPU migration, and heterogeneous multi-processing. The remainder of this section describes the three methods in more detail.

2.5.1 Cluster Switching

With cluster switching, the LITTLE cores and big cores are grouped into *clusters* with the big cores in one cluster and the LITTLE cores in another cluster. At any point in time, the OS scheduler will only schedule tasks to cores within a single cluster. When the system gains enough load so that a big core is needed, then the OS scheduler must switch to the big cluster and only use the big cores. This is the simplest method but can still waist a lot power by unnecessarily switching to the big cluster. Furthermore, only half of the cores are available at time which prohibits the use of full computational capacity. A diagram that illustrates cluster migration is shown in figure 2.5.

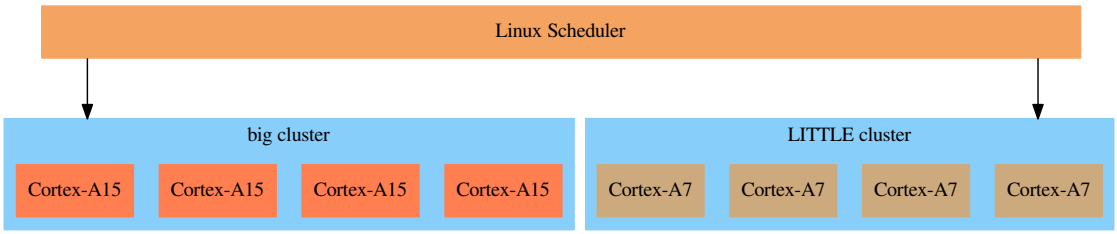


Figure 2.5: Cluster Switching

2.5.2 CPU Migration

CPU migration is the next step up from cluster switching. With CPU migration, each big core is paired with a LITTLE core and the OS treats them as a single virtual core. At any point in time, the OS scheduler will only schedule tasks to only one of the physical cores within each virtual core. The advantage of this approach over cluster switching is that if the load on the system is only large enough so that a single big core is needed then power isn't waisted because only a single big core will be switched on. Like cluster switching though, only half the of CPU cores are available at a time which still prohibits full computational capacity of the system. A diagram that illustrates cpu migration is shown in figure 2.6.

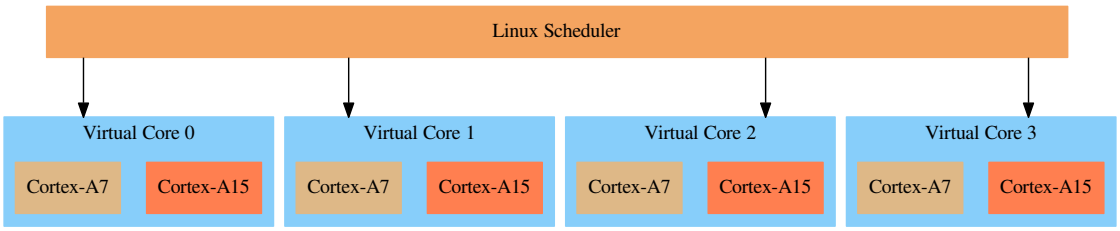


Figure 2.6: CPU Migration

2.5.3 Heterogeneous Multi-Processing (HMP)

Heterogeneous Multi-Processing or Global Task Scheduling (GTS) allows simultaneously scheduling to all CPU cores at the same time. The tasks that have a higher priority or require more processing power will be scheduled to the big cores whereas the tasks with low priority that don't require much processing power, such as background tasks can be scheduled to the LITTLE cores. This is the most complex approach and is hard to implement because the OS scheduler must not treat all cores the same. Policies to allocate tasks must be implemented in the proper manner as well as policies for migration of task to and from the big cores and LITTLE cores. Much research is still in progress to determine the right policies. A diagram that illustrates HMP is shown in figure 2.7.

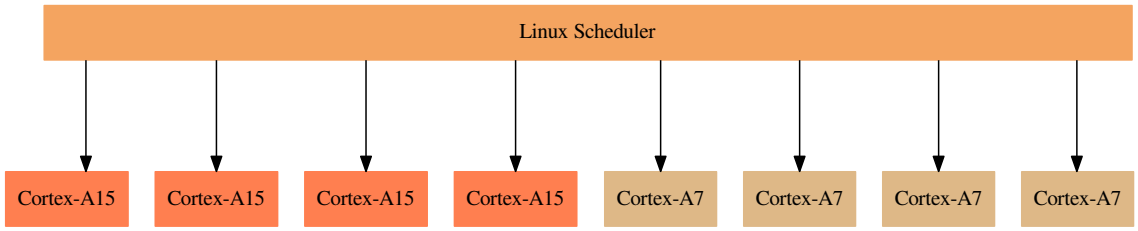


Figure 2.7: Heterogeneous Multi-Processing

Chapter 3

Related Work

This chapter give an overview of some of the most popular Time Warp implementations. For each impementation the design will be described at a high level with a focus on the target architecture. The featured data structures and algorithms for each will also be described as well as the strengths and weaknesses.

3.1 Georgia Tech Time Warp (GTW)

Georgia Tech Time Warp wass a general purpose Time Warp Simulator designed specifically for shared memory multiprocessors. It is not used anymore, however, because it was only written to target SparcStation and SGI PowerChallenge which are now obsolete. Although GTW is not used any more, it's design has influenced the design of other simulators that are still used widely in practice. GTW simulation models run in a single processes, multi-threaded environment and uses only shared memory to communicate between threads that are bound to single processor.

The LP's for all models are statically allocated to a single thread so that events for the LP's are processed only on a single processor. The pending event set is distributed among threads with each having its own data structures for the set of LPs that belong to it. Since the threads can only run on a single processor, the data structures also belong to a singl processor. The pending event set for each processor consists of three main data structures listed below [4]:

1. The *Message Queue* is a linked list that contains positive messages that are destined for the LP's

mapped to the owning processor. Access to the message queue must be synchronized because it can be accessed by tasks running on any processor.

2. The *Cancel Queue* is a linked list that serves exactly the same purpose as the message queue except that it containes only negative messages(anti-messages). Access to this queue must also be synchronized.

3. The *Event Queue* is used to hold unprocessed and processed events and is directly used to schedule events to be processed. The event queue is actually made up of different data structures, one for processed events and one for unprocessed events. The processed events are contained within a doubly linked list and the unprocessed events are contained within a priority queue which can be configured to be either a calendar queue or a skew heap depending on a user configuration.

When messages are sent between LP's, they are inserted directly in the message queue or the cancel queue depending on whether they are positive or negative. Each thread processes events by first moving events from the message queue to the event queue and processing rollbacks. Then, the messages from the cancel queue are removed and cancellations are processed and any more rollbacks are processed. One or more of the smallest events from the event queue are then processed and added to the processed event list. This procedure is repeated over and over again by all processors. Psuedocode for the main event processing loop in GTW is show in figure 1.

Algorithm 1: GTW Main Event Processing Loop [4] [5]

```
while eventQ is not empty do
    move messages from MsgQ to EvQ and process any rollbacks
    remove anti-messages from CanQ, process annihilations and rollbacks
    remove N smallest timestamped events E from EvQ
    process the N events
```

To avoid accessing the message queues and cancel queues too often, which is a contention point, a larger value of N can be used. GTW calls this batch processing and N is the batch interval. With batch processing, multiple events will be processed at a time without processing any rollbacks or cancellations.

Since GTW is only uses shared memory communication, anti-messages do not need to be explicitly sent. Only a pointer to the event that needs to be cancelled is necessary. Fujimoto calls this method *direct*

cancellation. Also by only using shared memory, GVT can be calculated very quickly using shared data structures instead of passing messages around to all processors. The downfall of GTW, however is that it was limited to only a single multiprocessor machine and it was only designed and optimized for specific architectures.

GTW also imposed some unnecessary requirements for the developer of particular simulation models. The partitioning of the LP's among processors must be done in the simulation model during initialization. That means that the model developer must understand some the features of the underlying architecture such as the number of processors, to effectively partition the LP's. Furthermore, the initial partitioning of the LP's is hard to dynamically balance during the simulation because there are no seperate input queues for each LP but rather a single message queue to hold all unprocessed events for each processor.

3.2 Clustered Time Warp (CTW)

Clustered Time Warp [6] (CTW) uses a hybrid approach by processing events within a *cluster* of LP's sequentially and using the Time Warp mechanism between the clusters. This design was chosen because it works well for digital logic simulation which tend to have localized computation within a group of LP's. Furthermore, digital logic simulation tends to have low computational granularity and lot's of LP's which can lead to a lot of rollbacks and a large memory footprint in a traditional Time Warp simulator. CTW is implemented for shared memory multiprocessors but only uses shared memory for use with a custom message passing system so that it can better support NUMA architectures. It is not designed for use on a network of machines such as a beowulf cluster.

Each cluster of LPs has a timezone table, an output queue, and a set of LP's which each have an input queue and a state queue. The timezones in the timezone table are divided by timestamps of the events received from LP's on different clusters. Whenever an event is received from a remote cluster, a new timezone is added. Only a single output queue is needed per cluster because anti-messages can only be sent between clusters and not between LP's on the same cluster.

When a straggler event arrives at a cluster, all of the LP's that have processed events that are greater than the timestamp of the straggler will be rolled back. This rollback scheme is called *clustered rollback*. The alternative to clustered rollback is *local rollback*. In a local rollback scheme the straggler event would be

inserted into the receiver LP's input queue and the LP will roll back when it is detected. Although clustered rollbacks may cause some LP's to be rolled back unnecessarily leading to slower compuation, the approach was chosen for CTW because processed events will not have to be saved which requires less memory.

CTW uses a form of infrequent state savings with the timezone table used to determine the frequency. When an event is about to processed for an LP, the timezone of the last processed event is looked up and if event that is about to be processed is in a different timezone then the state is saved. This approach in which all LP's save their state every time an event is processed in a new timezone regardless of whether it receives an event from a remote cluster is called *local checkpointing*. This method reduces the state saving frequency more than a *clustered checkpointing* approach in which only the LP that receives an event from a remote cluster saves its state. The local checkpointing approach was chosen for CTW because a larger state saving frequency can increase rollback computation and it can even lead to more memory consumption because more events must be saved for coast forwarding during state restoration.

3.3 Rensselaer's Optimistic Simulation System (ROSS)

ROSS [7] is a general purpose simulator that is capable of running both conservatively and optimistically synchronized parallel simulations as well as sequential simulations. It is most often used for optimistic parallel simulations which is achieved with the time warp mechanism. ROSS started as a reimplementation of GTW and is still modeled after it but has had many enhancesments. The same basic event scheduling mechanism is used but ROSS supports different priority queue implementations and different algorithms are used for fossil collection, state saving, and gvt calculation. In addition, ROSS uses processes instead of threads and uses message passing explicitly with MPI instead of shared memory for communication among the processes.

Just as in GTW, ROSS maps every LP to a process and each process contains its own pending event set structures. No locks are needed explicity within each process because there are no shared data structures among processes. The data structures are very similar to those used in GTW but have a different naming convention. The main data structures in ROSS are:

1. The *Event Queue* is analogous to the message queue in GTW. It contains the positive events for all

LP's in the corresponding process. In addition, an event queue is used to hold all remote events regardless of whether it is positive or negative. The event queue is implemented as a linked list.

2. The *Cancel Queue* is a linked list which is used to hold negative events for all LP's for the corresponding process. The cancel queue is used in the exact same way as GTW except that no locks are necessary.

3. The *Priority Queue* is analogous to the unprocessed event queue in GTW and contains events in timestamp order. ROSS also allows the priority queue to be implemented as a calendar queue, heap, splay tree, or avl tree depending on user configuration.

3.4 WARPED

WARPED is the predecessor of WARPED2 and serves as the basis for the design and architecture of WARPED2. WARPED started as a completely processed-based solution with only message passing for communication. Eventually, with the development of multicore processors, each process was extended into multiple threads to further enhance concurrent processing of events. Over the years, with many students developing new algorithms in WARPED2, the complexity of WARPED became unmaintainable, mainly because of the great deal of indirection which, although made for a super configurable and modular design, became too complex for new students to learn it and enhance it.

3.5 Others

3.5.1 The ROme OpTimistic Simulator (ROOT-Sim)

ROOT-Sim is another general purpose Time Warp Simulator that uses message passing via MPI [8]. Like WARPED, ROOT-Sim is a more classic Time Warp implementation with each LP having their own input queues, and output queues. What sets ROOT-Sim apart from other time warp simulators is the internal instrumentation tool, Dynamic Memory Logger and Restorer (DyMeLoR) that can optimize memory usage. DyMeLoR can determine whether the simulation models are better fit for copy-state saving or incremental state saving and transparently switch between them during runtime. Another service that ROOT-Sim offers

is the Committed and Consistent Global Snapshot (CCGS). After each GVT calculation, ROOT-Sim transparently rebuilds a global snapshot of all LP states. Each LP can access its portion of the global snapshot on every GVT calculation. With this service, a simulation model can implement any custom global snapshot algorithm.

3.5.2 ROSS-MT

ROSS-MT [9] is a multi-threaded version of ROSS which is optimized to use shared memory to communicate among threads. The use of message passing with MPI was completely removed and all events are sent by direct insertion into the event queues. To reduce the added contention on the event queues, they are further divided by possible senders. ROSS-MT also optimizes the free memory lists so that they are more NUMA aware. Instead of allocating memory from the receiver processors free list, it is allocated from the senders free list. Furthermore, a LIFO approach is taken to improve cache reuse.

Chapter 4

The WARPED2 Simulation Kernel

This chapter describes the software architecture of the WARPED2 simulation kernel. The main compenents are described and the dependencies between each of them. The modeling API is also described and an example model is shown to illustrate the API.

4.1 The Software Architecture of WARPED2

The WARPED2 simulation kernel uses a modular design to make it more configurable, extendable, and maintainable. All components are configured and created at startup individually and subcomponents are accessed through pointers. Furthermore, WARPED2 is written in C++ so each component can also be an object of derived subclass that implement a well-defined interface by a base class. The main central component is the event dispatcher which is responsible for calling the LP callback methods during the course of the simulation as well as determing when rollbacks and cancellation is necessary and carrying them out. The main components and how they depend on eachother are illustrated in figure 4.1.

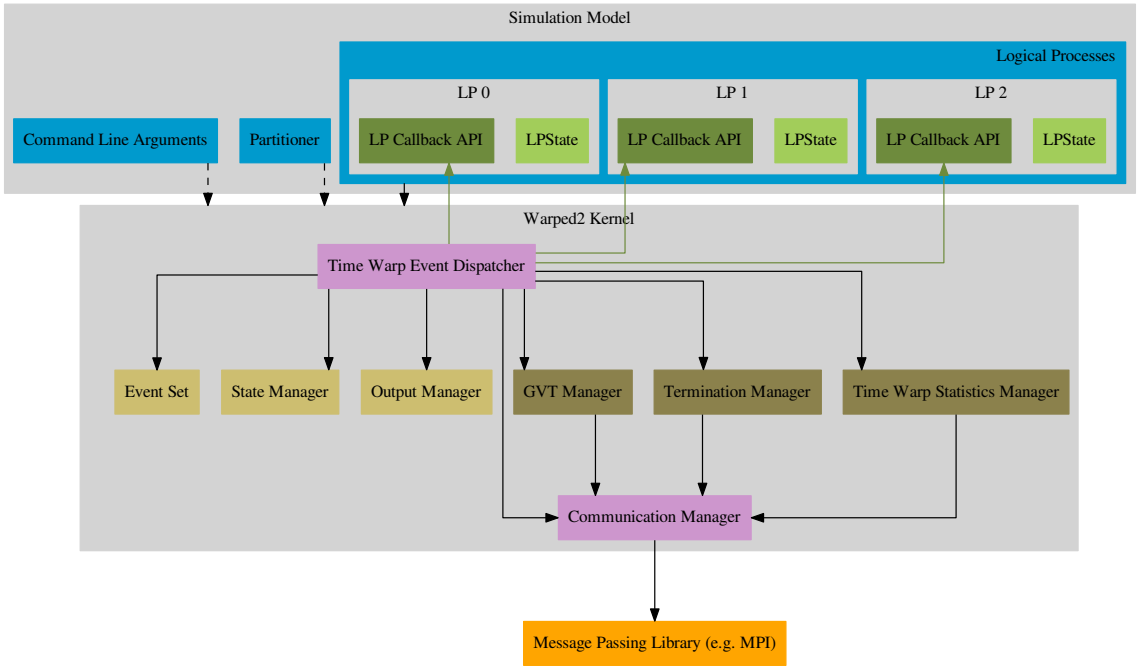


Figure 4.1: Time Warp Components in WARPED2

Warped2 also supports a sequential event dispatcher. The sequential event dispatcher, however does not depend on any other components and just contains a single list of all events since only a single event is processed at a time.

The time warp components can be categorized as either a local time warp component or a global time warp component. The local time warp components are used only for the local control mechanism of the individual LP's such as rollbacks, cancellation and fossil collection whereas the global time warp components are concerned with the global control mechanisms such as GVT, termination detection and statistics counting. The global time warp components must be able to communicate with all other processes in the system so that it is possible to determine the global state of the system.

4.1.1 Local Time Warp Components

The Event Set contains the data structures for all of the unprocessed and processed events for the LPs that are local to the process. This includes the input queues for each LP as well as a scheduling data structure.

The event set provides methods for retrieving the next event, processing a rollback, and fossil collecting the processed events. All the data structures that have pending events must necessarily be thread safe.

The Output Manager contains all of the output queues and implements a single cancellation technique. The output manager base class provides a set of methods that a derived class must implement. The derived class must have methods for adding an event to an output queue, processing a rollback, and fossil collecting old output events. Currently, WARPED2 only implements aggressive cancellation but may support more cancellation techniques in the future.

The State Manager contains all of the state queues and implements a single technique for state saving and state restoration. Just like the output manager, it also hase a base class. The derived class provide methods for saving the state of an LP, restoring the state of an LP, and fossil collecting old states. Currently, WARPED2 only implements periodic state saving.

4.1.2 Global Time Warp Components and Communication Manager

The GVT manager implements an algorithm for determining the Global Virtual Time of the simulation. Because of the nature of WARPED2, the GVT algorithm has a base message passing algorithm with a shared memory subalgorithm. The termination manager impements an algorithm for determining when all processes become inactive and initiates termination when that occurs. A more detailed description of the GVT and termination detection algorithms used in WARPED2 are given in chapter 8. The Time Warp statistics manager keeps track of all local statistics and provides methods for global reductions on the them.

The communication manager provides an interface between the underlying message passing library and the global time warp components in the WARPED2 simulation kernel. Any interprocess communication must go through the communication manager, including remote events that must be sent to another process or received from another process.

```
WARPED_DEFINE_LP_STATE_STRUCT(ExampleState) {
   unsigned int messages_received_;
   ...
};
```

Listing 4.1: Example WARPED2 State Definition

4.2 The Modeling API of WARPED2

The modeling interface of warped2 is a set of abstract base classes that contain methods to be implemented in a derived class in the simulation model. The three main base class types that must be implemented are LogicalProcess, LPState, and Event. Optionally, the user may create a custom partitioner from the Partitoner base class. In the remainder of this section, each class is described in more detail and sample implementations are shown for each.

4.2.1 The LPState Structure

The state of the LP's must be defined with the WARPED_DEFINE_LP_STATE_STRUCT macro which automatically derives from an intermediate structure that defines a necessary clone method so that the model developer does not have to. It is needed to ensure that the warped2 kernel can save a copy of the state and restore the state from a pointer to the LogicalProcess base class. A simple example of a LP state that contains just message counts is shown below in listing 4.1.

If the state contains complex data structures that contains pointers then the default copy constructor and default copy assignment operator will only perform shallow copies. In this case the user must implement a custom copy constructor or a custom copy assignment operator or both. The copy constructor will define the behavior for saving the state whereas the copy assignent operator will define the behavior for restoring the state. Note that the copy assignment operator will most likely not be needed since a shallow copy will usually suffice for restoring the state.

4.2.2 The Event Class

The event base class is used as the basis for creating model specific events. The user must implement at least two methods: (1) `receiverName()` and (2) `timestamp()` . It is necessary so that the

```
class ExampleEvent : public warped::Event {
public:

    ...

    const std::string& receiverName() const { return receiver_name_; }
    unsigned int timestamp() const { return time_stamp_; }

    ...

    std::string receiver_name_;
    unsigned int timestamp_;

    ...

    WARPED_REGISTER_SERIALIZABLE_MEMBERS(cereal::base_class<warped::Event>(this
        ),
                                receiver_name_, timestamp_, ...)
};
WARPED_REGISTER_POLYMORPHIC_SERIALIZABLE_CLASS(ExampleEvent)
```

Listing 4.2: Sample WARPED2 Event Definition

name of the receiver and receive time, respectively, can be obtained for each instance of an event within the kernel. The user must also register all member variables with the serialization API so that a storage order can be defined for events that are sent and received through the message passing system. To do this, the WARPED_REGISTER_SERIALIZABLE_MEMBERS macro is provided. All member variable must be passed to this macro as well as `cereal::base_class<warped::Event>(this)` to ensure that all members that are inherited are also serialized. The order that the members are listed is completely arbitrary and does not matter. In addition, the derived event type must be registered using the WARPED_REGISTER_POLYMORPHIC_SERIALIZABLE_CLASS macro. A basic example of an event implementation is shown below in listing 4.2.

4.2.3 The LogicalProcess class

The most important class definition in the simulation model is the LogicalProcess class. The implementation of the LogicalProcess class defines the callback functions that the event dispatcher with the warped2 kernel calls and thus it defines the behavior of the simulation. The user must include a single LPState as a data member of the LogicalProcess and provide three callback method implementations:

```
class ExampleLP : public warped::LogicalProcess {
public:

   ...

   warped::LPState& getState() { return this->state_; }

   std::vector<std::shared_ptr<warped::Event> > initializeLP() override {
      this->registerRNG(this->rng_);
      std::vector<std::shared_ptr<warped::Event> > events;
      ...
      return events;
   }

   std::vector<std::shared_ptr<warped::Event>> receiveEvent(const warped::
       Event& event) {
      ++this->state_.messages_received_;
      std::vector<std::shared_ptr<warped::Event> > response_events;
      ...
      return response_events;
   }

   ExampleState state_;
};
```

Listing 4.3: Example WARPED2 LogicalProcess Definition

1. The `initializeLP` method is called to perform any initializations of an LP that must be done prior to the start of the simulation and must return a set of initial events.

2. The `receiveEvent` method is called to perform the forward computation based on the event that is passed. The implementation of this method must process the event by updating the state of the LP and returning a set of new events with future timestamps.

3. The `getState` method provides a way for the warped2 kernel to get the current state of the LP so that it can be saved in the state queue.

It is necessary that at least one LP has an initial event that is returned by initializeLP, otherwise no events can be received and simulation will terminate immediately. Also note that the it will be called once for *every* LP instance so it is possible that initial events are returned only in some cases. An example of a LogicalProcess implementation is shown below in listing 4.3.

```
class ExamplePartitioner : public Partitioner {
   std::vector<std::vector<LogicalProcess*>>
   partition(const std::vector<LogicalProcess*>& lps, const unsigned int
       num_partitions) {
       std::vector<std::vector<LogicalProcess*>> parts;
       ...
       return parts;
   }
};
```

Listing 4.4: Example WARPED2 Partitioner Definition

4.2.4 The Partitioner class

The warped2 kernel already provides a set of partitioners but the model developer can define their own partitioner that is customized for the model. The user must derive from the Partitioner base class and implement just a single method which takes a vector of all LPs and the number of partitions desired and returns a vector of vectors of LP's. In general, the partitioner should work for any number of partitions and not impose any constraints because the partition method is called back from the kernel. A simple template of a partitioner is shown in listing 4.4.

4.2.5 Random Number Generation

If the simulation model must use random number generators, then they must all be registered with the warped2 kernel so that the state of the random number generator can be saved and restored in case of rollbacks and provide deterministic result. The random number generators can be any type as long as they implement the `<< operator` and `>> operator` to allow the kernel to save and restore the internal state of the random number generator. To register the random number generator, the registerRNG template function must be used which is a member of the LogicalProcess class. All LP's must have separate random number generators and must be registered in the initializeLP callback function as shown in listing 4.3.

4.2.6 Command Line Arguments and the Kernel Entry Point

Once all the necessary structures and classes have been defined, the model's main function must be implementd which is where all calls into the kernel are made. First, the model can create specific command line

arguments but they must be registered with the kernel. This must be done first so that it can be passed to the constructor of a `Simulation` instance and so that the command line arguments can be displayed without unnecessarily running a simulation. The kernel uses a third-party library called TCLAP for command line arguments. Then, after setting up the command line arguments, all of the LP objects and optionally a partitioner object must be instantiated and passed to the kernel through the `simulate` method of `Simulation` object. Two versions of the simulate methods are available, one for a model with a custom partitioner and one without as listed below:

1. `void simulate(const std::vector<LogicalProcess*>& lps);`

2. `void simulate(const std::vector<LogicalProcess*>& lps,`
   `std::unique_ptr<Partitioner> partitioner);`

A sample implementation of a models main function is shown in listing 4.5.

```
int main(int argc, const char **argv) {
   unsigned int num_lps = 10000;

   TCLAP::ValueArg<unsigned int> num_lps_arg("o", "lp-count", "Number␣of␣lp's"
      , false,
                                     num_lps, "unsigned␣int");
   std::vector<TCLAP::Arg*> cmd_line_args = { &num_lps_arg };
   warped::Simulation simulation {"Sample␣Simulation", argc, argv,
      cmd_line_args};

   num_lps = num_lps_arg.getValue();
   std::vector<SampleLP> lps;
   for (unsigned int i = 0; i < num_lps; i++) {
      std::string name = std::string("LP_") + std::to_string(i);
      lps.emplace_back(name, 1, i);
   }

   std::vector<warped::LogicalProcess*> lp_pointers;
   for (auto& lp : lps) {
      lp_pointers.push_back(&lp);
   }
   simulation.simulate(lp_pointers);

   return 0;
}
```

Listing 4.5: Sample WARPED2 Main Definition

Chapter 5

Experimental Setup

5.1 x86 SMP Nodes and Cluster

The single node SMP experiments will be carried out on a

5.1.1 State Saving

5.1.2 Fossil Collection

5.1.3 Memory Allocation

5.1.4 Pending Event Set

5.1.5 Partitioning

5.2 ARM big.LITTLE nodes

		Intel Core i7-4770	Intel Xeon E5410	Exynos 5422	
				Cortex-A15	Cortex-A7
Processor	ISA	x86_64	x86_64	ARMv7	
	# Cores	4	8 (2x4)	4	4
	# Threads	8	8	8	
	Frequency	3.4 GHz	2.33 GHz	2.0 GHz	1.4 GHz
	L1 Data Cache	32kB	32kB	32kB	32kB
	L1 Inst Cache	32kB	32kB	32kB	32kB
	L2 Cache	256kB	6MB	2MB	512kB
	L3 Cache	8MB	N/A	N/A	
Memory	Type	2 x DDR3-1333/1600	?	2 x LPDDR3-933	
	Max Bandwidth	25.6 GB/s	?	14.9 GB/s	
Runtime	Compiler	?	?	?	
	MPI Version	?	?	?	

Table 5.1: Summary of Assessment Platforms

5.3 Simulation Models used for Assessment

5.3.1 PCS

The model describes a type of wireless communication network known as a Portable Cellular Service (PCS) network. A PCS network provides services for a number of *portables* which are the subscribers to the network. The service area of the network is divided into *cells* and a single *port* covers a cell which has a certain number channels that it can allocate for calls. When an incoming or outgoing call is made, a port must allocate a channel from the port if one cannot be allocated then the call is *blocked* [10]. The only type of logical processes in the PCS simulation are the cells. The cells in the WARPED2 model form a rectangular grid as shown in figure 5.1. Portables can only move to other cells from an adjacent cell with a wrap around occuring at the edges.

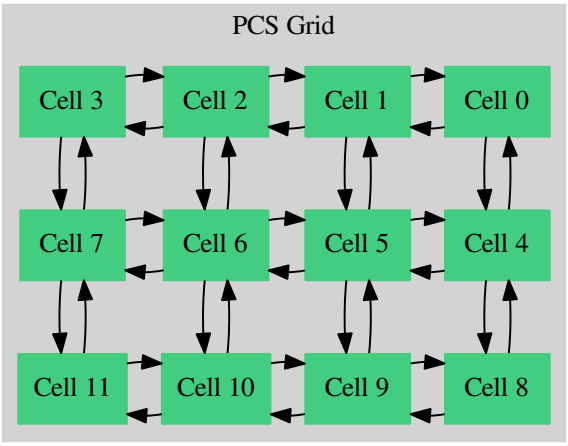


Figure 5.1: PCS Model Logical Processes

The portables stay within a cell for a period of time which follows a poisson distribution before moving to another cell. If the all cells are busy in the cell that the portable is moving to then the call is dropped. This is called a *handoff block*.

Every LP keeps track of the following state variables:

- Number of idle channels
- Number of call attempts
- Number of channel blocks
- Number of handoff blocks

Call arrivals to a portable also follow a poisson distribution. The cells in the model generate all of their own incoming calls in a self-initiating process. Four types of events are used to model the network: (1) NextCall, (2) CallCompletion, (3) PortableMoveOut, and (4) PortableMoveIn . The NextCall, CallCompletion, and PortableMoveOut events are all sent to self whereas the PortableMoveIn event is sent to an adjacent cell based on a random variable with uniform probability.

5.3.2 Traffic

The traffic model describes a grid of intersections and the movement of cars through the intersection and between the intersections. All intersections are four way intersections and have three lanes in each direction. The LPs in the traffic model are the intersections and form a rectangular grid in the same way as that of the PCS model. The layout is illustrated in figure **??**.

The simulation starts with a uniform distribution of cars at each intersection and each car is a assigned a destination which it finishes at. With each arrival at an intersection the car always goes in the direction that gets it closer to its destination. The number of cars going out of an intersection in the same direction is limited to a maximum number due traffic on the approaching road. An incoming car that cannot travel in a specific direction is forced back in the same direction it came from and tries another route. For each intersection, the state consists of:

- Number of cars coming into the intersection from each direction
- Number of cars going out of the intersection to each direction
- Total cars arrived at the intersection
- Total cars finished at the intersection

A car goes through three event phase in every intersection as shown in figure 5.2. The three types of events are: (1) Arrival, (2) Direction Select, and (3) Departure . The arrival and departure events are mainly used for simple state updates and timing of the next events. The direction select event is where the complexity of the simulation is and determines the direction that the car should go in order to reach its destination or whether the car should take another route due to traffic on the road.

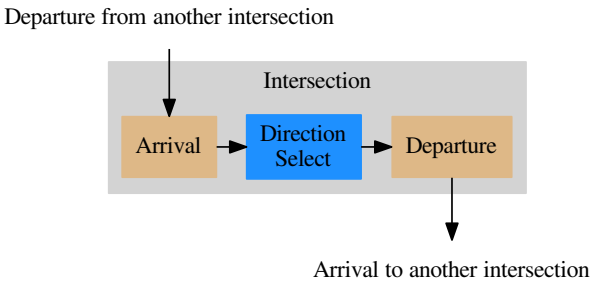


Figure 5.2: Intersection Event Sequence

All events in the traffic model follow an exponential distribution and a single event is generated with the processing of each event. The departure and direction select events are self-generated but the arrival event is from an adjacent intersection.

5.3.3 Epidemic

The epidemic model describes the spreading of an infectious disease across a set of geographic locations in a region and across a set of regions. The logical processes in the simulation are geographic locations which represent a portion of the population. The interactions between people in different geographic locations and regions are modeled with a diffusion network. An abstract view of the simulation model is shown below in figure 5.3.

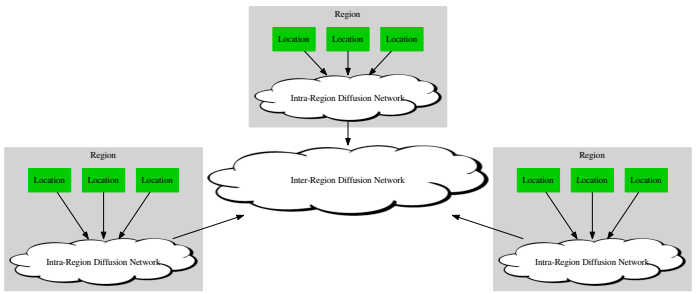


Figure 5.3: Epidemic Model

The model is based on reaction-diffusion model [11]. The reaction process models the behavior of the disease within a person as well as the transmission of the disease between people within the same location. A probabilistic reaction function defines the behavior of the disease between individuals [12]. The disease within an individual is modeled by a Probabilistic Timed Transition System (PTTS) [12] which is a finite state machine describing the disease states and transitions between states. Together, the interhost and intrahost models of the disease form the *disease model*. The diffusion network models the social interactions of people between locations and between region [12].

5.3.4 Airport

The airport model describes the departure and arrivals of airplanes between a connected set of airports. The logical processes represent airports and the events are simply departures and arrivals. For simplicity, the airports are connected in a rectangular grid and can only fly to aiports to the north, east, south, or west of the aiport departed from. The time spent between takeoff and landing and between landing and takeoff both follow an exponential distributions.

Chapter 6

WARPED2 Data Structures and Their Organization

6.1 Pending Event Set Data Structures

The pending event set is the set of all unprocessed events. Every process contains a logically seperate pending event set for a dedicated set of LPs. Exhanging of events between LPs of different processes is achieved through the manager thread of each processes.

Each LP has an unprocessed queue which contains both positive events and anti-messages and remains sorted at all times. The anti-messages are given priority over their positive counterparts to prevent any unnessary rolbacks and prevent cascading of rollbacks that can cause instability [13]. To avoid any copying, the unprocessed queue only contains pointers to the unprocessed events which are allocated by the simulation model. The worker threads or manager thread send events to LPs by simply inserting the pointers into their unprocessed queue.

A second type of data structure, an *LTSF (Lowest TimeStamp First) Queue*, provides order among events from multiple LPs. At most, a single event from each LP is *scheduled* into a single LTSF queue. Scheduling an event to an LTSF queue means only inserting a pointer into the LTSF data structure with no removal from the unprocessed queue. The pending event set contains one or more LTSF Queue to process events from.

A third type of data structure is also used to keep track of the events that have been scheduled or are

currently being processed and is organized by receiving LP. It is used for two main reasons. First, it provides a way to determine if a smaller event is scheduled for an LP upon the arrival of another event into its unprocessed queue. The new event can be scheduled in place of the already scheduled event if that occurs. Second, it serves to prevent multiple worker threads from processing events from the same LP which could cause out of order committing of events violating the local causality constraint [1].

Figure 6.1 illustrates the pending event set data structures in WARPED2.

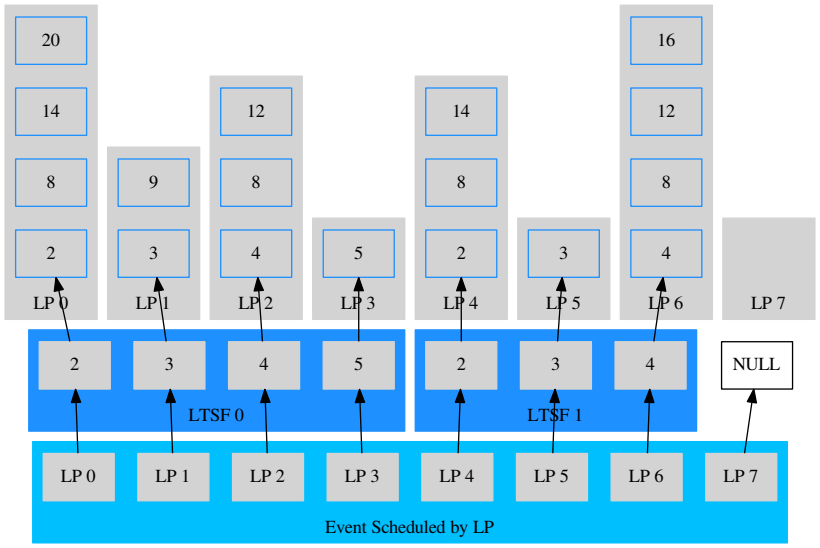


Figure 6.1: WARPED2 Pending Event Set Date Structures

6.2 Processing Events

One or more of the worker threads are assigned to each LTSF Queue to process events from and they all follow the exact same procedure. First, an event is removed from an LTSF queue to be processed but remains in the unprocessed queue until it is processed. The event is then checked to see if it is a straggler, and rolled back if so. An anti-message is also considered a straggler if its positive counterpart was the last processed event since WARPED2 makes the assumption that events cannot be sent out of order. If the event is an anti-message, the positive counterpart is assumed to be in the LPs unprocessed queue after rolling back and is

cancelled out and a new event is scheduled. If the event is positive it is processed normally, the state of the LP is saved, and new events are sent to other LPs. The event is then inserted into the processed queue of the LP and a new event is scheduled if one exists for the LP. Lastly, the data structure that keeps track of scheduled events by LP is updated. To summarize the worker thread processing loop is shown in psuedocode in algorithm 2.

Algorithm 2: WARPED2 Main Event Processing Loop

```
   while termination not detected do
1      e ← getNextEvent()
       lp ← receiver of e

       if e < last processed event for lp then
           rollback lp

       if e is an anti-message then
           cancel event with e
3          schedule new event for lp
           continue

       process event e
       save state of lp
2      send new events
3      replace scheduled event for lp
```

When events are sent to an LP that is local to the process, they are directly inserted into the unprocessed queue and checked against the currently scheduled event. If the new event is less than the scheduled event, the scheduled event is replaced. When events are sent to an LP that is not local to the process, they are inserted into a shared data structure called the *remote event queue*. The manager thread removes the events from the remoted event queue, serializes them into messages and sends them to remote processes through the communication manager.

6.2.1 Ordering of Events

The time warp system assumes that every event is labeled with a totally ordered clock value based on virtual time [2]. This is important so that causal dependencies are preserved and so that simulation results are deterministic [14]. WARPED2 uses a four tuple scheme which provide a total ordering of events:

1. Receive Time
2. Send Time
3. Sender LP Name
4. Generation

The last three serve as a tie breaker for simultaneous receive times. The send time is necessary to ensure correct causal dependencies. It is analogous to a Lamport logical clock [15] in real time distributed systems but for virtual time systems. The send time works for this as long as LPs only send a single event with the same send time to any individual LP. The sender LP name is necessary to ensure that order is determined between events received with the same receive time and send time from different senders. Without it, it is possible that different results could occur on different runs of the simulation [14]. The generation is necessary for distributed systems to differentiate between the same events which could be resent after rolling back [14]. In WARPED2 it is implemented as a single counter per LP and keeps track of the number of events that have been sent from the LP. The count is then tagged in the event when it is sent.

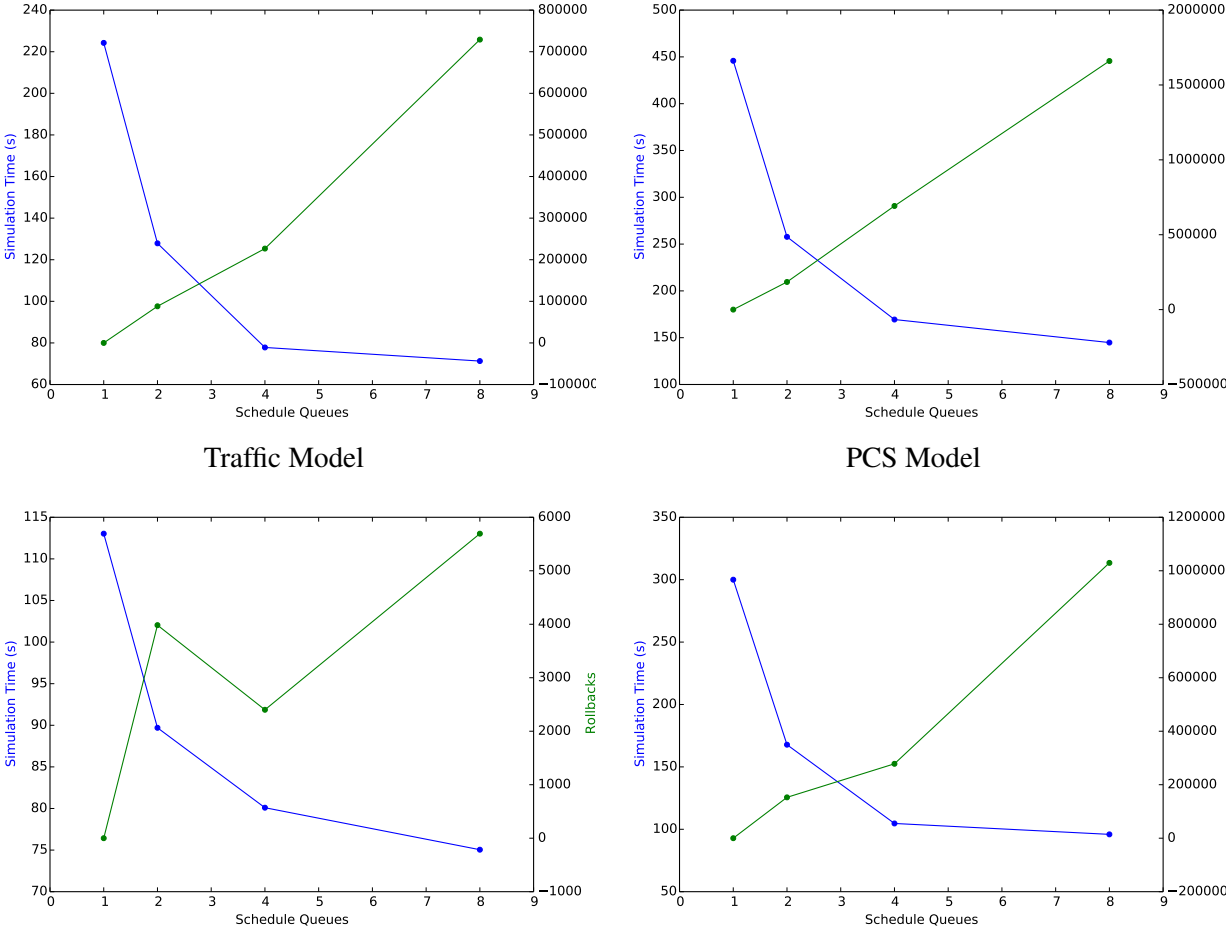
6.2.2 LTSF Replication

The number of LTSF queues that are shared by a single worker thread is configurable in WARPED2 because the best configuration is not really known. If all worker threads share a single LTSF queue then a huge contention point is created and performance will suffer. At the opposite extreme, however, with all worker threads processing events from a seperate LTSF queue, the the critical path of the simulation will be spread out and the number of rollbacks can increase significantly. To make matters more complicated this tradeoff can vary as the event processing algorithms and LTSF Queue data structures in WARPED2 evolve.

The plots shown in figure 6.2 provide some initial insight on the best number of LTSF Queues for each of the simulation models. All simulations were run with on a single Intel Core i7 machine with 8 worker threads.

From these results it is apparent that contention to the LTSF queues far outweigh the effects of increased rollbacks. This can be explained by looking back at the current event processing algorith in algorithm 2. The lines with a number on the margin indicate possible access to the LTSF queue. For every event the LTSF

queue is accessed at least 3 times: (1) getting the next event; (2) sending new events; and (3) rescheduling new events .



Traffic Model

PCS Model

Epidemic Model

Airport Model

Figure 6.2: Schedule Queue Performances

6.3 Data Structures to Support Rollback and Cancellation

Jefferson [2] described the rollback and cancellation process in terms of three data structures: the input queue, output queue, and state queue. The data structures that are implemented in WARPED2 follows pretty closely to Jefferson's description except that the input queue holds both unprocessed and processed events whereas WARPED2 has two seperate data structures to allow easier scheduling of new events.

6.3.1 Processed Queue

The processed queues are pretty straightforward and only contain pointers to processed events for the owning LP. When a straggler event is detected, the incorrectly processed event pointers are moved back to unprocessed queue where the event can be cancelled out by an anti-message or reprocessed as necessary. Anti-messages cannot be processed so they will never exist in the processed queue.

6.3.2 Output Queue

An output queue is used to hold previously sent events by a single LP. They are necessary so that the LP can send anti-messages during a rollback. The output queue in WARPED2 contains a set of tuples that each contain a pointer to a sent event and a pointer to the source event that triggered the sent event. The LP determines which events should be sent as anti-messages by comparing the straggler event with the source event.

6.3.3 State Queue

The state queues in WARPED2 contain a three-tuple value for each state saved. The three-tuple consists of (1) a copy of the LPs state at the time it was saved, (2) a copy of the internal states of the LPs random number generators, and (3) a pointer to the event that was processed just before the state was saved. . The internal states of the random number generators are saved in a linked list and are necessary so that simulations can yield deterministic results. Just as with the output queue, the source event is used for comparison against straggler events except that it is for deciding the state which restore the LP back to.

6.3.4 Organization of Data Structures

Pointers are used in all data structures to avoid unnecessary copying and to save memory space. There can be multiple pointers to the same events from the processed queue, unprocessed queue, output queue, or state queue as shown in figure 6.3. This can be rather complex to deal with when freeing memory so WARPED2 uses smart pointers objects that count references to the dereferenced memory location.

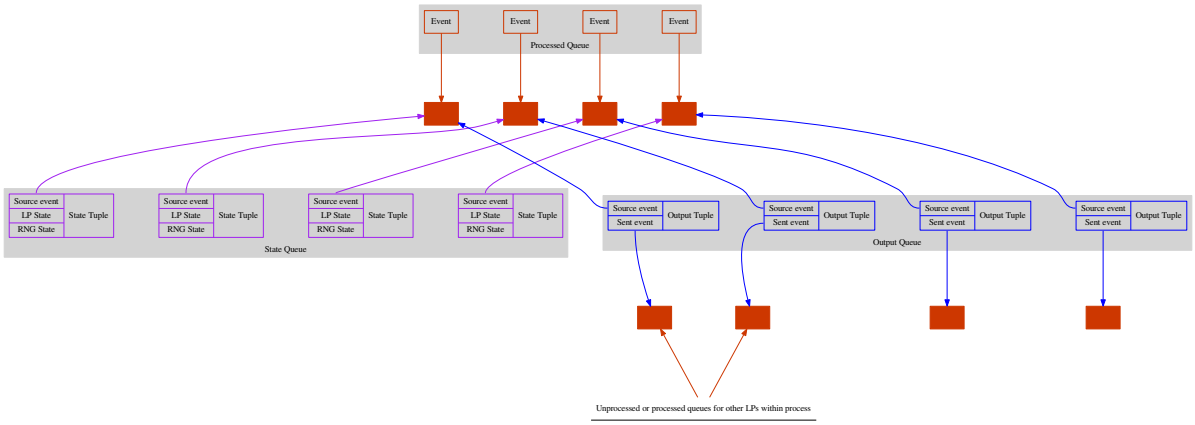


Figure 6.3: Rollback and Cancellation Data Structures in WARPED2

6.4 Protecting Access to Shared Data Structures

6.4.1 Motivation

Worker threads must be able to access shared data structures without the possibility of race conditions. To deal with this, WARPED2 uses mutexes to serialize worker thread access to any section of code that accesses the LTSF queue. This method, however significantly slows down access to the LTSF queues. With the results from the last section, it is obvious that quick access to the LTSF queues is necessary for the best performance.

Traditionally, mutexes are implemented in such a way that if a thread cannot acuire the lock, the thread is put to sleep and rescheduled for execution at a later time. This method, however, can waist a lot of unnecessary time putting the thread to sleep and waking it back up again if each thread only holds the lock for a short period of time. WARPED2 is the perfect example of this behavior since accesses to the LTSF

queues only perform very simple and fast operations.

A *spinlock* is a type of mutex that does not force threads to be rescheduled but instead continue to attempt to aqcuire the lock over and over until it successfully acquires the lock or the timeslice of the thread expires. The downside of spinlocks is that the CPU time can be waised, especially as the number of contending threads increase. Furthermore, if the number of total threads is larger than the number of available processors, spinlocks are event worse because a thread can be involuntarily pre-empted while holding the lock.

6.4.2 Spinlocks in WARPED2

There are many ways to implement spinlocks like test-and-set or exchange based locks, queueing locks, and ticket locks to name a few. Test-and-set or exchange based locks are the simplest to implement but do not guarantee fairness among threads. Queueing based locks are more complicated to implement but also guarentee fairness and are more scalable. However they have a higher latency for uncontended aqcuisition. Ticket locks, on the other hand are simple to implement, provide fairness, and generally have good performance for a small number of threads.

Due to the advantages mentioned above, WARPED2 implements ticket spinlocks.

Algorithm 3: Ticket Lock Procedures

```
lock
    my_number ← fetchAndIncrement(next_number)
    repeat
        // Do nothing
    until ticket = my_number

unlock
    fetchAndIncrement(ticket)
```

Chapter 7

Partitioning

7.1 Overview

Partitioning the work in a distributed system is always important to minimize interprocess communication. In parallel discrete event simulation, the work is usually partitioned by the LPs. Each process in the system can then process events from a dedicated subset of LPs.

In a message based system, communication latency is much larger than the frequency of computation. The disparity of of communication time and compute time greatly increases the chances of rollbacks occuring in a time warp system. Partitioning can greatly reduce communication between processes, effictively pushing each process closer to a sequential simulation.

In a shared memory system, partitioning can aid reducing the number of simulateous accesses to shared data structures (e.g. unprocessed queues) by localizing access to less threads. Although it may not reduce the total number of accesses to a data structure, it can force worker threads to access the same subset of data structures reducing contention. This can also enhance the number of cache hits by increasing locality.

7.2 Partitioning in WARPED2

WARPED2 uses multiple worker threads that communicate through shared data structures within processes that communicate through message passing. Therefore, to get the best performance, two levels of partitioning should be done. The first level of partitioning is for allocating LPs to processes and the second level is

for allocating LPs to LTSF queues. Figure **??** illustrates partitioning in WARPED2 and how worker threads fit with it.

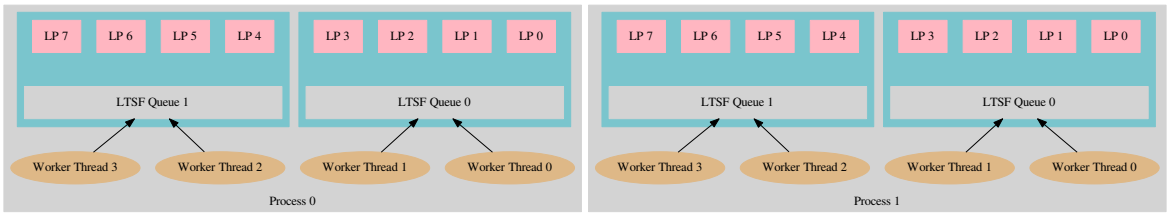


Figure 7.1: Partitioning in WARPED2

WARPED2 currently supports thee types of partitioning: (1) round-robin, (2) model-specific (3) profile-guided .

7.2.1 Process Partitioning

7.2.2 LTSF Queue Partitioning

Each process further partitions the LPs by the LTSF Queue that the LPs schedule their events to. It is effectively partioning among worker threads if each worker thread has its own LTSF Queue and has no effect if a single LTSF is shared among worker threads.

Chapter 8

GVT and Termination Detection

Algorithms that compute the Global Virtual Time (GVT) and detect termination conditions are both examples of algorithms that can be solved by determining the global state of a distributed system. The global state of a system is defined as the combination of all the local states of the processes in the system and all messages in transit. Global state determination algorithms are also commonly used for deadlock detection, garbage collection, debugging, and checkpointing for failure recovery in distributed systems.

8.1 Global Snapshots

Chandy and Lamport [16] described a basic global state algorithm by using *global snapshots* for distributed systems that use only FIFO channels for communication. To start the algorithm, an initiator process records it's state and sends a control token out of all outgoing channels. When a process receives a control token, and it hasn't yet recorded its state, the process records its state and sends more control tokens out of all its outgoing channels. The algorithm terminates at each process when it has recieved a token through all of its incoming channels.

Lai and Yang [17] describe an algorithm for non-FIFO systems which computes a consistent cut by piggybacking a control bit onto basic messages. The control bit is used to indicate whether or not the sending process has recorded its state. The processes can be explained in terms of colors. A process that has not recorded its state is colored white and a process that has recorded its state is red. White processes send

white messages and red processes send red messages. The control bit in the messages indicates the color. All processes are initially white and turn red when a red message arrives. When a red message arrives at a white process, the process must record its state BEFORE actually receiving the message. The snapshot only relies on basic messages and no control tokens are used. This algorithm has several drawbacks. First, it assumes that every process will eventually receive a red message and record its state which is not guarenteed. Second, to ensure transient messages are recorded, the processes must record all incoming and outgoing messages and send them to other processes within the basic messages. That way the transient messages can be calculated by differences in incoming and outgoing messages.

Mattern [18] extended the algorithm by Lai and Yang by adding a seperate control token which is used to create two cuts by cirulating the control token to every process twice. The control token is used to color the processes instead of the basic messages. This guarentees that every process will eventually record its state because the token is always circulated. Furthermore, processes do not have to keep track of sent and received messages. Instead counters can be used to keep track of the differences in sent and received white messages at each process. The control token can then accumulate the counts. When the white message counts accumulate to zero when the token arrives back at the initiator proccess, it can be determined that the snapshot is complete. These counters can be *vector counters* or *scalar counters*. Vector counters keep track of messages to/from each process individually whereas scalar counters keep track of just a single count at each process. When the algorithm is complete, only the initiator process can produce a global snapshot so if all processes must use the snapshot, it must be broadcasted to the other processes.

8.2 GVT

Although GVT algorithms can be implemented with the basic global snapshot algorithms described above, it is usually more efficient to build custom solutions on top of the basic algorithms. In a GVT algorithm, the local minimum clock of process is the local state and the basic messages are events that are sent between processes. This section first describes the key ideas that must be considered when developing a GVT algorithm. Then, a few of the classic GVT algorithms and modern GVT algorithms that are commonly used in practice today are described. Finally, the algorithm that is implemented in WARPED2 is described as well as

the reasons for choosing the approach.

GVT algorithms can be synchronous or asynchronous. In a synchronous GVT algorithm, since all other computation will be blocked, event processing will be halted. Synchronous GVT algorithms are usually very simple to implement but halting the event processing can be very costly. On the other hand, asynchronous GVT algorithms calculate GVT concurrently with event processing. For this reason, asynchronous GVT algorithms perform much better than synchronous GVT algorithms but are harder to implement because special cases must be considered.

There are two special cases that must be considered in any GVT algorithms:

Transient Message Problem: is caused by messages(events) that have been sent by the sending process but not yet recieved by the receiving process. If not carefully considered in a GVT algorithm, these *transient messages* can be completely missed which could lead to an erroneous GVT calculation if they contain the minimum timestamp of all other events in the system.

Simultaneous Reporting Problem: is caused because processes can report their local minimum clock at different points in real time. Consideration must be taken into account to ensure that a process does not report its local minimum clock value and then receive an event from a process that has not yet reported its local minimum clock value, completely missing the event.

There are many synchronous and asynchronous GVT algorithms that have been developed over the years that all have some method of either solving these problems.

8.2.1 Asynchronous GVT Algorithms

Message Passing Algorithms

Most of the time warp systems in use today are based on message passing and classically GVT algorithms have been designed for message passing systems. In this section Samadi's and Mattern's message passing algorithms are discussed and in the next section Fujimoto's shared memory GVT algorithms is discussed as well as the Seven O'Clock algorithm which is an extension of Fujimoto's shared memory algorithm extended for distributed memory systems.

Samadi's Algorithm [19] in the most general form uses acknowledgements for all events that have been received. All processes must track all events that have been sent but have not been acknowledged. Furthermore, the received messages must also be tracked so that acknowlegements can be sent. All transient message can then be calculated from the tracked messages. A process initiates the GVT algorithm by broadcasting a start message to all processes. After this start message is received by a process, it is marked(colored) and all acknowledgements sent from a marked process are also marked(colored). All processes then calculate their local minimum by taking the minimum of the unacknowledged received events, the marked acknowledgments sent, and the local simulation clock. Marking the acknowledgements sent after the start of the GVT calculation ensures that the simultaneous reporting problem will not occur.

Mattern's Algorithm [18] for GVT calculation is an extension on his general snapshot algorithm that was described above. The white message counts are used to determine whether transient messages are still in the sytem. They also serve as the basis for determining when the snapshot is complete which occurs when the accumulated white message count of all processes is zero. To ensure that the simultaneous reporting problem does not occur, red messages received at a white process are recorded and used in the local minimum value. The algorithm is initiated by sending a control token to all processes in some defined order. The token accumulates the white message counters, local minimum clocks, and minimum red message timestamps. When the accumulated count reaches zero and the token is back at the initiator process, the GVT is approximated using the minimum of the accumulated clocks and timestamps.

Other GVT Algorithms that are based on message passing model are usually based on the same ideas from Samadi's algorithm or Mattern's algorithm. Many algorithms are just extensions or variations of the algorithms that aim to optimize it in some particular way.

Shared Memory Algorithms

Fujimoto's Algorithm [5] is a fast GVT algorithm that exploits properties of shared memory architectures. In most shared memory architectures, processors cannot observe memory operations in different orders. For this reason, it is not possible to have transient messages if a shared data structure is used to communicate between tasks running on different processors. Furthermore, a shared flag variable can be

used to initiate the GVT algorithm. However, it is still possible that the flag can be read at different times, so the simultaneous reporting problem can still occur. To solve the simultaneous reporting problem, two things must be done, First, the start flag is checked after sending events and recorded if the sending task has not yet reported its local minimum value and is eventually used when the reporting is done. Second, the start flag must be read into a temporary local variable before obtaining a new event to process.

Seven O'Clock Algorithm [20] is an extension of Fujimoto's algorithm for distributed memory systems. Although the algorithm can be used in message passing systems, the algorithm does not use any messages and is still uses shared memory ideas. The key idea in the algorithm is that all processors in a distributed system all have a consistent view of wall clock. Hence, the processors can carry out an operation atomically, without having to explicitly interact. The atomicity can be achieved by using cycle level counters which are available on most modern architectures. Unlike Fujimoto's algorithm though, transient messages can still be missed in the calculation. To solve that problem, each processor must wait a small time interval which is calculated based on the worst round trip time for network transactions.

8.2.2 Synchronous GVT Algorithms

Global Reductions provide a good way to do a synchronous minimum calculation which and is a common way to implement a synchronous GVT algorithm. ROSS implements a synchronous GVT algorithm which uses global reductions [21]. First, to prevent the transient message problem, a global reduction on message counts of each process is performed until it reaches zero which is guarenteed because the reduction acts as a barrier for each process. The message counting is necessary because asynchronous communication is used for sending and receiving events. When the messages count reaches zero a global reduction is performed on the local minimum timestamp of all processes to give a GVT value which is broadcast to all processes. This algorithm is efficient when time warp simulation are run on large supercomputing platforms like the blue gene machine which perform collective operations very quickly.

8.2.3 GVT Calculation in WARPED2

To calculate the GVT in WARPED2, Fujimoto's shared memory algorithm is used between worker threads and a variation of Mattern's algorithm with scalar message counters is used between processes. Hence,

Fujimoto's algorithm is a subalgorithm of Mattern's algorithm. The details of the Mattern's algorithm implementation in WARPED2 is detailed below as well as how it is used with Fujimoto's algorithm.

In the WARPED2 implementation, scalar counters are used to keep track of message that are sent that carry the initial color. That means that each process keeps track of the number of sent messages minus the number of received messages to and from all other process as a single counter. The reason that scalar counters are used over vector counters is that vector counters require that communication is stopped after receiving a control token until all known transient messages bound for the receiving process are received. With scalar counters, this is not possible because it is only a single counter. However, it is possible that more than two rounds of the control token will be necessary. This is usually not a problem in practice though since two rounds is usually sufficient. With scalar counters, each process must maintain a counter for both white messages *and* red messages so that counters can be consistent between multiple runs of the algorithm. Moreover, the role of colors must be switched between successive runs.

In the description of the WARPED2 algorithm, the colors will be described as *initial* and *final*. Each process also keeps track of its current color, a minimum timestamp of an event received with the final color, and temporary variables to hold accumulated values from the token. The variables that are maintained at each process and are used in the succeeding discussion are listed below in algorithm 4.

Algorithm 4: Process Variables in WARPED2 Mattern Implementation

$\boldsymbol{ts_{min}}$ **:** Minimum timestamp of event messages received with final color
$\boldsymbol{color}$ **:** Current color of the process
$\boldsymbol{color_{initial}}$ **:** Intial color of all processes
$\boldsymbol{msgcount_{initial}}$ **:** Messages sent minus messages received with initial color
$\boldsymbol{msgcount_{final}}$ **:** Messages sent minus messages received with final color
$\boldsymbol{clock_{min}}$ **:** Temporary variable to hold accumulated minimum clock from all processes
$\boldsymbol{msgcount}$ **:** Temporary variable to hold accumulated initial color message count from all processes

The event messages that are sent between processes contain the format:

$< sender, receiver, mcolor, ... >$

where $sender$ is the id of the sender process, $receiver$ is the id of the reciever process, and $mcolor$ is the color of the message. The message color is the color of the sending process at the time it is sent. The message count for the current color is also incremented when it is sent. Psuedocode illustrating this procedure is shown in algorithm 5.

Algorithm 5: Event Message Send

$mcolor \leftarrow color$
if $color = color_{initial}$ **then**
  $msgcount_{initial} \leftarrow msgcount_{initial} + 1$
else
  $msgcount_{final} \leftarrow msgcount_{final} + 1$

When an event is received by a process the message count for the current color of the process is decremented. If the color carried by the message is the final color in the algorithm, indicating that the sender has already reported a local minimum value, then the timestamp of the event is recorded as ts_{min}. Psuedocode illustrating the receiving procedure is show in algorithm 6.

Algorithm 6: Event Message Receive

if $color_{message} = color_{initial}$ **then**
  $msgcount_{initial} \leftarrow msgcount_{initial} - 1$
else
  $msgcount_{final} \leftarrow msgcount_{final} - 1$
  $ts_{min} \leftarrow \min(ts_{min}, ts_{event})$

The control token that is sent between processes contains the format:

$< sender, receiver, mclock, msend, mcount >$

where $mclock$ is an accumulated minimum of all local minimum clocks at all processes visited by the token, $msend$ is an accumulated minimum of all ts_{min} values at all processes visited by the token, and $mcount$ is an accumulated sum of $msgcount_{final}$ from all processes visited by the token.

In the WARPED2 implementation, the token is passed in logical ring fashion to increasing process ids and is initiated by process with id 0. When the token reaches process $N - 1$, where N is the number of processes in the system, it is sent back to process 0. To start the algorithm, process 0 toggles its color, resets minim values to infinity, stores the initial color message count into a temporary variable $msgcount$, and starts Fujimoto's algorithm. When Fujimoto's algorithm is complete, it will send the token $< 0, 1, lvt_{min}, ts_{min}, msgcount >$ with lvt_{min} being the result of Fujimoto's algorithm. Psuedocode illustrating the start procedure is shown in algorithm 7.

When a token is received at a process that is not the initiator and it still has the initial color then the color

Algorithm 7: Mattern Algorithm Start Procedure

if $color = WHITE$ **then**
  $color \leftarrow RED$
else
  $color \leftarrow WHITE$
$ts_{min} \leftarrow \infty$
$clock_{min} \leftarrow \infty$ $msgcount \leftarrow msgcount_{initial}$
$msgcount_{initial} \leftarrow 0$
$lvt_{min} \leftarrow$ `doFujimoto()`
`SendToken` $(0, 1, lvt_{min}, ts_{min}, msgcount)$

is toggled. Then, the minimum timestamp received from final colored messages, the minimum clock value, and the initial message count for the process are accumulated with the token's value. The initial message count is then reset so it will not be used in the next round and Fujimoto's algorithm is initiated. When Fujimoto's algorithm is completed, the token is sent to the next process with all of the accumulated values including lvt_{min} calculated from Fujimoto's algorithm. This procedure is illustrated in psuedocode shown in algorithm 8.

Algorithm 8: Mattern Control Token Receive Procedure: Non-initiator Node

if $color = color_{initial}$ **then**
  $ts_{min} \leftarrow \infty$
  if $color = WHITE$ **then**
    $color \leftarrow RED$
  else
    $color \leftarrow WHITE$
$ts_{min} \leftarrow \min(ts_{min}, msend)$
$clock_{min} \leftarrow \min(clock_{min}, mclock)$
$msgcount \leftarrow msgcount_{initial} + msgcount$
$msgcount_{initial} \leftarrow 0$
$lvt_{min} \leftarrow$ `doFujimoto`
`SendToken` $(i, (i+1) \bmod N, \min(lvt_{min}, clock_{min}), ts_{min}, msgcount)$

When a token is received at a the initiator process, it can be assumed that a minimum clock value has already been included in the token that is received. If the accumulated message count for the initial color has reached zero, then the GVT can be appoximated and the algorithm is terminated. If there are still transient message that were not accounted for, the initiator process initiates a new round until the resulting message

count accumulation is zero when received at the initiator. The GVT is approximated at the initiator by taking the minimum of the accumulated minimum clock values and the accumulated minimum timestamp values and broadcast to the rest of the processes. This procedure is illustrated in the psuedocode in algorithm 9.

Algorithm 9: Mattern Control Token Receive Procedure: Initiator Node

```
if mcount = 0 then
    gvtApprox ← min(mclock, msend)
    SendGVTUpdate()
    if color_initial = WHITE then
        color_initial ← RED
    else
        color_initial ← WHITE
    clock_min ← ∞
else
    ts_min ← min(ts_min, msend)
    msgcount ← msgcount_initial + mcount
    msgcount_initial ← 0
    lvt_min ← doFujimoto
    SendToken(i, (i + 1) mod N, lvt_min, ts_min, msgcount)
```

8.3 Termination Detection

Termination detection, like GVT, is a problem of determining the global state of the system. Termination is a *stable property* of a distributed system, meaning that once termination conditions occur, the system will remain with termination conditions forever until further action is taken.

A process in the system can be in one of two states at any time: active or passive. A process is considered active if some basic computation still remains and passive otherwise. When all processes in the system become passive and no messages are left in transit, then the system should be terminated. The purpose of the termination detection algorithm is to determine when this occurs. A passive process can become active with the arrival of an *activation message*. For parallel discrete event simulation the basic computation is event processing and the activation messages are events. A termination detection for any parallel discrete event simulation must satisfy the following properties:

Safety: Termination will not be detected if any unprocessed event is still present in system including all

local pending event sets and events still in transit.

Liveness: Termination will be detected at some finite amount of time after all events have been processed.

Just like GVT algorithms, termination detection algorithms can implemented with message passing or shared memory. Termination algorithms vary widely because different systems can define termination conditions in such different ways. Furthermore, termination usually does not affect system performance so correctness is more important than optimization. For these reasons only the termination algorithm that is implemented in WARPED2 will be discussed any further.

8.3.1 Termination in WARPED2

The termination detection algorithm in WARPED2 is actually two independant algorithms, a message passing algorithm and a shared memory algorithm. In the opposite manner as the GVT algorithm, the shared memory algorithm is actually used to initiate the message passing algorithm.

Shared Memory algorithm

The purpose of the shared memory algorithm is to determine when all of the worker threads in a process become passive (inactive). The algorithm is fairly straightforward and uses just a single shared counter variable to keep track of the number of active worker threads and a boolean array to keep track of which worker threads are active. Worker threads can only read or write to their own status values. The counter is updated atomically to avoid possible race condtions.

The safety property is achieved because there can be no transient messages. If a thread sends an event to an LP that will be processed by another thread and then becomes inactive, false termination cannot detected because the receiving thread becomes active again after seeing the event. Come back to this, it may not be accurate. The liveness property is achieved because the manager thread periodically checks the active worker thread count so the inactivity of all worker threads will eventually be discovered. If the count is zero, then the message passing algorithm will be initiated between processes.

Message Passing Algorithm

The algorithm that is carried out among processes is an asynchronous message passing algorithm based on Mattern's "sticky flags" algorithm [18]. Just like the GVT algorithm, the token is passed in a logical ring to increasing process ids. However, the initiator process of the algorithm is dynamic. The initiator is the first active process that the token reaches during circulation. For the first circulation, the initator is process 0.

Each process has two states, one for the actual state and one for the *sticky state*. The sticky state is the state that is actually used in the algorithm. It becomes active when the actual state becomes active but sticks to active when the actual state becomes passive. The sticky state can only changes to passive on the arrival of a token. By using this scheme the token is forced to circulate two times with no process becoming active and ensures the safety property. Without this scheme, a process that has already forwarded the token because it was passive could receive an activation message and then the sender of the activation message could become passive before receiving the token. False termination could then be detected.

The process variable names that are used for the succeeding discussion are listed in algorithm 10.

Algorithm 10: Process Variables in Termination Detection Algorithm

$\boldsymbol{state_{actual}}$ **:** Indicates the actual state of the process at all times
$\boldsymbol{state_{sticky}}$ **:** Indicates the state of the process if active, but may stick to active if passive
$\boldsymbol{initiator}$ **:** Boolean variable indicating process is leader

The termination token that is sent between process has the format:

$< sender, receiver, mstate, minitiator >$

where $sender$ is the id of the sending process, $receiver$ is the id of the receiving process, $mstate$ is the partial state of system based on visited processes, and $minitiator$ is the process id of the initiator.

The algorithm is started when the initiator process becomes passive which is determined by the shared memory algorithm. When a process receives the token it becomes the initiator but loses its roles as initiator when it sends the token. That way, only a single process can be the initiator. Furthermoe, since the token always stops at an active process, the token is always guarenteed to start again when it becomes passive which also guarentees the liveness property. When the token is received back at the initiator with a passive state, then termination is signaled to all processes. The procedure is illustrated in algorithm 11.

It should also be noted that this algorithm is only guarenteed to work if the message order is preserved.

Algorithm 11: Termination Token Receive Procedure

$initiator \leftarrow true$
if $state_{sticky} = PASSIVE$ **and** $state_{process} = minitiator$ **then**
    if $mstate = PASSIVE$ **then**
        Signal termination
else if $state_{sticky} = PASSIVE$ **then**
    $initiator \leftarrow false$
    `SendToken` $(mstate, minitiator)$
$state_{sticky} \leftarrow state_{actual}$

If that is not the case, then message counters are necessary to ensure transient activation messages are not missed.

Chapter 9

Memory Management

This chapter will focus on three main topics that are all related to the management of memory. First, state saving techniques will be discussed as well the analysis of state saving in WARPED2. This will include both space and time requirements. Then fossil collection and its overheads are introduced and a comparison of worker thread versus manager thread fossil collection will be examined. Lastly, memory allocation and deallocation and its importance in time warp is presented as well as how it is accomplished in WARPED2.

9.1 State Saving

Copy-state Saving is the classic state saving technique that saves every past state of every LP and requires that the all states are copied and saved into the state queues. This method requires a lot of extra memory and requires a lot of extra computation to copy the states and insert them into the state queue. For this reason, copy-state saving is usually used in conjuction with other technique or not at all.

Periodic State Saving is another state saving technique where the state of each LP is saved only once every N events, where N is a number greater than one. Periodic state saving can significantly reduce the overhead of the time taken to copy the state of the LPs into the state queue. However, due to state history being lost, more processed event have to be saved so that the state of LPs can be restored correctly. These saved events must be reprocessed to restore the state in a process known as *coast forwarding*. During coast forwarding the events are processed normally but only to update the state. The only difference is that no

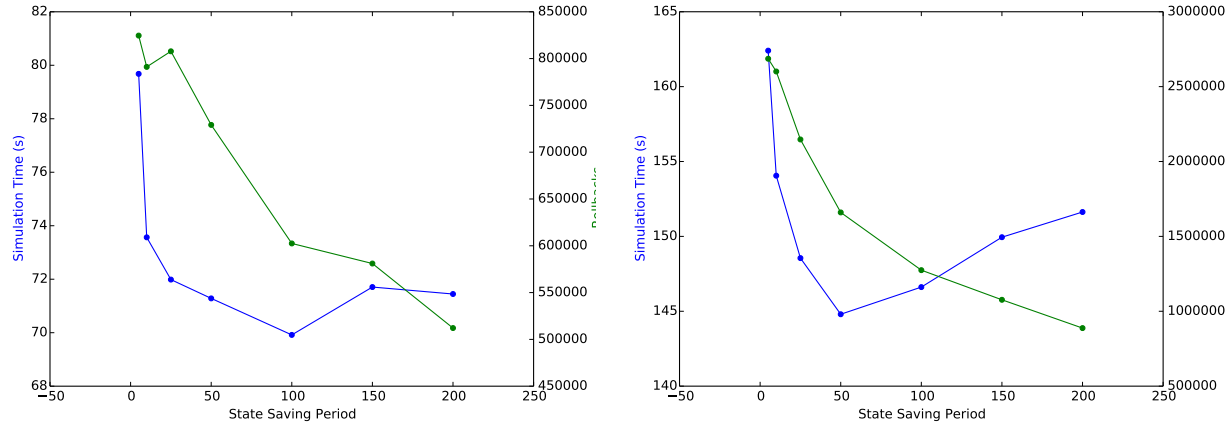
events are sent during the coast forwarding phase. The downside is that the rollback length is increased due to the loss in state saving. However, periodic state saving is a significant improvement over the base copy-state staving approach since the reduced time in state saving far outweighs the extra rollback time for increased rollback length.

Incremental State Saving is a technique that aims to reduce the amount of memory needed to store past states of the LPs. Instead of saving entire snapshots of the past states, the differences in specific state variables are saved. This method requires that metadata describing which variables are modified for each event. Therefore, this approach works well as long as only a small portion of the state variables change during the forward execution of events.

Reverse Computation is a modern approach to state state saving that does not actually save any past states of the LPs. Instead, the *control state* of the forward computation is saved as a set of bits that describe the control flow. By saving the control flow of the forward execution for each event, the state variables that are modified and how they are modified are known and can be reversed. That is, at least, if all state variables are reversible without the need to know the histories of the variables. Also, a reversible random number generator is required. These are both practical requirements, though. The major problem with this approach is the necessity to understand forward and reverse computation precisely in the implementation of the simulation model.

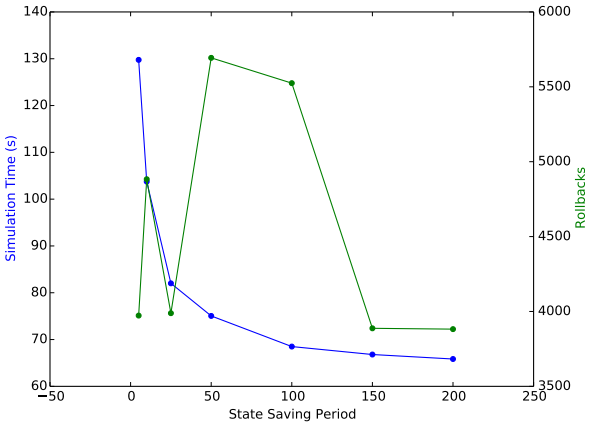
9.1.1 State Saving in WARPED2

WARPED2 currently implements periodic state saving because it can work well regardless of the simulation model and it can be implemented transparently in the simulation kernel.

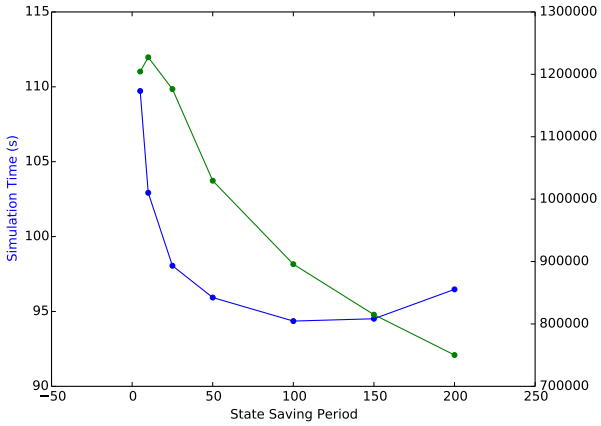


Traffic Model

PCS Model



Epidemic Model



Airport Model

Figure 9.1: State Saving Performances

9.2 Fossil Collection

When the events and states in the processed queues, output queues, and state queues are no longer needed, the memory that they consume is no longer needed and can either be freed or reused. Fossil collection is the process of reclaiming memory for future use. There are many methods of fossil collection that have been used in practice.

GVT-based Fossil Collection is the traditional method of fossil collection. Since rollbacks cannot occur past the GVT, it can be used be used as a marker to indicate what can be fossil collected. In this case, the calculation of the GVT generally triggers the fossil collection process and all memory is explicitly freed so that it can be reused again.

On-the-fly Fossil Collection is a method of fossil collection used in GTW which does not use GVT and memory is not explicitly freed. Instead, after events are processed, they are added to a free memory list. New events can then be allocated from the list as long the first event in the list has a timestamp less than the GVT. If it does not then the current event being processed is either aborted until the GVT reaches a higher value or the list is searched for a lower timestamp event.

Optimistic Fossil Collection is another method of fossil collection which aims to reuse memory instead of explicitly freeing it. However, optimistic fossil collection is based on a statistical bound for rollbacks. It is still possible that memory that was fossil collected could still be used causing an OFC fault. Therefore, methods such as global state checkpointing must also be implemented to recover from these faults.

9.2.1 Fossil Collection in WARPED2

Fossil collection in WARPED2 is currently based on GVT. When the GVT is determined all queues are fossil collected for each LP one at a time.

Although the manager thread determines the GVT, it is undecided whether the worker threads should fossil collect or the manager thread. If the manager thread fossil collects then it can slow down interprocess communication by taking up too much of the manager threads time. Furthermore, this will require extra

synchronization for all of the LPs processed queues, state queues, and output queues since they would no longer be dedicated to a single thread.

9.3 Memory Allocation

Parallel discrete event simulation systems must frequently allocate and free memory for events and states. This can be a huge overhead if used memory and free memory is not maintained in an efficient way. Furthermore, memory must be allocated to deallocated from each process by the operating system as needed which can be even more costly.

In a standard system, the application will never know when this occurs because it is handled by runtime libraries which provide an API for allocation/deallocation for each specific process. For this reason, some time warp systems allocate a large fixed sized memory space before the start of the simulation and use their own memory management schemes.

Dynamic memory management in WARPED2 is achieved through the standard malloc/free interface just like any other C/C++ application and does not pre-allocate any memory. Custom memory management is usually not necessary because modern implementations of malloc/free usually manage memory efficiently. Furthermore, the default allocator can be overriden by other allocators such as tcmalloc, jemalloc, hoard, etc.

9.3.1 Thread-Caching Malloc (TCMalloc)

TCMalloc is a memory allocator that is designed to efficiently allocate memory in multi-threaded applications such as WARPED2. In TCMalloc, seperate thread-local caches of free memory are maintained to avoid contention that would be caused by shared caches.

Chapter 10

Observations with the ARM big.LITTLE Platform

Chapter 11

Conclusions and Suggestions for Future Research

11.1 Summary of Findings

11.2 Detailed Conclusions

11.3 Suggestions for Future Work

Bibliography

[1] R. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, pp. 30–53, Oct. 1990.

[2] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 405–425, July 1985.

[3] M. P. Forum, "Mpi: A message-passing interface standard," tech. rep., Knoxville, TN, USA, 1994.

[4] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: a Time Warp system for shared memory multiprocessors," in *Proceedings of the 1994 Winter Simulation Conference* (J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, eds.), pp. 1332–1339, Dec. 1994.

[5] R. M. Fujimoto and M. Hybinette, "Computing global virtual time in shared-memory multiprocessors," Aug. 1994.

[6] H. Avril and C. Tropper, "Clustered time warp and logic simulation," in *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 112–119, 1995.

[7] C. D. Carothers, D. Bauer, and S. Pearce, "Ross: A high-performance, low memory, modular time warp system," *Journal of Parallel and Distributed Computing*, pp. 53–60, 2000.

[8] A. Pellegrini, R. Vitali, and F. Quaglia, "The rome optimistic simulator: Core internals and programming model," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, (ICST, Brussels, Belgium, Belgium), pp. 96–98, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011.

[9] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, "Optimization of parallel discrete event simulator for multi-core systems," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 520–531, May 2012.

[10] Y.-B. Lin and P. Fishwick, "Asynchronous parallel discrete event simulation," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 26, pp. 397–412, Jul 1996.

[11] K. S. Perumalla and S. K. Seal, "Discrete event modeling and massively parallel execution of epidemic outbreak phenomena," *Simulation*, vol. 88, pp. 768–783, July 2012.

[12] C. Barrett, K. Bisset, S. Eubank, X. Feng, and M. Marathe, "Episimdemics: An efficient algorithm for simulating the spread of infectious disease over large realistic social networks," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–12, Nov 2008.

[13] B. D. Lubachevsky *et al.*, "Rollback sometimes works...if filtered," in *Winter Simulation Conference*, pp. 630–639, Society for Computer Simulation, Dec. 1989.

[14] R. Rönngren and M. Liljenstam, "On event ordering in parallel discrete event simulation," in *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation*, PADS '99, (Washington, DC, USA), pp. 38–45, IEEE Computer Society, 1999.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of ACM*, vol. 21, pp. 558–565, July 1978.

[16] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, Feb. 1985.

[17] T. Lai and J. Yang, "On distributed snapshots," *Information Processing Letters*, vol. 25, pp. 153–158, May 1987.

[18] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 423–434, Aug. 1993.

[19] B. Samadi, *Distributed Simulation, Algorithms and Performance Analysis*. PhD thesis, Computer Science Department, University of California, Los Angeles, CA, 1985.

[20] D. Bauer, G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman, "Seven-oclock: A new distributed gvt algorithm using network atomic operations," in *In Proceedings of the Workshop on Parallel and Distributed Simulation (PADS) 05*, pp. 39–48, IEEE Computer Society, 2005.

[21] A. O. Holder and C. Carothers, "Analysis of time warp on a 32,768 processor ibm blue gene/l supercomputer," in *Proceedings of the 2008 European Modeling and Simulation Symposium (EMSS '08)*, 2008.