University of Cincinnati

Date: 7/19/2013

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

It is entitled:

Event List Organization and Management on the Nodes of a Many-Core Beowulf Cluster

Student's name: **Thomas J Dickman**

This work and its defense approved by:

Committee chair: Philip Wilsey, Ph.D.

Committee member: Fred Annexstein, Ph.D.

Committee member: Fred Beyette, Ph.D.

6778

Event List Organization and Management on the Nodes of a Many-Core Beowulf Cluster

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 Electronic and Computing Systems
of the College of Engineering and Applied Sciences

July 19, 2013

by

Thomas J. Dickman
BSEE, University of Cincinnati, 2013

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

Abstract

Parallel Discrete Event Simulation (PDES) has been widely studied for many years. One of the principle mechanisms for the distributed synchronization of concurrent event execution in a PDES is an optimistic method called the *Time Warp* Mechanism. Researchers at the University of Cincinnati have been studying the Time Warp mechanism for many years and they have developed an implementation of the Time Warp mechanism for execution on Beowulf clusters that is called WARPED. In recent years, the WARPED kernel has been modified to add integrated thread support for execution on multi-core and many-core Beowulf clusters. While this threaded WARPED kernel provides effective execution support for Beowulf clusters containing small multi-core processors, contention for the shared input event queue (or *pending event set*) on a node has a negative impact on the overall performance of the simulation once the number of threads in the nodes grows above 6-8.

This thesis explores the design of the software architecture for the pending event set in the WARPED kernel. In particular, we examine a solution where the pending event set on each node is partitioned into multiple input queues (called LTSF queues) with disjoint subsets of the event processing threads bound to each LTSF queue to help reduce contention. While this reduces contention for the shared pending event set to managable levels, it can potentially lead to an unbalanced advancement of simulation time among the subsets of event processing threads. Thus, this thesis also examines the implementation of methods to dynamically adjust the partitioning of events distributed to the various LTSF queues on a node. Several algorithms and metrics are used to decide when and where to adjust the workload with improved run time results occurring as a result.

Acknowledgments

I would like to thank my parents for all of their support and encouragement as I completed all of my years of schooling. I would also like to thank Dr. Philip A. Wilsey for all of his advice and assistance along the way. I appreciate and thank Sounak and Xinyu for all of their encouragement and help along the way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many fields, including economics, the military, computer science, and engineering employ simulation to model the state of a system over time [1]. These fields use simulation in situations where constructing and testing the actual device would be too costly, or in which testing would be prohibitively complex. There are two common types of simulation models, namely: *continuous* and *discrete event*. Continuous simulation models generally represent systems as a collection of equations that define the state of the system at all (modeled/simulated) time points. In contrast, discrete event simulation models represent a physical system by a series of events and transforms on events that occur at specific time points [1, 2]. This thesis examines discrete event simulation, or DES (*Discrete Event Simulations*). A normal DES operates by processing the lowest time-stamped event one at a time, creating new events and updating states as each event is processed. For large simulations, this can lead to exceedingly long simulation run times with huge memory requirements [3]. As a result, the DES community has turned to parallelism to help attack the speed and memory needs of large simulation models. Parallel DES has become known as PDES (Parallel Discrete Event Simulation) and there are several implementation strategies for coordinating the parallel execution of events within the simulation model. Of particular importance among these strategies are the distributed synchronization of event execution. In a distributed synchronization mechanism, the determination of event execution (also called event scheduling) is determined in a distributed manner across the processing nodes of the PDES. Solutions for distributed synchronization are generally subdivided into two categories, namely: *conservative* and *optimistic* [1, 4–6]. This work examines the organization and management of

event scheduling in an optimistically synchronized PDES simulation kernel.

The most widely used optimistic synchronization mechanism for PDES is called the *Time Warp* mechanism [1,5,7]. The Time Warp mechanism is optimistic in that it does not always enforce strict causal orders in the scheduling of events for execution. Instead it aggressively schedules events and maintains some roll-back mechanism to recover should an event be prematurely executed. While the Time Warp mechanism has been extensively studied in the research community [1], these studies have been primarily on either shared memory multi-processors [8] or on distributed memory systems (both on parallel hardware and on Beowulf clusters) [9–12]. Studies on multi-core and many-core Beowulf clusters are just beginning to emerge and they are uncovering some challenges in achieving substantial performance improvements [13–15]. In particular, the software architecture and data structures for managing the pending event set can quickly become a bottleneck as the number of threads grows (triggered by the increasing number of cores in a many-core processor) [14]. For example, scheduling events to follow the lowest-timestamp favors a unified pending event queue; however, lock contention to the unified event queue favors a design with multiple pending event queues from which the event processing threads can access events ready for execution.

Research in PDES at the University of Cincinnati has produced a PDES simulation kernel called WARPED. WARPED was originally developed to support studies of Time Warp PDES on Beowulf clusters [12]. More recently, it has been extended to support threaded execution within the nodes of a multi-/many-core Beowulf cluster [14]. In the initial implementation of threaded WARPED, the immediately available events from the pending event are sorted into an LTSF (Least-Time-Stamp-First) queue that is a shared (lockable) data structure used by the threads to execute events following the Time Warp synchronization paradigm. While this solution works well for small multi-core Beowulf clusters, once the number of threads exceed 6-8, contention for the shared LTSF queue begins to negatively impact run time performance. This thesis studies modifications to the software architecture for managing the pending event set in the WARPED simulation kernel. The principle objective of this study is to reduce contention to the pending event set while maintaining effective event scheduling among all of the worker threads within a many-core processor.

1.1 Motivation and Principle Hypothesis

In recent years, modern microprocessors have been moving from multi-core to many-core. For example, existing multi-chip motherboard configurations already support up to 64 threads and this number is growing. Efficiently supporting fine-grained parallel applications on these hardware platforms can easily encounter performance impacting lock contention when accessing shared data structures. Within the context of PDES, the pending event set is a critical resource that must be shared by the event execution threads.

The goal of this research is to modify the threaded WARPED kernel LTSF queue structure to better utilize the resources on many-core nodes of a Beowulf cluster. The principal hypothesis is that *optimizing the LTSF queue structure and properly balancing the separate queues, makes it possible to increase the performance of the* WARPED *Time Warp simulation kernel*.

This thesis explores the use of a multi-LTSF queue strategy in order to supply threads with events for execution. It also investigates load balancing algorithms for maintaining balanced execution speed across all threads.

1.2 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 provides an introduction into some common terminology within the field. This includes DES, PDES, and the Time Warp Mechanism.

Chapter 3 describes load balancing, and common uses for it. This includes different types of load balancers, the steps used for load balancing, and some common load balancing metrics.

Chapter 4 provides an overview of WARPED, and the data structures used in Threaded Warped.

Chapter 5 presents the modifications made to the Threaded WARPED kernel to support multiple LTSF queues, and load balancing between LTSF queues.

Chapter 6 details several experiments that are performed to explore the performance impact of multiple LTSF queues and load balancing. This section includes an explanation of the experiments run, results, and an explanation of results.

Finally, Chapter 7 explains insights gained from this research and future research work that can be

explored on the WARPED simulation project.

Chapter 2

Background

2.1 Introduction

This chapter provides a brief introduction into Discrete and Parallel Event Simulation and some of the common terminology used within the field. It also provides an overview of the Time Warp Mechanism [7] that is used in the WARPED simulator implementation.

2.2 Discrete Event Simulations

A Discrete Event Simulation or DES is used to model a system by keeping track of the state changes each piece of the simulation advances through the discrete points in time in the simulation model [16]. These discrete points in time are normally represented by time-stamps stored in the data structure with the event. A traditional, event driven DES contains the following items [17]:

Event list: Contains future events and an associated time for execution. Events are retrieved from the event list for execution.

Clock: Contains the time of the next event to be processed. The clock is used to track the overall simulation time.

State: The current state.

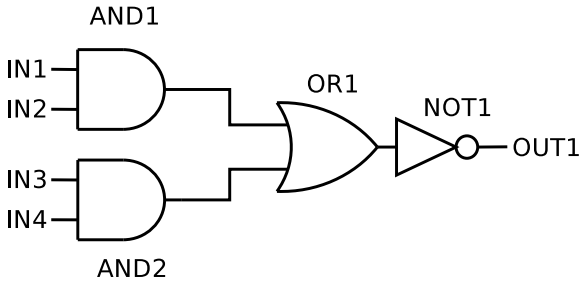

Figure 2.1: Circuit Model

Physical processes are typically used to represent the individual pieces of system being simulated by a DES. Each physical process is represented in the simulation by a Logical Process (LP) [17]. Each LP has a state variable that stores the local state of the process and a clock that stores the local simulation time of the process. A common example given for a DES is a simulation of a logic circuit, such as the one shown in Figure 2.1. An LP represents each logic gate in the simulation. The inputs and outputs to each discrete logic gate represent the state of the LP. The simulation events represent the signals changing value and are generated and sent to the next logic gate. For example, AND1 uses IN1 and IN2 to generate an output event that is sent to OR1. A DES typically goes through the following procedure to process events:

1. Retrieve the lowest timestamped event from the event list.
2. Simulate the fetched event (the simulation model defines this process).
3. Update the object state, and insert any generated events into the event list.

2.3 Parallel Discrete Event Simulations

As the size of the system being simulated increases, long simulation run times become increasingly problematic. One potential way to decrease the simulation time is by using a Parallel Discrete Event Simulator (PDES). A PDES runs on multiple cores, or on a group of computers that are connected together. Instead of executing one event at a time, simultaneous execution of events from separate LPs occurs, and messages sent between separate instances facilitate communication between LPs. Execution of events from each LP occurs in a non-decreasing order [17, 18] so local causality [19] is guaranteed not to be broken. All events in an LP are not necessarily causally dependent, so while this will guarantee proper execution, it is not always

necessary.

Unfortunately, with a Parallel Discrete Event Simulation, causality errors can occur. When events are not executed in a strict non-decreasing time-stamp order, and the events are causally dependent, a causality error occurs. One instance in which this might occur is in a simulation with two LPs running on two separate machines, with each executing events from one LP. If the first machine is executing an event from the first LP, it is possible that the second machine could execute an event that causes an event to be inserted into the second LP. If this event has a lower timestamp than the event being executed, the simulator will not execute the event in the proper increasing timestamp order, and therefore cause a causality error. There are several approaches to solve this problem in PDES, and two of the most common approaches for the distributed synchronization of PDES are *conservative* [20] and *optimistic* [7].

Conservative approaches avoid the possibility of a causality error occurring by defining a method to determine when it is safe to execute an event. For example, if an LP contains an event with a timestamp T0, and it can determine that no other LPs can generate events with timestamps less than T0, it can safely process the event as it is guaranteed that a causality error will not occur later [1]. Unfortunately, if two events are waiting on each other, this can lead to a deadlock situation, so some deadlock avoidance mechanism must be utilized.

Optimistic approaches do not strictly enforce causal relations to events. Thus, optimistic methods introduce the possibility of a causal violation that must somehow be addressed. That is, if a causality error occurs, the system utilizes a strategy of reverting the simulation state to fix the error. This approach makes the assumption that optimistically executing events and fixing any causality errors that may occur is faster than preventing the issue in the first place. One popular implementation of this strategy is known as the Time Warp mechanism [8, 10, 21], but various other algorithms exist [22].

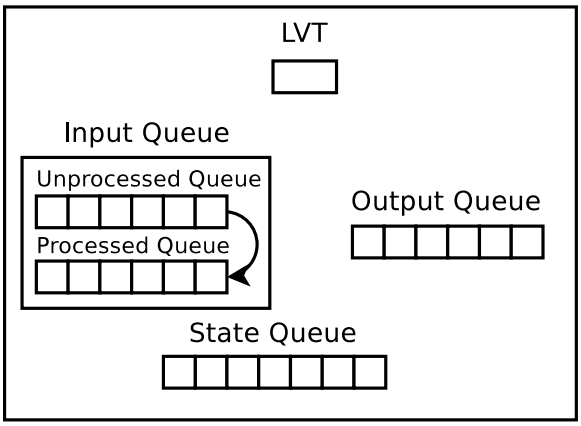

Figure 2.2: Logical Process Model

2.4 Time Warp Mechanism

The Time Warp mechanism executes and processes events for each LP as a separate DES. Figure 2.2 illustrates the data structures used for each LP. The input queue contains events that are waiting to be processed by the system and events that have already been processed. The output queue contains anti-messages, which are sent to other LPs to cancel events. The state queue stores previous system states, with varying granularities depending on the setup. Each LP also has its own LVT value, which is the timestamp of the last event executed by that LP. The combined progress of all LPs is known as the the *Global Virtual Time* (or GVT) and can be determined using one of several possible algorithms [23–26].

Each LP processes events from the input queue and generates other events that are passed to other LPs. A *straggler* is any event with a timestamp less than the currently executing event, potentially resulting in a causality error. Time Warp recovers from this situation by issuing a rollback. A rollback causes the simulation to roll back to a state saved from before the timestamp of the straggler event (this is called a *rollback*). This allows the straggler event to be processed in its proper causal order.

Any events that were prematurely sent to other LPs also need to be canceled during a rollback operation. This is accomplished by issuing *anti-messages* for the events in the output queue that were prematurely generated. These messages are either sent immediately for *aggressive cancellation* [7, 27], or delayed until re-processing shows that they are actually incorrect (called *lazy cancellation* [28, 29]). When an LP receives an anti-message, it removes the event from its input queue. If processing of the event has already occurred, that LP must also rollback, which can potentially lead to a state of cascading rollbacks [5].

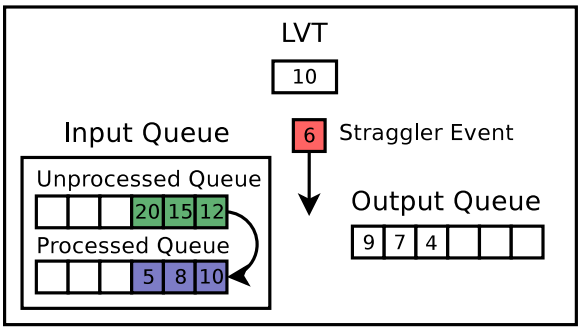

Figure 2.3: Rollback Process — Straggler Received

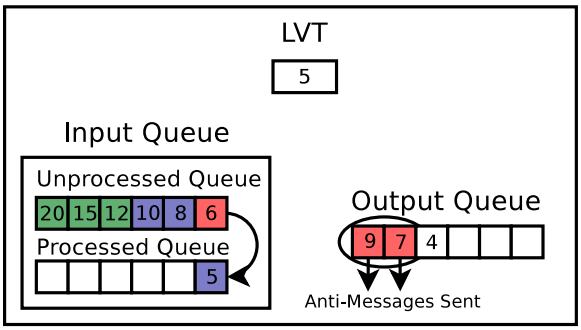

Figure 2.4: Rollback Process — Rollback Processed

Figure 2.3 demonstrates the process that occurs when an LP receives a straggler event (an event with a timestamp less than the LVT). All events with timestamps greater than the straggler event are moved from the processed queue to the unprocessed queue, and the straggler event is placed in the unprocessed queue as shown in Figure 2.4. Next, anti-messages are sent for any events in the output queue with timestamps greater than the straggler event. These anti-messages cancel events previously transmitted to other LPs.

Chapter 3

Related Work

This chapter describes load balancing and its common uses. It reviews the different types of load balancing and the steps normally used for load balancing a system. Finally, it provides a brief description of some common metrics for load balancing a Parallel Discrete Event Simulation.

Load balancing spreads processing load for a specific application across separate systems in order to obtain an overall performance increase [30]. It is used in a variety of systems, from web servers to simulators. The goal of any load balancing system is to distribute resources in an optimal and efficient manner in order to reduce the time needed to perform some action. For a discrete event simulator, the goal is to reduce the overall simulation time, while the goal for a web server is to reduce the response time and increase the number of requests that the server can handle per second (req/s). There are several different ways to perform load balancing. The proper load balancing approach depends on how the load of the system being balanced changes over time.

Two commonly used types of load balancing are *static* and *dynamic* load balancing. Static load balancing assigns resources beforehand, while dynamic load balancing reassigns machine resources during operation. A common example of static load balancing is the splitting of resources for a webserver. A webserver is normally composed of several different processes, including a database process and a webserver process. By assigning these processes to separate machines, the load can be distributed such that multiple machines are sharing the processing load, but this only divides the load between two separate machines. By creating multiple web server frontend machines, and randomly directing incoming traffic, the load can

be distributed even more. As the load becomes more distributed, there is a greater likelihood of the load becoming unbalanced, where a dynamic load balancing system would enhance overall performance. This thesis will explore both types of load balancing, but will primarily focus on dynamic load balancing due to the temporal variation in event arrival rate between LPs as explained later in this thesis.

Dynamic load balancing uses a master process that calculates some sort of metric to decide when to perform load balancing and what action to perform during load balancing. This process typically involves four basic steps:

1. **Calculate Metric:** Measure the load of the different parts of the system.
2. **Initiate Load Balancing:** Determine if the metric exceeds a threshold or other parameter; continue if it does, otherwise exit.
3. **Choose Items:** Determine which items should be moved or reassigned using the calculated metrics.
4. **Perform Migration (reassignment):** Migrate the items chosen in Step 3 to balance the system load.

These steps are typically performed either at a regular frequency by a manager process, or at a rate that slows down as the metric stabilizes. The simplest method is to calculate this metric at a regular interval, such as every few seconds, and reassign load as needed. If the load stabilizes after some time, it makes more sense to perform this process less often if fewer reassignments are occurring. For example, the load balancing calculation rate could be cut in half every time the process does not perform any reassignments.

3.1 PDES Load Balancing Algorithms

Load balancing metrics differ depending on the specific needs of the target application. This section will discuss several load balancing algorithms that are commonly used for balancing the load between the LPs of a Parallel Discrete Event Simulation. Each metric is discussed in terms of its applicability to the WARPED simulator.

Each algorithm involves three distinct steps. The *load parameter* describes the metric that is calculated for load balancing. This is typically calculated for each LP, or for groups of LPs. *Migration* describes LP

reassignment in order to balance load after the calculation of the metric. *Frequency* describes the incidence of this algorithm's calculations.

3.1.1 Tropper's Algorithm

Avril and Tropper designed Tropper's algorithm for Clustered Time Warp (CTW) [9]. It is ideal for simulations of fine granularity (short event execution time), such as logic simulations. The event execution time of different events should be relatively similar. The load balancing in CTW operates on "Clusters (or collections) of LPs" so the granularity of migration will be much coarser than is possible in WARPED (which operates on LPs). However, it is instructive to review the basic concepts in Tropper's load balancing algorithm.

The basic operating steps in Tropper's algorithm are:

Load Parameter: A measure of the number of events processed by the simulator. This includes events that are processed and then later rolled back. Portions of the simulation with a large number of rollbacks will have a reduced number of events processed due to the high number of rollbacks.

Migration: Clusters are migrated from the section with the highest load to the section with the lowest load if the load parameter exceeds some threshold. LPs that increase the interprocessor communication are not migrated in order to negate the effects of increased cost of communication between processors.

Frequency: The calculation of the load parameter occurs at some regular interval that is set before the simulation begins. Load balancing does not occur again until the throughput due to reassignment offsets the initial cost for the load balancing and reassignment.

This algorithm assumes that rollback time is similar to process execution time. This may be true for logic simulation and simulations of fine granularity, but it certainly is not true for all simulations. Clusters are also used for this system, both for load parameter calculation and for migration. This algorithm would require extensive modifications since the WARPED simulation system does not operate on Clusters, but instead operates on individual LPs.

3.1.2 Das's Algorithm

Das and Sarkar designed Das's algorithm [31] to balance the simulation clocks of the separate parts of a simulation. Balancing the simulation clocks prevents cascading rollbacks that can occur if a simulation clock is far behind the others. This algorithm operates as follows:

Load Parameter: This metric is a measure of the system time used to advance the simulation clock forward one time unit. It is referred to as the *Rate of Progress* (ROP) of the simulation, and is calculated for each individual LP. This metric is typically calculated as:

$$\frac{LVTtimechange}{realtimechange} \tag{3.1}$$

The *LVT time change* is the amount of simulation time that passes for each LP in the simulation. The *real time change* is a measure of the actual, real world (wall clock) time that passes between measurements.

Migration: A combination of the ROP values for each individual LP are used to calculate the mean ROP of the simulation. If the maximum deviation exceeds some preset threshold, load balancing occurs. A migration of LPs from low ROP to high ROP occurs. Object selection is performed by selected a random LP, and moving it if it is expected to improve the load balance of the system.

Frequency: The load parameter is calculated for each individual LP at a fixed frequency. This value is typically calculated empirically by performing several simulations, and determining the ideal value.

The metric for this algorithm is relatively easy to calculate, and provides a simple way to compare the process of the different LPs in a simulation. Unfortunately the WARPED simulation system does not provide an easy method for determining whether or not moving an LP is expected to improve the load balance of the system without actually moving the object and observing what happens. The ROP metric is definitely interesting, and will be considered in our study.

3.1.3 Distributed Control Algorithm

Kannikeswaran designed the Distributed Control Algorithm for the WARPED simulation system [32]. Distributed load balancing algorithms allow any simulation object to initiate the load balancing procedure. This reduces the calculations that need to be performed by a central processor, and should ideally improve the load balancer performance. This algorithm operates as follows:

Load Parameter: This metric is a measure of the number of events committed, and is calculated for each individual LP.

Migration: The LP is chosen that communicates the most with other LPs on separate nodes. This LP is moved to the node that it communicates with the most in order to reduce costly interprocess communication.

Frequency: The load parameter is calculated for each individual LP at a regular, preset frequency, such as every few seconds. The frequency method from Tropper's algorithm can also be used to prevent wasted processing time.

This algorithm is ideal for situations where external communication between nodes in a simulation is costly. In our situation, all LPs reside on a single node as separate threaded processes, so interprocess communication is low cost. The load parameter is an interesting value to calculate though.

3.1.4 Load Sharing Algorithm

Vitali designed the Load Sharing Algorithm [33]. Multiple kernel instances are initiated with threads evenly divided, and threads are dynamically reassigned between instances to balance load. This algorithm operates as follows:

Load Parameter: This metric is a measure of the number of events that are committed and are not later rolled back for each kernel instance k_i. This metric is typically represented as evr_i, and the average CPU time required to commit an event for each kernel instance is Δi. Combining these forms, the following equation represents the relative CPU requirements of each kernel instance:

$$wevr_i = evr_i \times \Delta i \tag{3.2}$$

Migration: Threads are reassigned to kernel instances based on $wevr_i$ measured during the previous period. CPU power is assigned relative to these values for each kernel instance, allowing each instance to proceed at a similar rate and prevent causality errors, which lead to rollbacks.

Frequency: The load parameter is calculated during each GVT calculation, since the event rate is also gathered during this time. This allows a reduced number of calculations to be performed. Threads are reassigned to kernel instances during every Migration phase, and are not subject to any threshold values due to the low cost of thread reassignment.

The Load Sharing Algorithm by Vitali is designed based on the assumption that reassigning LPs between kernel instances is costly. While this is true when separate kernel instances are used to divide resources, the threaded model used in WARPED has certain design benefits that make it more efficient. A single kernel instance is shared for all threads on each machine, and LP reassignments require simply updating two pointers, making it considerably less costly.

Chapter 4

Threaded WARPED Internal Data Structures

Several versions of the WARPED kernel exist, including a sequential version, a single threaded Time Warp version, and most recently a threaded Time Warp version that supports multiple threads on each node. Each node in the simulation runs an instance of the kernel, and communicates with other nodes via MPI (Message Passing Interface) as illustrated in Figure 4.1. The threaded kernel creates a single *master* thread for common housekeeping functions, including message passing, fossil collection, GVT/termination, and worker thread control. One or more *worker threads* are also created to process and generate events from the LPs in parallel. The single master thread allows the common housekeeping functions to be combined to better utilize newer multi- and many-core Beowulf clusters with less overhead. This chapter details some of the data structures used in the WARPED kernel, and walks through the execution process of the master and worker threads for the threaded WARPED kernel.

Each simulation model used in WARPED is composed of simulation objects (or LPs). Each LP represents some portion of the simulation, such as a single logic gate in a circuit simulation. The threaded WARPED kernel evenly distributes these LPs between the simulation managers. Figure 4.2 illustrates a simulation with ten logical processes and two simulation managers. The LPs are evenly distributed to the simulation managers. In this instance, each simulation manager is assigned five LPs. Any communication between LPs that are assigned to separate simulation managers is facilitated by the simulation managers. In general only one simulation manager operates on a node of the cluster and communication between simulation managers occurs using MPI.

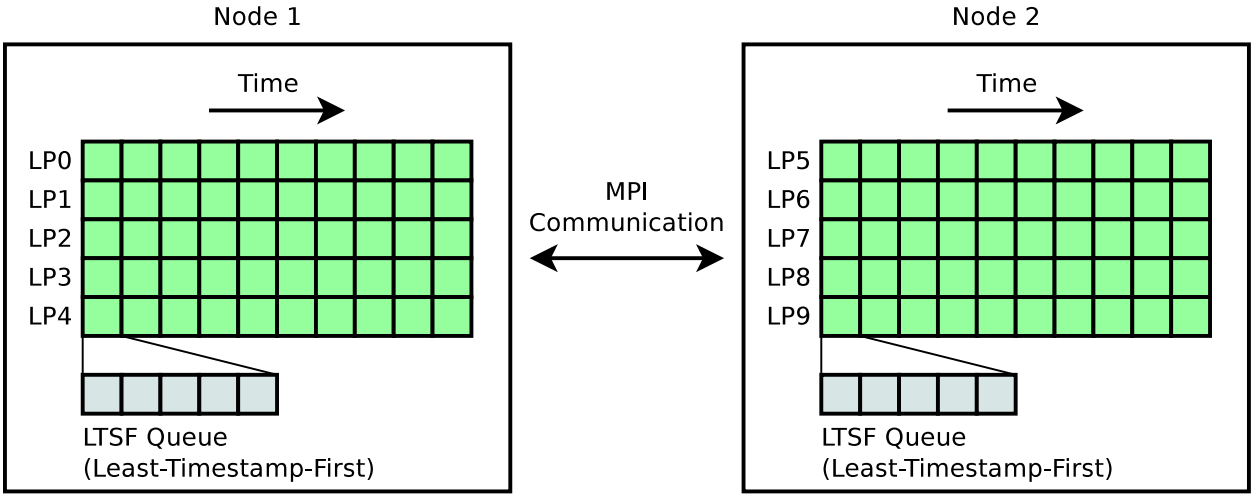

Figure 4.1: Multiple Node Threaded Warped

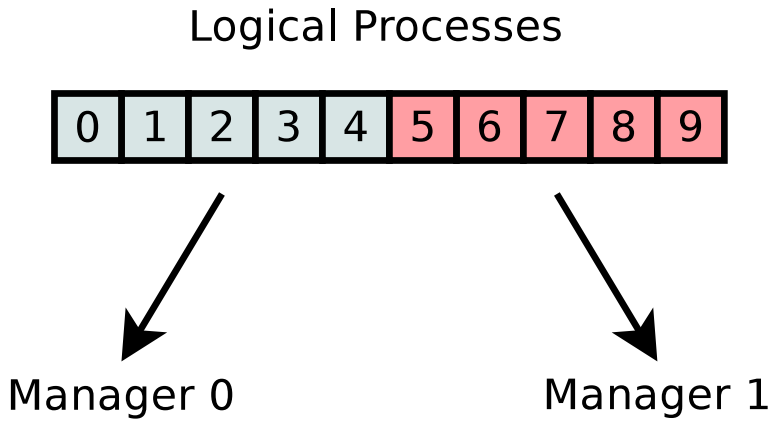

Figure 4.2: Threaded Kernel Simulation Managers

4.1 Data Structures

Several data structures are utilized to store events for execution in the threaded WARPED kernel. The input queues contain events pending for execution by the worker threads. The output queues contain events generated by the worker threads, and are used to properly refill the input queues when rollbacks occur. All data structures are protected by atomic locks, so only a single worker thread can access each at a time. The following two sections show the data structures used for each simulation manager and describe how events are stored and processed in these data structures.

4.1.1 Input Queues

The input queues are used to store events for execution by the worker threads (Figure 4.3). For each LP assigned to the simulation manager, a sorted vector of input events for that LP is defined. This vector is organized into two parts, namely: the *processed* queue and the *unprocessed* queue. The current head input event for the LP corresponds to the head event in the unprocessed queue. Events in the processed queue are used to restore events into the unprocessed queue when rollback occurs and events are removed from the processed queue as GVT advances past. Both of these queues are locked by a mutex lock.

The head input event for each LP is also placed in a sorted *LTSF* (Least-Time-Stamp-First) queue. The LTSF queue is locked and accessed by the worker threads to pull events for execution. Each worker thread removes the event from the head (the smallest timestamped event) of the LTSF queue, processes the event,

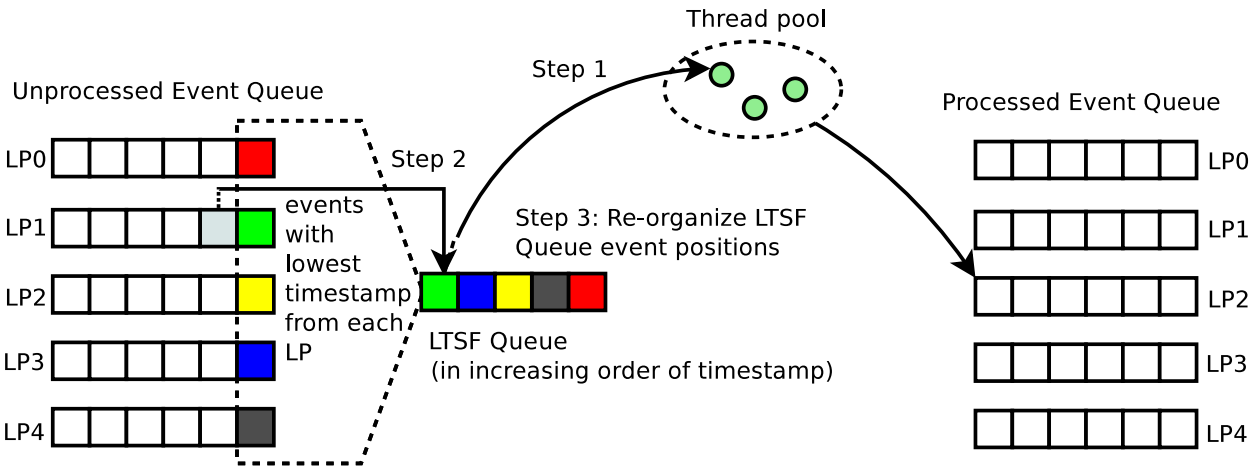

Figure 4.3: Input Queue Data Structures

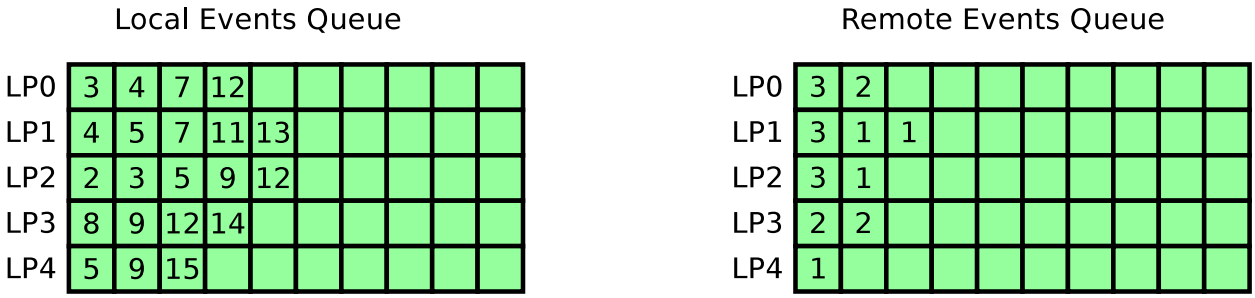

Figure 4.4: Output Queue Data Structures

and then moves the next unprocessed event for that LP into the LTSF queue.

This design for the pending event set works reasonably well provided the number of worker threads is limited to 5-7 [14]. However, once the number of worker threads increases, contention for the LTSF queue lock will often negatively impact performance. This contention problem is the principle motivator for the work in this thesis.

4.1.2 Output Queues

LPs generate events that are sent to both local and remote LPs. These events are stored in the output data structures to accommodate rollbacks. These are composed of a local and a remote events queue for each LP. An LP transmits an event to a local LP by placing it in the local events queue as shown in Figure 4.4. To transmit an event to a remote LP, the event is placed in the remote event queue.

Rollbacks are handled by sending anti-messages, or requests for cancellation for all events with a timestamp greater than the timestamp of the point being rolled back to. For example, observe what happens if a rollback is caused by an event with a timestamp of 7. This causes anti-messages to be sent for all events with timestamps less than 7 shown in Figure 4.4.

4.2 Manager Thread Operation

A single manager thread is assigned to each simulation manager instance and is in charge of all housekeeping functions, including inter-node communication, GVT calculation, termination detection, and thread control. When the simulation begins, the manager thread starts by setting up the simulation. This includes instantiating all of the necessary data structures, including the input and output queues and the atomic locks

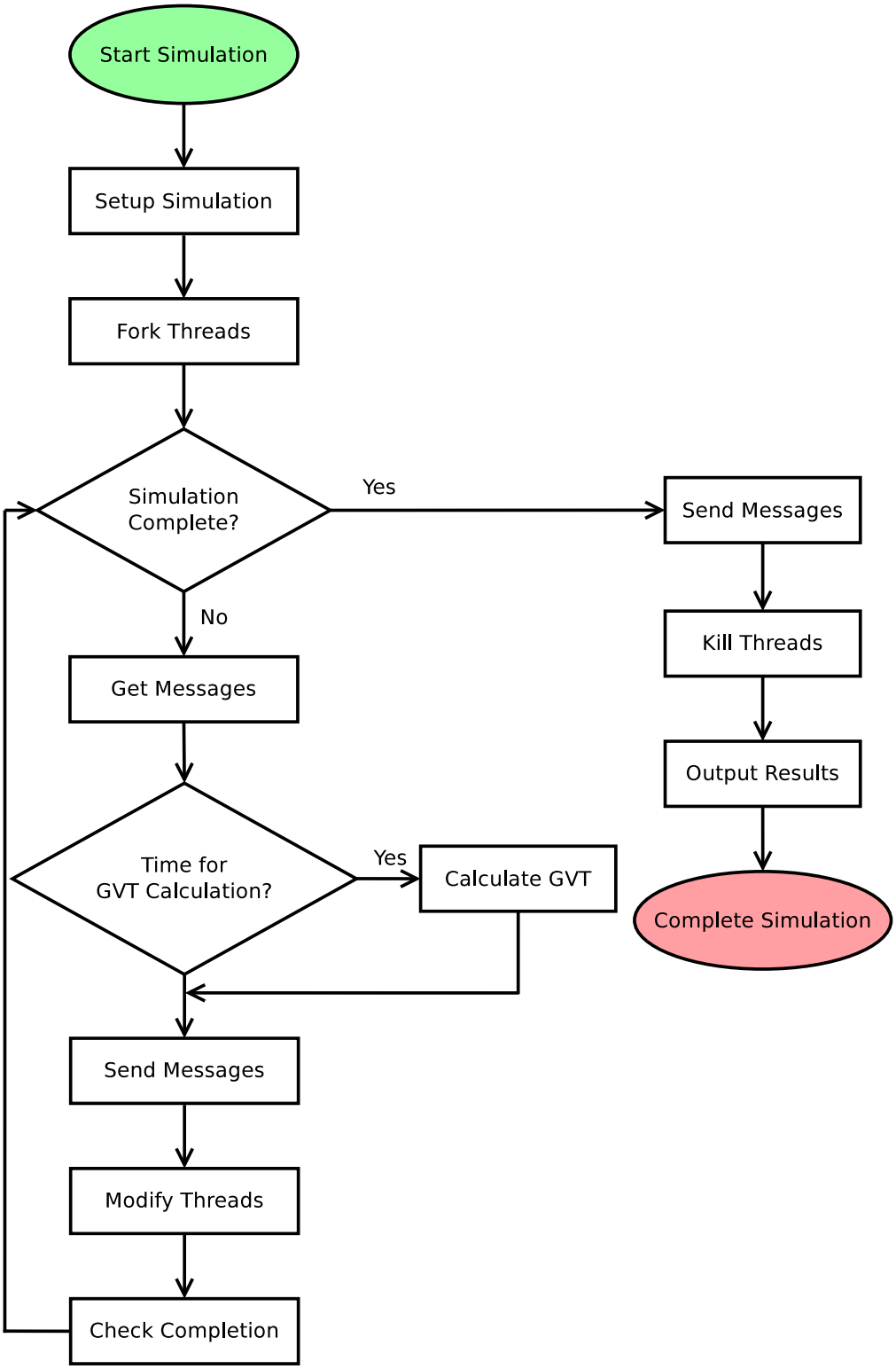

Figure 4.5: Output Queue Data Structures

protecting these data structures. The initial events for all LPs are placed in the appropriate unprocessed queues for execution.

The manager thread then instantiates the appropriate number of worker threads (defined by a runtime configuration parameter). The next section further details the steps the worker threads complete to process events. The manager thread completes the next steps in a continuous loop, until the simulator determines that the simulation is complete.

The manager thread next checks for any messages sent by remote simulation managers. This includes anti-messages for rollbacks, GVT calculation requests, and other messages needed for inter-node communication.

After this, the manager thread determines if it is time for a GVT calculation, and computes the current GVT value if it is. This GVT value is then transmitted to other nodes to complete the GVT calculation.

Any pending messages for external simulation managers are sent next. These include outgoing anti-messages, GVT calculation tokens, incoming events for local LPs, and other necessary communication.

The manager thread controls worker threads by suspending and resuming them to control optimal simulation execution. Worker threads notify the manager thread if they run out of events, and the manager thread puts them to sleep after they empty their input event queues. This allows any threads without work to go to sleep, so the computer resources can be divided among the other worker threads.

Each simulation manager determines if the simulation is complete by checking several criteria. First, they check if all worker threads assigned to that simulation manager are busy. If the all worker threads run out of work, that simulation manager sets its status as passive, and communicates this status with the other simulation nodes. Once all simulation nodes are in the passive state, the simulation is marked as complete, and the simulation is terminated.

The simulation is terminated by sending any pending messages to remote nodes. Then, all threads are rejoined with the master thread. Simulation results and statistics, including simulation time, rollback statistics, anti-message counts, and other pertinent information are output. Finally the master thread terminates and the simulation is complete.

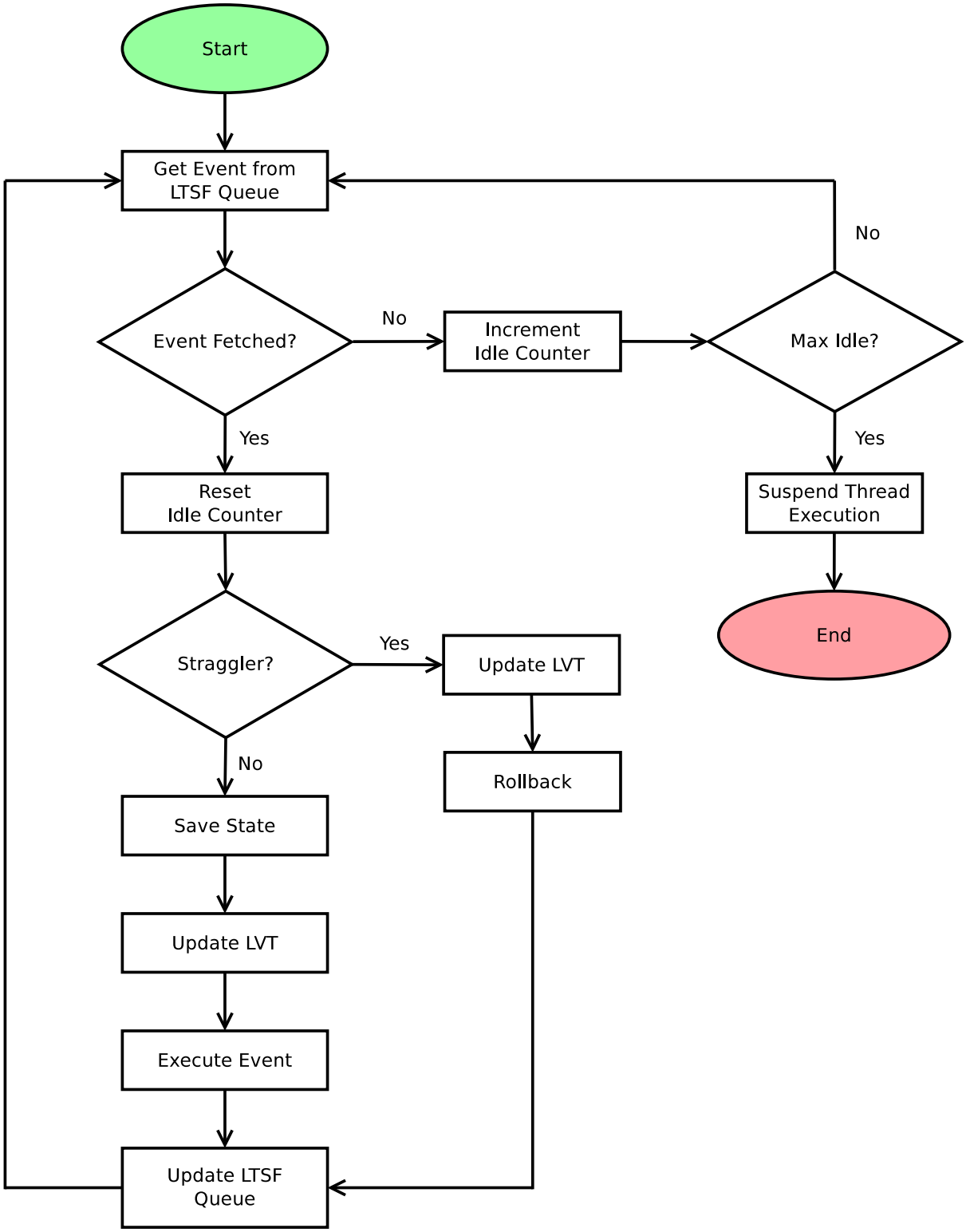

Figure 4.6: Output Queue Data Structures

4.3 Worker Thread Operation

The manager initiates (forks) the worker threads when the simulation begins. Worker threads are used to process events from the individual LPs. Worker threads also handle straggler events, save system state, roll back events, and update the LTSF queue.

A worker thread starts by acquiring the first, lowest time-stamped event from the LTSF queue. The LTSF queue is a priority queue sorted by event time-stamp, so the worker thread is simply able to acquire the first event and be guaranteed to retrieve the lowest time-stamped event.

If the worker thread is unable to fetch an event from the LTSF queue, such as when the LTSF queue is empty, the worker increments an idle counter. When this idle count exceeds some threshold, the thread is no longer doing useful work, and is suspended by the manager thread until more events are detected in the LTSF queue. This prevents the thread from endlessly polling the LTSF queue looking for events to execute. Once the thread successfully acquires an event for execution, it resets this idle count.

When a straggler event is encountered, the Local Virtual Time (LVT) [7] is updated. The LVT value is a measure of the lowest timestamped event for that specific LP, and is used during a rollback. The calculated LVT value is then used to complete the rollback.

If the event is not a straggler, the current system state is saved for future rollbacks. Then, the LVT value is updated as detailed above.

Next, the event is executed. Event execution is handled by the simulation model currently being used. This always results in some sort of output, typically as another event that is inserted into either that event's LP, or an external LP.

Finally, the LTSF queue is updated with the next lowest event from the LP of the event that was just processed. This keeps the LTSF queue as an up-to-date ordered list for worker threads to get the next lowest event for execution.

Worker threads continue this process until they are terminated by the manager thread, or are suspended by the manager thread. Several threads are able to acquire, handle, and execute events simultaneously, fully utilizing the multiple CPU cores in modern machines. All data structures are protected by atomic locks to prevent collision errors.

4.4 Critical Path

The goal of an optimistic PDES simulator is to simulate the events from a process such that the simulation aggressively processes events without strictly enforcing causality constraints. A conservative PDES strictly enforces all causality constraints and therefore will wait to execute an event until all causally dependent events have been processed. For example, if it is possible for event E1 to affect event E2 in any way, E2 is not executed until E1 has been completed. This can lead to the parallelism in a simulation not being fully utilized. In contrast, the optimistic approach assumes that no events are causally related, and executes events as aggressively as possible. If two events end up being processed in violation of their causally dependency, the simulator initiates a process to repair the violation.

The *critical path* of execution is a standard for performance that PDES researchers will commonly use to evaluate their simulators [34]. The critical path is the optimal path a conservative simulation can take so it completes in a minimum amount of time [35]. This execution time is used as a minimum against which conservative PDES kernels are compared. Some implementations of optimistic PDES, such as Time Warped, are potentially able to perform better than critical path execution by using lazy cancellation. Lazy cancellation allows computation results that are correct, or partially correct to be utilized instead of discarded, which can lead to rollbacks.

4.5 Event Execution

This section details specifically how events are fetched and executed in the threaded WARPED kernel with a focus on the LTSF queue. It explains how events are fetched, exactly which data structures are accessed, which locks are necessary, and other pertinent details.

The unprocessed queues contain events pending execution. Each LP has its own queue which contains pending events [36] for execution stored in a priority queue, in time-stamped order. Multiple threads are executing events from these unprocessed queues. In order to stay on the critical path, each thread always chooses the lowest time-stamped event from the unprocessed queues.

Instead of checking each unprocessed queue to find the lowest time-stamped event, the LTSF queue acts as a sorted buffer, so threads can simply remove the first event from the queue for execution, and be

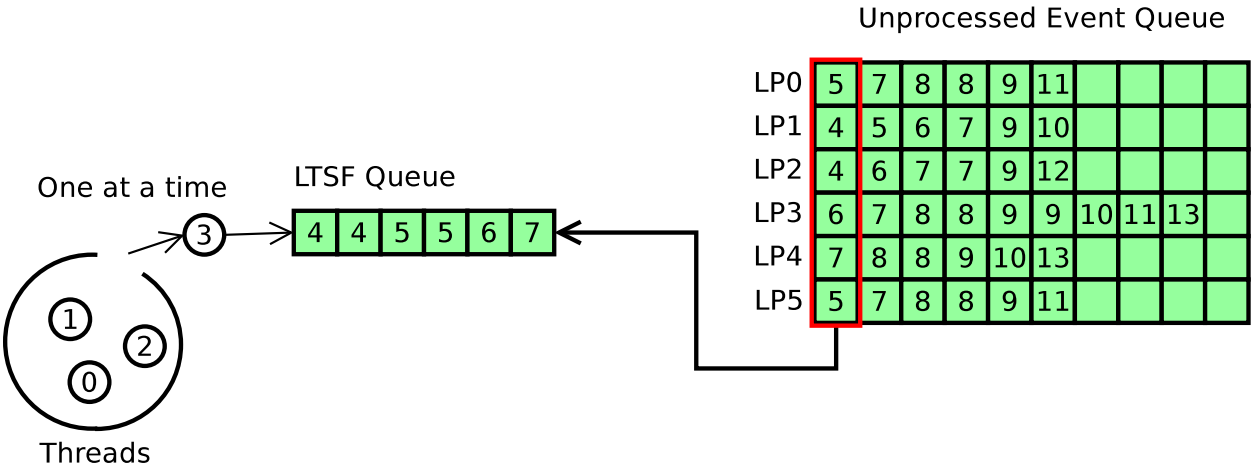

Figure 4.7: Single LTSF Queue Processing

guaranteed to receive the lowest time-stamped event.

Each unprocessed queue is protected by an atomic lock to prevent multiple threads from modifying them, and causing data corruption issues. The LTSF queue is also protected by a single atomic lock. Threads access the LTSF queue during two steps of their execution:

Retrieval: Events are retrieved from the LTSF queue for execution.

Update: Insertion of the next event from the unprocessed queue after execution.

Two other events also occur that require LTSF queue access:

Insertion: The LTSF queue is updated when new events are inserted into the unprocessed queues.

Deletion: Rollbacks cause event reinsertion into the unprocessed queues, causing updating of this.

Figure 4.7 illustrates worker threads accessing the LTSF queue to retrieve events. LTSF queue access is required for all four situations explained above.

4.6 Contention Issues

Each unprocessed queue is protected by a separate atomic lock, and there are typically a large number of unprocessed queues, therefore there is almost no contention when accessing this data structure. In contrast,

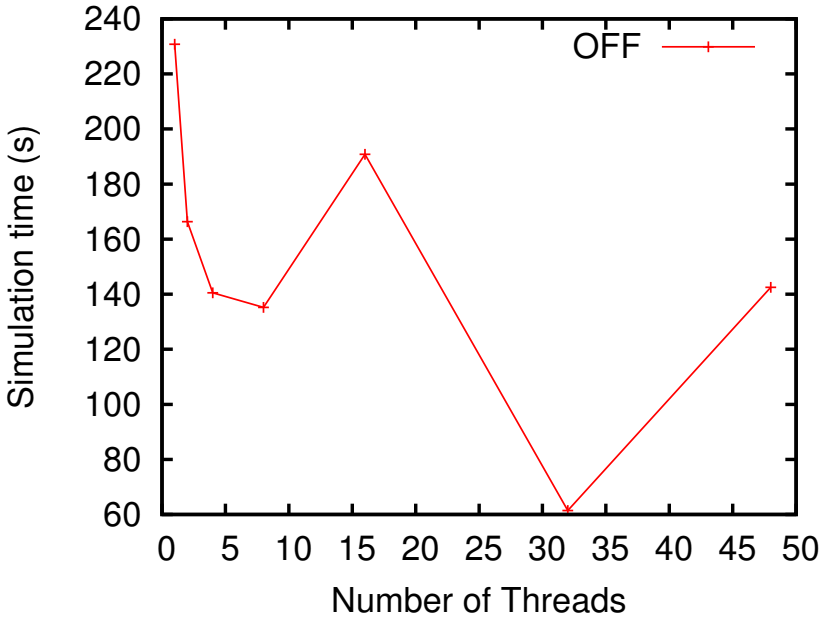

Figure 4.8: Multiple LTSF Queue Processing

the LTSF queue data structure is accessed and updated at least twice during each event execution cycle for each thread, once to get the next event for execution, and once after event execution to update the LTSF queue. The LTSF queue can only be accessed by one thread at a time, as each thread is required to obtain the atomic lock to prevent data corruption issues.

Historically, processors have had a relatively low number of cores, so waiting for other threads to release the LTSF queue was not a significant issue. However, AMD and Intel now have consumer level processors with support for up to 16 simultaneous threads. Some current motherboards support up to four of these chips on a single board, supporting up to 64 threads. Intel is developing a 48 core x86 processor, and a 50-core Kinghts Corner processor [37]. Following these patterns, it is obvious that future consumer processors are going to contain support for even more simultaneous threads.

Profiling of the threaded WARPED kernel as shown in Figure 4.8 shows performance issues and possible contention issues as the number of threads is scaled. Performance becomes flat, and even becomes worse as the number of threads is increased. Every thread is locking and accessing the LTSF queue at least twice during each execution cycle, forcing threads to wait for other threads to release the LTSF queue lock before they can proceed. These threads are wasting execution time waiting on other threads to release the LTSF

queue lock.

4.7 Attacking the LTSF Contention Problem

The goal of this section is to attempt to explore methods to reduce the contention on the LTSF queue data structure, to allow the threaded WARPED kernel to scale to a larger number of threads. Currently performance flattens out as the number of threads increases and the parallelism of recent many-core processor advances is wasted.

One potential solution is to create multiple LTSF queues, and split the threads and LPs among them. In this solution, each LTSF queue is assigned a group of LPs to store events as shown in Figure 4.9. For example, with four threads and eight LPs, two LTSF queues are created, with four LPs and two threads assigned to each. Thus, in this example, half as many threads access each LTSF queue, reducing contention significantly.

Increasing the number of LTSF queues reduces the contention each thread faces accessing the LTSF queue, so optimally it makes sense to create the maximum number of LTSF queues for maximum performance. The number of LTSF queues is limited by the number of threads (one thread assigned to each), but increasing the number of LTSF queues to this limit introduces other issues.

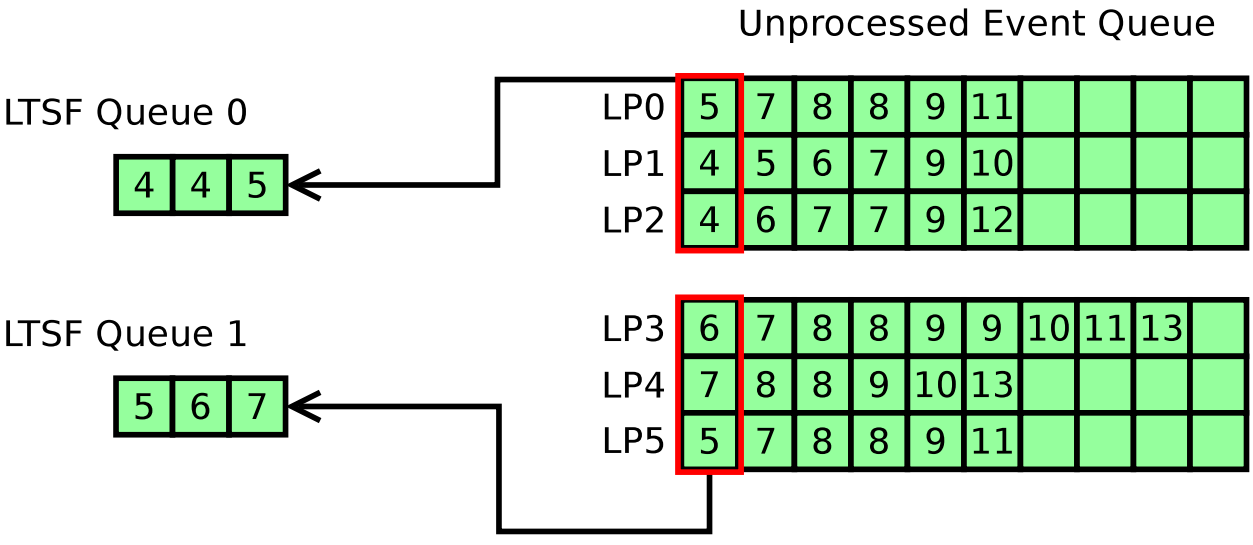

Figure 4.9: Multiple LTSF Queue Processing

4.8 Possible Issues with Contention Solution

LPs are statically assigned to LTSF queues at the beginning of simulation execution. This can result in an unbalanced distribution of load between LTSF queues, leading to reduced simulation performance. One LTSF queue can end up too far ahead or behind in the simulation leading to an increased number of rollbacks occurring due to this inbalance. Rollbacks are events that are essentially executed incorrectly, and have to be re-executed using the proper data. Therefore, any rollback is essentially wasted execution time, and should be minimized.

One potential solution is to dynamically reassign the number of threads assigned to each LTSF queue to try to speed up ones that are running behind, and speed up any that are running ahead. Unfortunately this will lead to increased contention for the slower LTSF queue, reintroducing the original problem multiple LTSF queues seeks to fix.

Another potential solution to this problem is to monitor the load on each LTSF queue, and rebalance the LTSF queues to maintain an even load. Rebalancing the LTSF queues can easily be accomplished by reassigning LPs between LTSF queues. By measuring the load on each LTSF queue and reassigning LPs, the simulation load should be more balanced, and an overall reduction in the number of rollbacks should occur. Ideally, this should increase the system performance.

Chapter 5

Implementation Details

5.1 Introduction

This chapter describes the modifications made to the threaded WARPED kernel to support multiple LTSF queues and load balancing between LTSF queues. It presents the class layout, new configuration settings, and the challenges that were encountered when adding support for a configurable number of LTSF queues. Finally, it details the approach that is used for implementing load balancing between LTSF queues, and the layout of this approach.

5.2 Multiple LTSF Queues

5.2.1 Introduction

The threaded WARPED kernel previously supported a single queue that was used to store the lowest time-stamped event from each input queue. This section describes the modifications made to add support for multiple LTSF queues. The number of LTSF queues is configurable via a global setting in the configuration file by the WARPED kernel. Finally, this section ends with an explanation of the challenges that were encountered during this implementation.

The goal of this research is to explore two approaches for reducing contention when accessing the LTSF queue object. The LTSF queue contains the lowest time-stamped events from the collection of LPs that

construct the simulation.

This LTSF queue contains upcoming events for execution by worker threads. The manager thread forks a defined number of worker threads at the beginning of the simulation. Each thread operates independently, accessing and updating the LPs and LTSF queue by retrieving the next event for execution, and inserting new events to be executed. These threads are used to execute events in parallel that are acquired from the LTSF queue data structure.

The first approach this research addresses is the creation of multiple LTSF queues and the assignment of LPs to a specific LTSF queue. In turn, each worker thread is also assigned to process events from a specific LTSF queue. This reduces the number of threads accessing each data structure, hopefully reducing the contention, and decreasing the simulation time by reducing the amount of execution time wasted on waiting to access a resource.

5.2.2 Class Layout

This section reviews the relevant sections of the threaded WARPED kernel, and details the modifications made to each part to allow the use of multiple LTSF queues. It presents the main data structures within each section.

Threaded Simulation Manager

The Threaded Simulation Manager pictured in Figure 5.1 handles event execution and communication between all separate simulation managers. Each Threaded Simulation Manager is instantiated with a single master thread. The master thread initiates all worker threads before event execution begins.

Master Thread

The master thread instantiates the appropriate number of worker threads using the `createWorkerThreads()` function. The number of worker threads is set before execution using the simulation configuration file. Each worker thread is started using the `startWorkerThread()` function, and begins execution in the `workerThread()` function. The **Worker Thread** section further details the execution process of the worker threads.

The master thread enters the `simulate()` function, which is where the majority of the master thread's

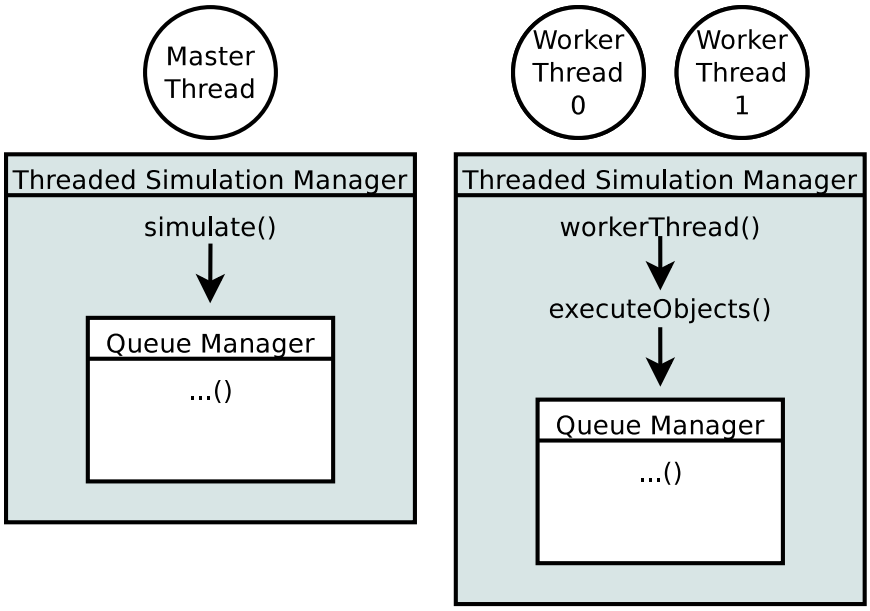

Figure 5.1: Multiple LTSF Queue Processing

work occurs. It executes in an infinite loop until the manager determines the simulation is complete. This function handles:

1. Communication between simulation managers,
2. Fossil Collection,
3. GVT Calculation, and
4. Worker thread control.

Each of these functions occur one time per loop as seen in Figure 5.2, and are constantly monitored to

```
simulate():

while !simulationComplete:

simulationManagerCommunication()
fossilCollection()
gvtCalculation()
workerThreadControl()
```

Figure 5.2: simulate() process

```
workerthreadcontrol()

if (busythreads < numworkerthreads-1):
for all ltsfqueues:
if schedulequeue not empty:
wake all threads assigned to schedulequeue
```

Figure 5.3: worker thread control process

```
workerThread():

while workRemaining:

if workExists:
executeObjects(thread)
else:
sleep(thread)
```

Figure 5.4: workerThread() process

determine if any work needs to be performed. Modifications were primarily made to the *Worker thread control* section to allow support for multiple LTSF queues. This section details this process.

Worker threads are placed into a sleep mode when the LTSF queue they are assigned to runs out of work. The worker thread control section monitors thread status, and wakes any thread that was previously in a sleep mode when new events are detected. Only threads assigned to the specific LTSF queue that receives new work are resumed, instead of resuming all threads. Figure 5.3 shows a detailed diagram of the process used for this.

Worker Threads

Each worker thread enters the `workerThread()` function (Figure 5.4) where it begins execution. Here it executes any available work, and sleeps if no work is available. The `executeObjects()` function has several different modes of operation depending on if the thread is in recovery mode or executing straggler events. Each thread has three primary tasks:

1. Retrieve events for execution,

2. Execute events, and

3. Update the appropriate LTSF queue.

Multiple LTSF queues are handled by the the Queue Manager, so no modifications were necessary in this section. The Queue Manager section will explain the necessary changes.

Queue Manager

The *Queue Manager* facilitates access to several different data structures used by the master and worker threads. The unprocessed queues store events pending execution for each LP. The processed queues store events that have been executed and are stored for future rollbacks. The LTSF queue stores the next event to be executed for each LP in a sorted queue to allow for easy event fetching by worker threads.

The Queue Manager has a single LTSF queue that is protected by an atomic lock, allowing only one thread to access and update the data structure at a time. As shown in Figure 5.5, the LTSF queue is simply a queue of type STL Multiset stored in the Queue Manager Class. Modifying the kernel to support multiple LTSF queues involves two steps.

First, the LTSF queue is removed from the Queue Manager Class, and a separate *LTSF Queue* class is created to store the Multiset data structure and facilitate access to this structure. This separates all of the functions and locks necessary for the LTSF queue to allow for the next step. This is shown in *New Kernel Part 1* of Figure 5.5. Step two is to make the number of LTSF queues a configurable parameter, and assigns threads and LPs to LTSF queues. For example, if two LTSF queues and four LPs are used in a simulation, two LPs are assigned to each LTSF queue. This is illustrated in *New Kernel Part2* of Figure 5.5.

Three lookup tables are used to enable access to the proper LTSF queue object when inserting, removing, or modifying objects. As seen in Figure 5.6, *SQByThreadId* is an array of pointers that indicates which LTSF queue object is assigned to each `threadId`. Each thread uses this when retrieving the next event for execution in the `peekEvent()` function. This allows the thread to use the object that it was assigned to quickly and easily.

A second lookup table, *SQByObjId* maps LPs to specific LTSF queues. For example, in Figure 5.6, LPs zero through two are assigned to the first LTSF queue object, and LPs three through five to the second LTSF queue object. Inserting new events occurs in the `insert()` function, to which is passed the id of the thread inserting the event and the event being inserted. From the event, the correct LP to insert into is known, but

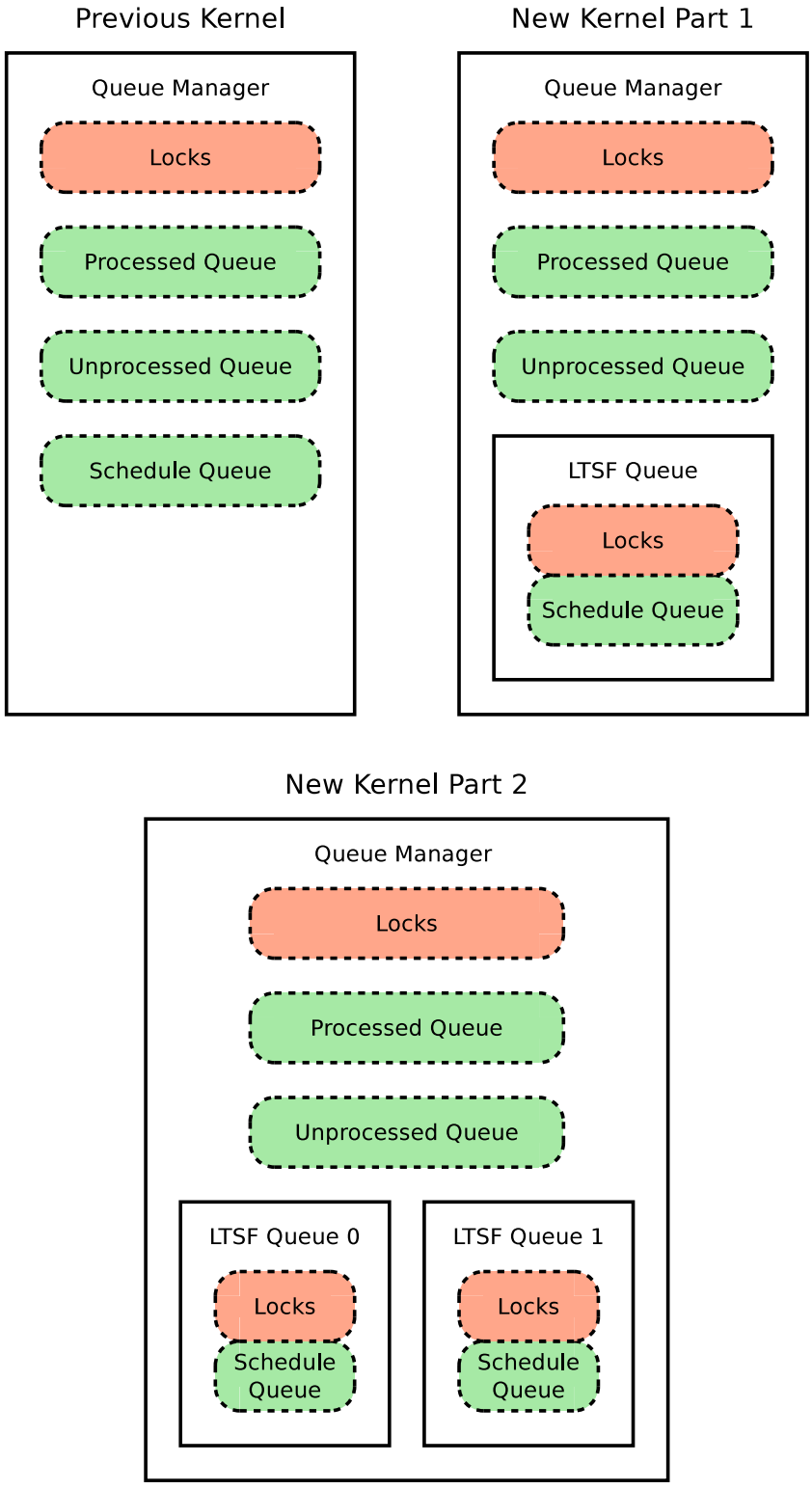

Figure 5.5: Queue Manager Data Structures

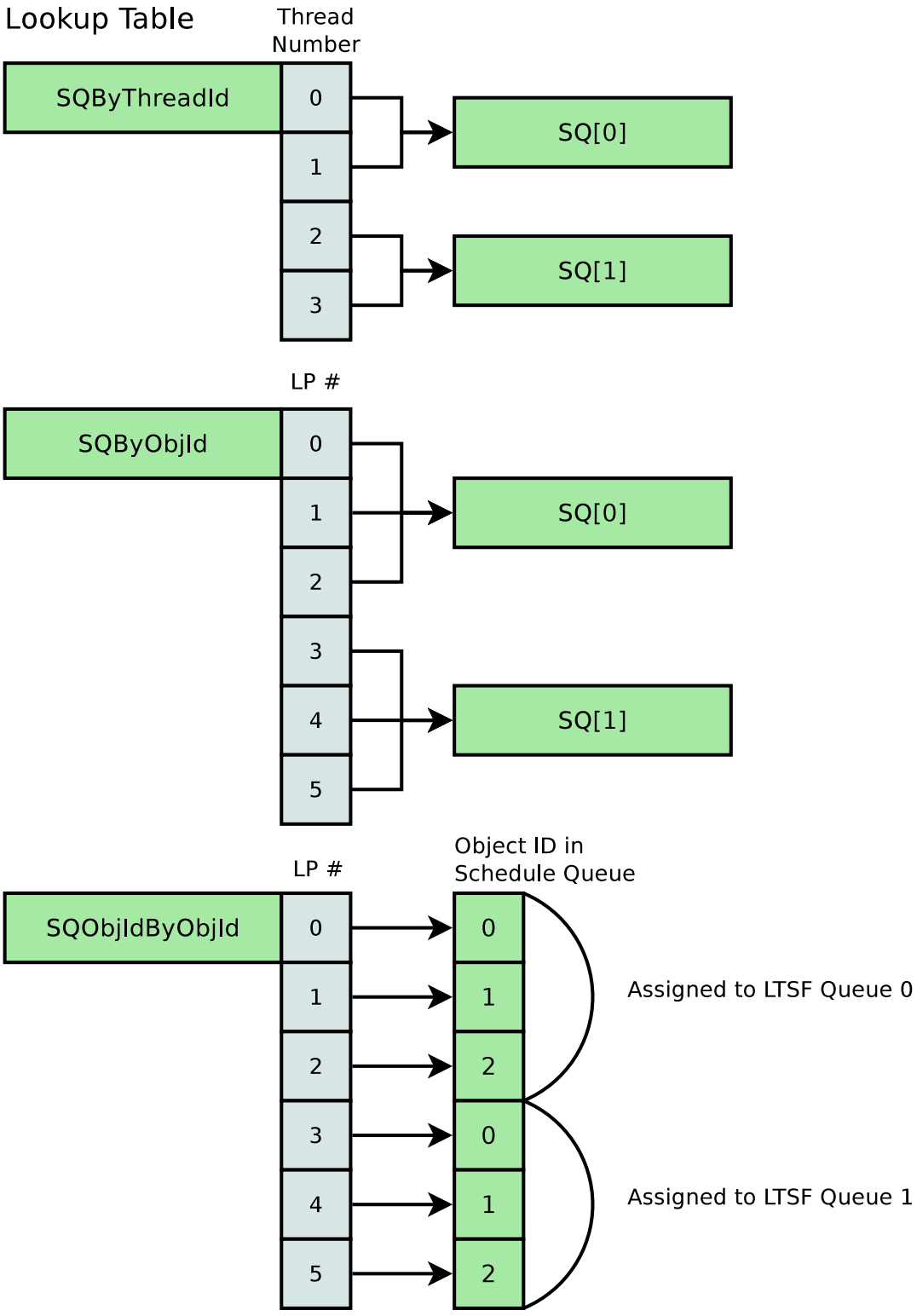

Figure 5.6: LTSF Queue Lookup Tables

the correct LTSF queue object to update is unknown. The *SQByObjId* lookup table is used to determine which LTSF queue object the event is assigned to for inserting the event.

The LTSF queue is updated after each thread executes an event, which occurs in the aptly named `updateScheduleQueueAfterExecute()` function. This retrieves the next event from the LP queue, and places it in the LTSF queue. The *SQByObjId* is used to determine the proper LTSF queue to place the event into which the event is placed.

Several LPs are assigned to each LTSF queue object. Each LP is mapped to a location within the LTSF queue object it is assigned. A third lookup table, *SQObjIDByObjId* is used to store this mapping, referred to as the local object id. For example, as shown in Figure 5.6, with six LPs, three are assigned to each LTSF queue object. LP 3 has a local object number of 0, because it is the first object assigned to LTSF queue one.

Three lookup tables are used to store this information. A simple array is used since the size of each lookup table will not change during execution, since the size is tied to the number of simulation LPs, which will not change. The lookup table objects are pointers to allow for simple and fast lookups for the *SQByThreadId* and *SQByObjId* tables, while integers are used for the *SQObjIdByObjId* lookup table. These lookup tables allow for simple remappings, which will be useful during load balancing LP reassignment. For example, moving an LP to a new LTSF queue object involves updating two lookup tables and moving one event pointer.

LTSF Queue

The *LTSF Queue* class controls access to the LTSF queue data structure via a number of different functions. A separate class is instantiated for each LTSF queue that is used. The lookup tables are used to access the proper LTSF queue as shown in Figure 5.6.

Figure 5.7 shows the main functions in the LTSF queue class, and the associated function each is called from in the Queue Manager class. The `insert()` function inserts a new event into the LTSF queue using the `insertEvent()` LTSF queue function.

The `updateScheduleQueueAfterExecute()` function updates the LTSF queue with the next event from the given LP using the `insertEvent()` function. If the LP has no further events, `insertEmptyEvent()` is used to insert an empty event.

`peekEvent()` fetches the next event for execution by a thread using `peek()`, which returns the next event from the given LTSF queue. `peekEvent()` is called by a worker thread, which uses the SQByThreadId lookup table to access the proper LTSF queue `peek()` function. Several other functions are used, including `nextEventToBeScheduledTime()`, which returns the timestamp of the lowest event in each LTSF queue for GVT calculation, and `clearScheduleQueue()`, which removes all events from the LTSF queue at the end of execution.

Figure 5.7 also shows some of the important data structures used in the LTSF queue class. `scheduleQueue[]` holds all events pending execution in a sorted queue, such as an STL Multiset. `lowestObjectPosition[]` stores the positions of all events within the LTSF queue to allow events from specific LPs to easily be removed. Two locks are used to prevent data destruction issues. `scheduleQueueLock` blocks all other threads from accessing the LTSF queue data structure. `objectStatusLock[]` blocks specific objects

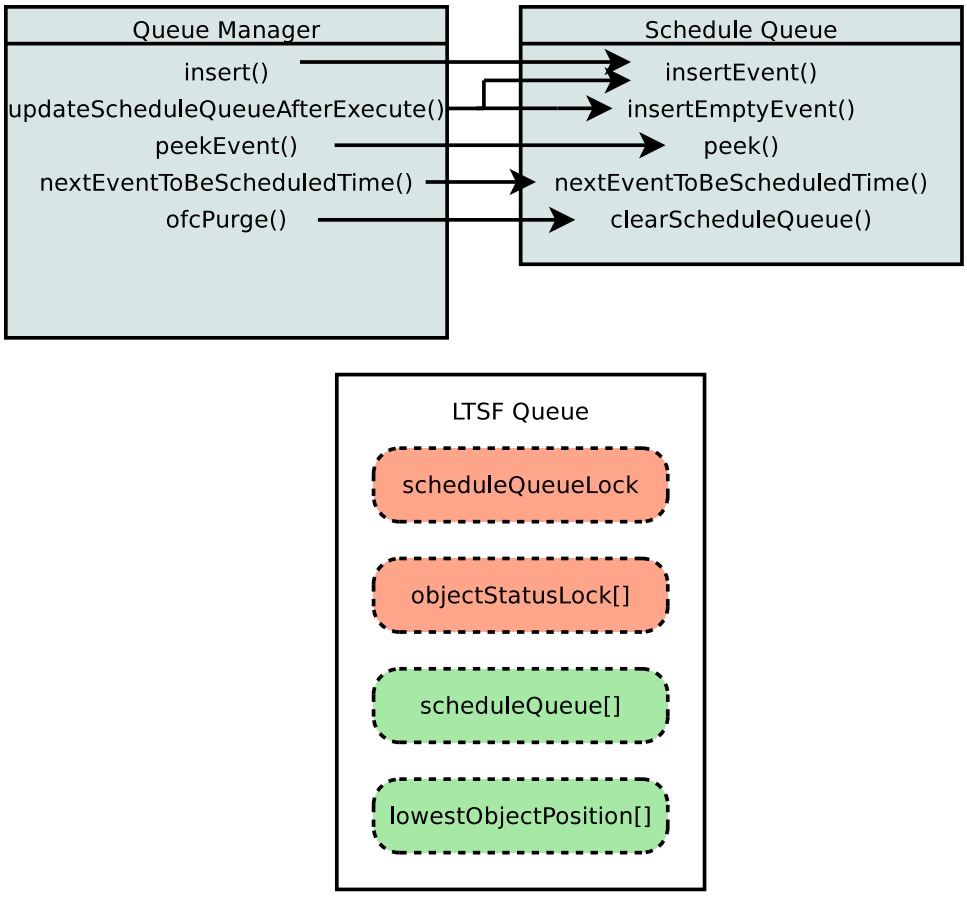

Figure 5.7: LTSF Queue Class Layout

```
TimeWarp{
ThreadControl {
...
}
Scheduler {
Type : MultiSet
ScheduleQScheme : MULTISET
ScheduleQCount : 2
}
...
}
```

Figure 5.8: LTSF Queue Configuration Parameters

within the LTSF queue to prevent modification during execution.

5.2.3 Configuration Parameters

WARPED uses a configuration file to determine the simulation configuration. A configuration parameter is added to the Time Warp `Scheduler` section as shown in Figure 5.8. The Scheduler section controls the scheduler that is used for the threaded WARPED kernel during execution. The `ScheduleQCount` parameter is added to this section to allow for multiple LTSF queues to be used. By setting this value to 1, a single LTSF queue is used, and the configuration is the same as before multiple LTSF queues were introduced.

5.2.4 Challenges

Several challenges were encountered while adding the aforementioned features to the threaded WARPED kernel. The Queue Manager `insert()` function places the requested event into the local unprocessed queue and then into the LTSF queue if it is the lowest timestamped event of that LP. This requires obtaining both the unprocessed lock and the LTSF queue lock to prevent issues from occurring. Previously these locks were obtained one after each other, and held separately, which previously caused no issues.

The introduction of multiple LTSF queues changed the execution profile in such a way that another thread was able to insert an event into the unprocessed queue before the event was placed into the LTSF queue. This caused the event to be executed twice, and in some cases be deleted before it was executed a

second time. This situation occurred infrequently and it was difficult to reproduce and debug.

Another challenge encountered was proper LP to LTSF queue object mapping. Each LP is mapped to a specific LTSF queue, and then to a specific location within that LTSF queue. Initially, the first `n/sq` LPs were assigned to each LTSF queue where `n` is the number of LPs in the simulation, and `sq` is the number of LTSF queues used. For example, the first half of the LPs were assigned to the first LTSF queue, and the second half are assigned to the second LTSF queue if two were used. An offset was stored within the LTSF queue class to denote this mapping.

This forced a specific LP to LTSF queue assignment and limited future enhancements. A lookup table was added, entitled *SQObjIDByObjId*, which mapped LPs to specific LTSF queue positions. Not only does this allow multiple static mappings to be set before execution, which can be beneficial, but it also allows dynamic remapping during execution, which will be useful to facilitate load balancing.

5.3 Load Balancing

5.3.1 Introduction

Creating multiple LTSF queues can lead to uneven loads on each LTSF queue, which leads to large numbers of simulation rollbacks. This second approach extends the multiple LTSF queue approach by allowing dynamic reassignment of LPs between LTSF queues. Load balancing allows the load placed on each LTSF queue to be modified during execution, so the load can be more evenly distributed in an attempt to further reduce simulation time.

5.3.2 Class Layout

This section walks through the relevant sections of the threaded WARPED kernel, and details any relevant additions or modifications that were made to add load balancing support. It details any additional data structures that are added, and the data structures in any classes that are added. Figure 5.9 illustrates the layout of the data structures and the interactions between the associated classes.

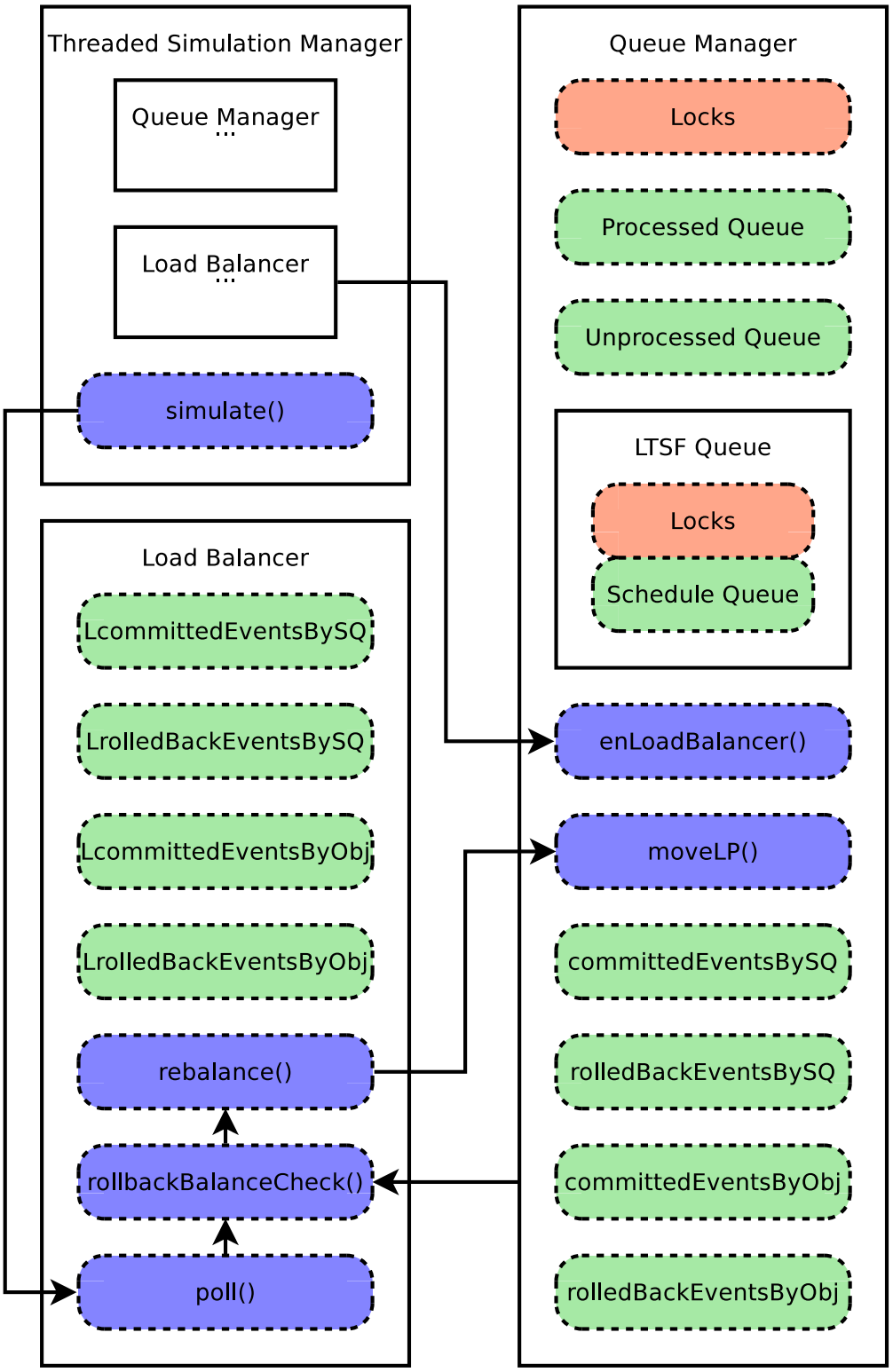

Figure 5.9: Load Balancer Data Structures and Layout

Metrics

Load balancing requires two key components. A metric is used to determine when to rebalance the load of the system, and another metric is used to determine how to rebalalnce the load. For this kernel, load is rebalanced by moving LPs between LTSF queues. Two values are stored during simulation execution for determining these metrics, the number of committed events and the number of rolled back events. This value is stored independently for each LTSF queue. Storing both values is redundant, but it adds minimal processing during simulation and makes calculations easier and faster.

These values are used to compute the *efficiency* of the simulation using the following equation:

$$\frac{committedEvents - rolledBackEvents}{committedEvents}. \tag{5.1}$$

The efficiency is calculated for each LTSF queue when a measurement is triggered. The rollback and committedEvent totals are stored in four integer arrays within the Queue Manager class, `committedEventsBySQ`, `rolledBackEventsBySQ`, `committedEventsByObj`, and `rolledBackEventsByObj`. A separate offset value is stored for each of these values from the last time a calculation was performed. An offset value is used instead of resetting each counter to simplify the process and reduce issues. If this value exceeds some threshold, a rebalance is initiated to equalize the load between LTSF queues using the `rebalance()` function.

Triggers

Triggers are used to initiate load balance checks, and determine if a rebalance needs to occur. Two different triggers are used in this system. The first trigger occurs whenever a rollback is triggered for a LTSF queue. The thread handling the rollback initiates a `rollbackBalanceCheck()`. This provides two benefits. First, the thread performing the rollback is likely off of the critical path, so the rebalance is performed using that thread. Also, rebalance checks are only initiated if rollbacks are occurring. If no rollback is occurring, a rebalance is not necessary, and no CPU time is wasted. A minimum amount of time must elapse between measurements to allow the system to stabilize.

An alternative trigger uses the master thread to poll the load balancer. If a minimum amount of time

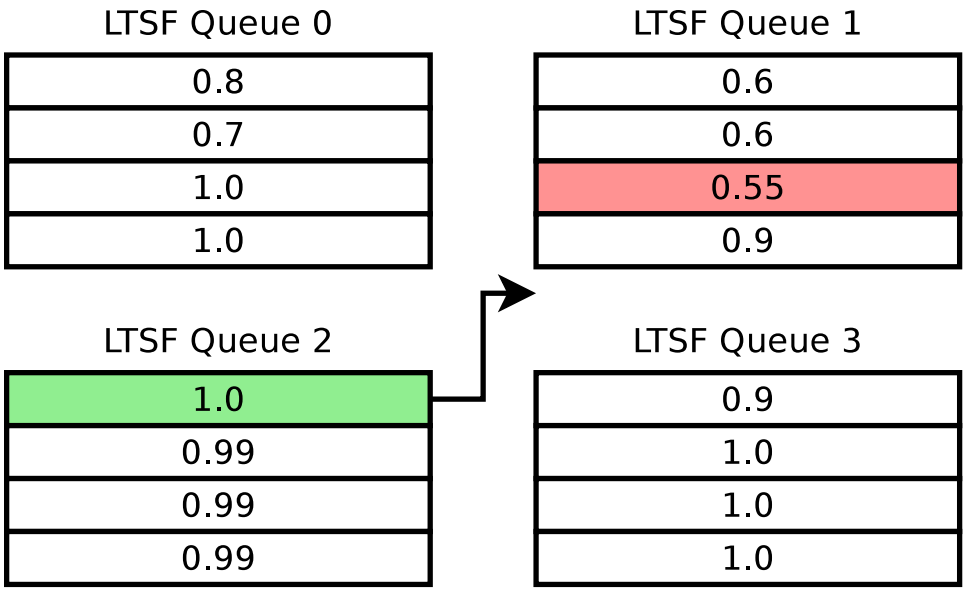

Figure 5.10: Load Balancing Diagram

has passed since the previous balance measurement, the master thread is used to rebalance the system. By using the master thread to rebalance the system, all of the worker threads are free to perform computations without wasting any CPU time on rebalancing or measurements.

Rebalance

If a rebalance check exceeds a preset value, a rebalance is performed using the `rebalance()` function. Figure 5.10 illustrates this process. The simulator selects the highest and lowest metric LTSF queues. The LP from the highest metric LTSF queue with the highest metric is moved to the LTSF queue with the lowest metric. For example, in Figure 5.10, the first LP from LTSF queue 2 is moved to LTSF queue 1. Moving the highest metric LTSF queue helps to distribute the LPs that are currently being executed on the critical path.

5.3.3 Configuration Parameters

Three configuration parameters are added for load balancing support as shown in Figure 5.11. *LoadBalancing* can be set to either ON or OFF, to enable or disable system load balancing. *LoadBalancingMetric* can currently only be set to EventExecutionRatio, but there is an allowance for future metrics to be added. *LoadBalancingInterval* is the minimum amount of time in seconds between load balancing measurements.

```
TimeWarp{
ThreadControl {
WorkerThreadCount : 4
SynMechanism : MUTEX
LoadBalancing : OFF
LoadBalancingMetric : EventExecutionRatio
LoadBalancingInterval : 3
}
Scheduler {
...
}
...
}
```

Figure 5.11: Load Balancing Configuration Parameters

This last parameter provides for the specification of a backoff period for the load balancing algorithm so that the system has time to stabilize before another reblance event occurs.

Chapter 6

Performance Analysis

6.1 Introduction

This chapter presents an overview of the two simulation models and the hardware configuration used to perform a performance analysis on the algorithms developed in this thesis. A detailed analysis of the performance when utilizing multiple LTSF queues is performed. Finally, an evaluation of numerous configurations of our load balancing algorithm and triggers is performed and reported.

6.2 Simulation Models

6.2.1 RAID

The RAID (Redundant Array of Independent Disks) simulation model simulates a RAID 5 disk drive system. A RAID system is composed of several disks connected together with data distributed between the drives, with parity on each disk to prevent data loss. Several simulation objects are used:

Source: Produces read and write requests that are sent to the forks.

Fork: Routes the read and write requests to the proper disk to be fulfilled.

Disk: Represents a disk in the RAID array. Each disk receives events that are routed to it by the fork(s), processes the requests, and returns the results.

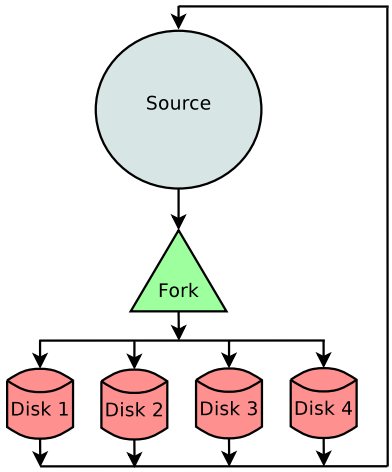

Figure 6.1: RAID-5 Simulation Model

Figure 6.1 shows an example of a RAID simulation construction.

6.2.2 ISCAS-85

The ISCAS benchmarks are a set of logic circuit models used to for research purposes. Each circuit component is a simulation object. Inputs are processed by the gates to produce output values.

Two of the ISCAS-85 circuit models have been built for the WARPED simulator and are included in this report. The c2670 is a 12-bit ALU and controller composed of 1193 gates. The c7552 model is a 32-bit adder / comparator composed of 3512 gates.

6.3 Experimental Setup

6.3.1 Simulation Configurations

Two different simulation types are used to perform the following simulations. A RAID simulation is used that models a RAID 5 disk drive setup. A RAID 5 disk array distributes the parity blocks between the connected disk drives.

The *LargeRAID* simulation model is used in the following simulations. This simulates a RAID5 setup, with 32 disks, 8 forks, and 96 sources. The full configuration file is shown in Appendix A. Each source produces 20,000 requests, for a total of 1.92 million requests. The RAID simulation is run until the GVT

Specification	Details
Processor	AMD Opteron Processor 6168
Sockets	4
Cores	48
Threads	48
Memory	64 GB
Operating System	Ubuntu 11.10 x86_64

Table 6.1: 48-Core Binghamton Machine

value reaches 100,000. This keeps simulation times short enough that multiple simulations can be run and easily compared.

Two *ISCAS-85* simulation models are used in the following simulations. The c2670 circuit is composed of 157 inputs, 64 outputs, and 1193 gates, which makes 1414 LPs (logical processes) in the simulation model. The c7552 circuit is composed of 207 inputs, 108 outputs, 3512 logic gates, for a total of 3827 LPs in the simulation model.

6.3.2 Machine Configuration

The machine used to perform these simulations has the configuration showed in figure 6.1. This machine is used to fully explore and exploit the performance impact of multiple LTSF queues and load balancing.

The introduction of multiple LTSF queues and load balancing results in inconsistent simulation result times. This occurs due to slight variations in the machine starting conditions, runtime conditions, and the dynamic nature of load balancing. Therefore, the median of several simulation runs, typically ten or more, is used for all plots presented below in order to make results comparable.

6.4 Experiments

The following experiments are run to understand and gauge the impact of the optimizations made to the *threaded* WARPED kernel:

1. Performance of multiple LTSF queues with RAID Simulation, and ISCAS c2670 and c7552 circuit simulation models.

2. Performance comparison with different load balancing triggers on RAID simulations.

3. Performance comparison with different load balancing configuration values.

4. Performance comparison with / without load balancing on RAID5 and ISCAS-85 benchmark simulations with different numbers of threads and LTSF queues. This includes partial and full simulations.

5. Performance with different backoff configuration settings.

6.5 Performance Results

6.5.1 Multiple LTSF Queue Performance

The first simulation model tests the performance of the RAID-5 and ISCAS-85 simulation models with varying numbers of threads and LTSF queues. This shows the performance impact of increasing the number of LTSF queues on different numbers of threads. All simulations are performed without load balancing enabled.

Figure 6.2 shows the performance for the RAID-5 simulation model. Increasing the number of LTSF queues from one to two results in a substantial decrease in simulation time. 32 threads shows the lowest simulation time, even lower than 48 threads. The machine used for these simulations has 48 cores, but an extra thread is always used as the manager thread. This results in 49 threads, which is one more than the number of cores in the machine, and likely results in frequent thread switching. 47 threads would likely work better, but would result in an uneven distribution of LPs assigned to each LTSF queue, which would cause an imbalanced simulation.

A noticeable spike occurs for almost all thread configurations when going from four to eight LTSF queues. This distributes the LTSF queues too much, such that the contention is no longer a problem, but the imbalance is a problem. Four LTSF queues seems to be the optimum number for all numbers of threads.

Figures 6.3 and 6.4 show the performance for the ISCAS c2670 and c7552 simulation models. Both show a reduction in the simulation times when the number of LTSF queues is increased from one to two, and then to four. Both the c2670 and c7552 simulation models show a more linear decrease in simulation time as the number of threads increased, likely due to the large number of simulation objects, or LPs. All

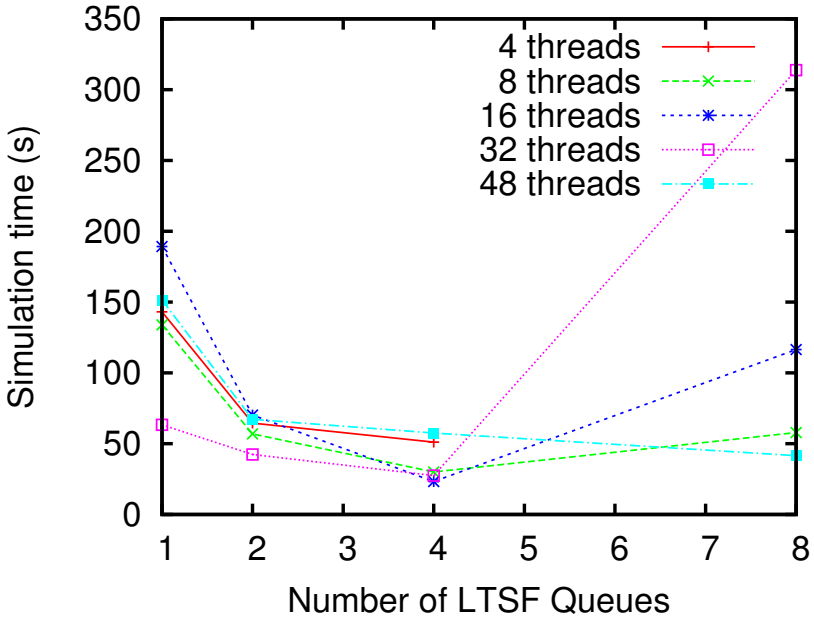

Figure 6.2: Performance of Multiple LTSF Queues on RAID-5

configurations show a minimum simulation time with between four and eight LTSF queues. These results show that the optimal number of threads per LTSF queue is between four and eight.

Muthalagu's [14] research with a single LTSF queue shows that simulation times start to increase when more than six threads are used. This research shows a similar optimal configuration of between four and eight threads per LTSF queue. By using multiple LTSF queues, machines with more than eight simultaneous threads can be more optimally utilized.

When the number of LTSF queues is increased beyond four, the average simulation times increase on average and become highly variant. This can be attributed to the increase in the number of rollbacks as the number of LTSF queues is increased. These rollbacks occur due to the imbalance that this introduces. If one LTSF queue's local time progresses too far ahead of the others, it gets off the critical path, resulting in a large number of rollbacks. This problem becomes more severe as the number of LTSF queues increases beyond four. Four or eight LTSF queues appears to be the optimal number that decreases contention but doesn't imbalance the simulation enough such that the simulation time increases.

Load balancing is a possible solution to this issue. By reassigning LPs between LTSF queues, further reduction of simulation time will hopefully be accomplished.

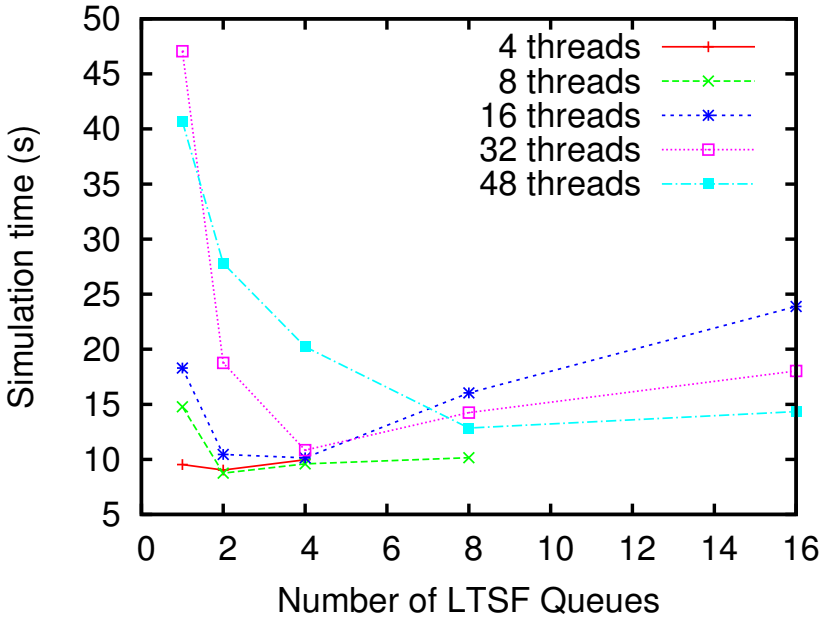

Figure 6.3: Performance of Multiple LTSF Queues on ISCAS c2670

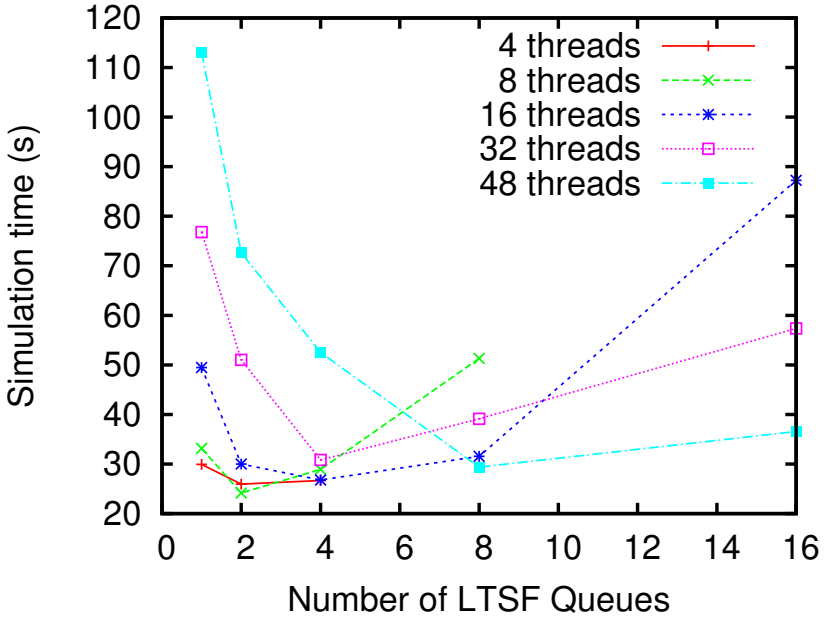

Figure 6.4: Performance of Multiple LTSF Queues on ISCAS c7552

6.5.2 Load Balancing Triggers

The following two sections explore different load balancing configuration options to determine the optimal settings. The determined values are used for the full simulations performed in section 6.5.4.

Triggers are used to initiate load balancing measurements. The following two triggers are implemented in the WARPED load balancer:

Rollback Trigger: The LTSF queues are rebalanced when a worker thread processes a rollback. The rollback is both triggered and processed by the worker thread.

Master Poll Trigger: The master thread calculates the load distribution every *LoadBalancingInterval*, and performs a rebalance if this value exceeds some threshold. The rebalance is performed by the master thread.

Figure 6.5 shows the performance of two different load balancing trigger options, Rollback and Master-Poll on a RAID5 simulation with 16 threads. Both show very similar performance, due to the low impact of measuring and performing load balancing. A load balance measurement requires a simple computation based on the values of some internal counters, and reassigning a LP between LTSF queues requires two simple pointer updates.

The following simulations are all performed using the rollback trigger method due to the benefits described above.

6.5.3 Load Balancing Configuration

Load balancing is controlled by several configuration settings in addition to the trigger settings described in Section 6.5.2. These configuration settings include:

LoadBalancingNormalInterval: The load balancing interval determines the minimum amount of time that must pass between load balancing measurements.

LoadBalancingVarianceThresh: The variance threshold value is the minimum variance for a load rebalance to occur.

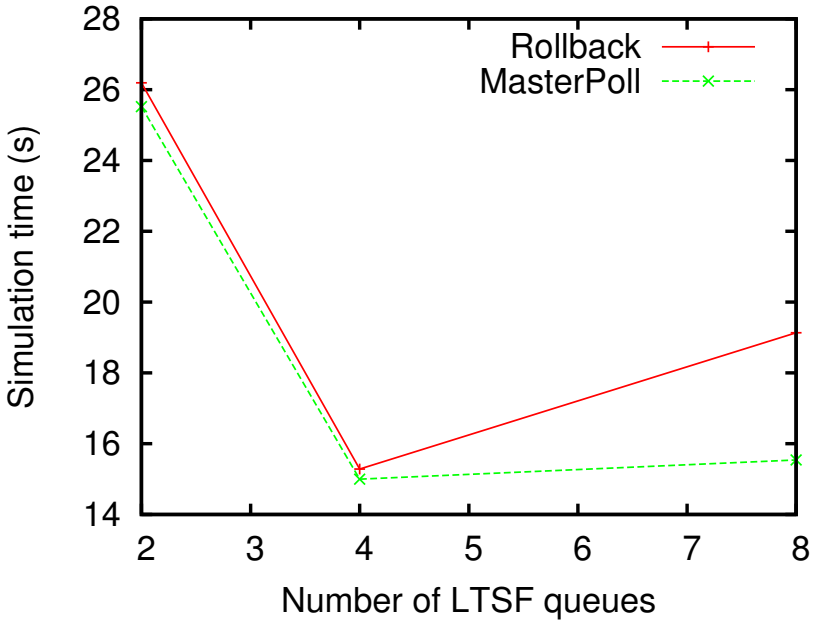

Figure 6.5: Load Balancing Trigger Comparison on RAID5 with 16 Threads

These configuration values control both the sensitivity of the load balancing and how often the load balancing is performed. This section explores the possible configuration values for the `LoadBalancingNormalInterval` and the `LoadBalancingVarianceThresh` values in order to determine the optimal values. The optimal values are those that result in the lowest simulation time. Both simulations performed below are performed with 16 worker threads and 2 LTSF queues.

The first simulation shown below in Figure 6.6 illustrates the results from the `LoadBalancingVarianceThresh` simulations. The simulation times mostly increase as the variance is increased, with the minimum at 0.3. By analyzing these performance results it can be determined that 0.3 is the optimal `LoadBalancingVarianceThresh` value, and should therefore be used for future load balancing simulations.

The second simulation shown below in Figure 6.7 illustrates the simulation times with different `LoadBalancingNorma` values. Interval values between one and five seconds demonstrate performance that is approximately equal, but a value of one generates the best performance. This occurs due to the low cost of a rebalance, and the increased performance that results from faster rebalancing.

These simulations show the optimal variance value is 0.3, and the optimal load balancing interval is one second. These two values are used for the rest of the load balancing simulations for optimal performance.

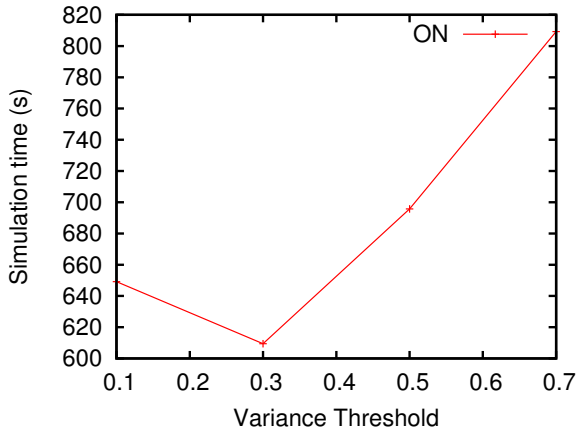

Figure 6.6: Load Balancing Variance Configuration on RAID5 with 16 Threads

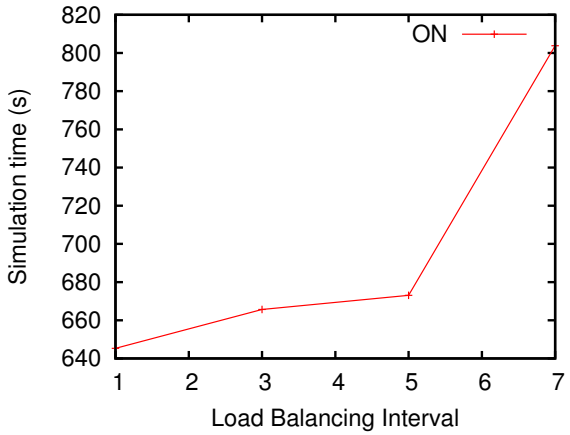

Figure 6.7: Load Balancing Interval Configuration on RAID5 with 16 Threads

6.5.4 Load Balancing Performance

This section compares the performance of several different configurations of load balancing. It observes the performance with different numbers of threads and LTSF queues, with and without load balancing enabled. Several settings are used to control the load balancer:

1. LoadBalancingMetric
2. LoadBalangingTrigger
3. LoadBalancingVarianceThresh
4. LoadBalancingNormalInterval

The follow tests are all performed using the EventExecutionRatio metric, as explained in previous sections. The previous three sections determined the optimal values for the *LoadBalangingTrigger*, the `LoadBalancingVarianceThresh`, and the `LoadBalancingNormalInterval`. These values are shown in the table below, and are used for all of the following simulations. All simulations are performed until the simulation time, or GVT is equal to 100,000 so it is feasible for all configurations to be simulated.

Configuration	Value
LoadBalancingMetric	EventExecutionRatio
LoadBalangingTrigger	Rollback
LoadBalancingVarianceThresh	0.3
LoadBalancingNormalInterval	1 second

Figures 6.8 through 6.11 show the effects of load balancing on the RAID5 simulation time. Each figure shows a different number of threads, with a varying number of LTSF queues across the x-axis.

All simulations have a consistent simulation time with and without load balancing for a single LTSF queue. This is expected since there is a single LTSF queue to which LPs can be assigned. This does show that load balancing measurements have a minimal impact on actual simulation run times.

Four, eight, and sixteen threads show a substantial decrease in simulation time when going from one to two threads. Load balancing reduces the simulation time substantially when two LTSF queues are used for four and eight threads as seen in Figures 6.8 and 6.9.

Four LTSF queues is the optimal configuration for four through sixteen threads. In all cases, load balancing reduces the simulation time, but not as much as it does for two LTSF queues. All configurations have trouble with poor performance when eight or more LTSF queues are used, which is consistent with the tests in the previous section. Load balancing seems to have a varying effect on this depending on the number of threads in the configuration.

Load balancing shows interesting results with 32 threads as shown in Figure 6.11. Simulations with and without load balancing are very similar for up to four LTSF queues, possibly due to the contention introduced by the large number of threads. Eight and sixteen LTSF queues result in a very noticeable negative performance impact with load balancing on.

The performance peaks when four LTSF queues are used for all configurations. Load balancing reassigns a single LP from one LTSF queue to another LTSF queue in an attempt to more evenly distribute the load. The load balancer is less effective when a large number of LTSF queues are used due to the inability of the load balancer to reassign LPs between LTSF queues quickly enough.

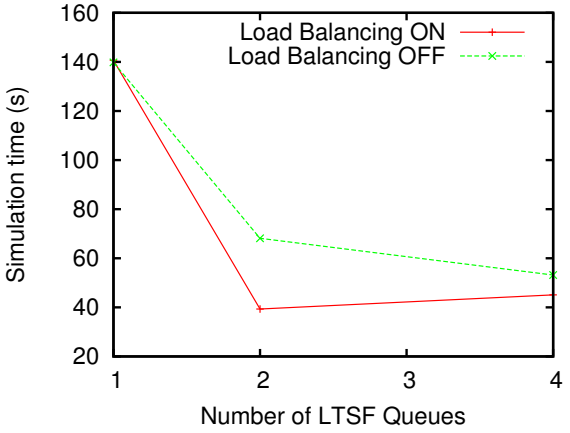

Figure 6.8: Performance of Multiple LTSF Queues and 4 Threads with Load Balancing on RAID5

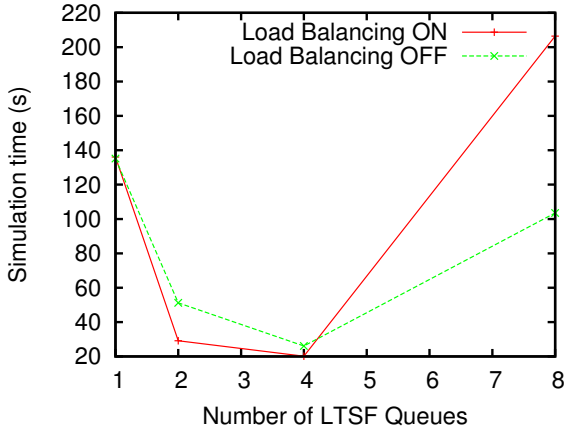

Figure 6.9: Performance of Multiple LTSF Queues and 8 Threads with Load Balancing on RAID5

Figures 6.12 through 6.15 illustrate the impact of load balancing on the ISCAS c2670 simulation model. This simulation is performed with varying numbers of threads and LTSF queues with and without load balancing.

Load balancing reduces the simulation time in almost all instances. This reduction is less significant than with the RAID5 simulation due to the much larger number of LPs. Since load balancing moves a single LP at a time, the load is balanced more slowly, and reacts less quickly. This results in a slower simulation.

Figures 6.16 through 6.19 show the impact of load balancing on the ISCAS c7552 simulation model, again with varying numbers of threads and LTSF queues. Most configurations result in a reduction in

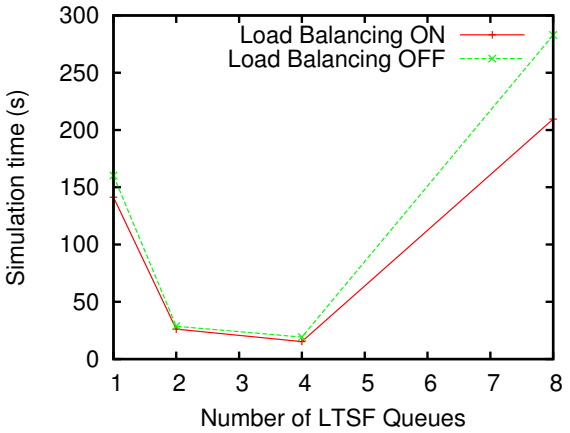

Figure 6.10: Performance of Multiple LTSF Queues and 16 Threads with Load Balancing on RAID5

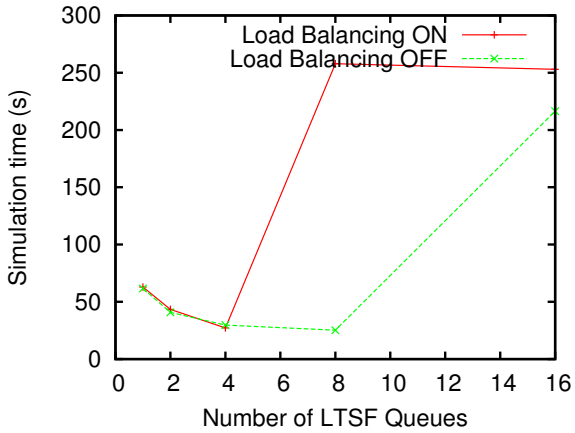

Figure 6.11: Performance of Multiple LTSF Queues and 32 Threads with Load Balancing on RAID5

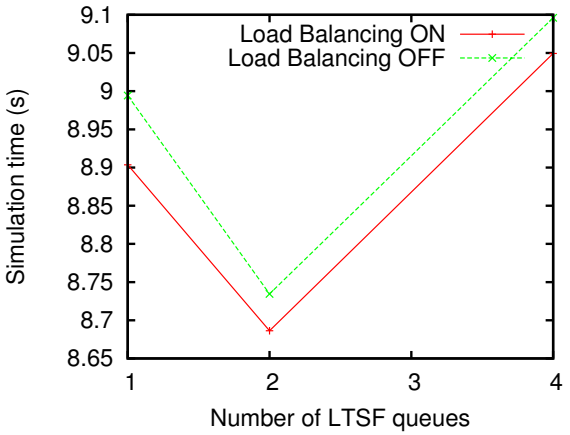

Figure 6.12: Performance of Load Balancing with ISCAS c2670 with 4 Threads

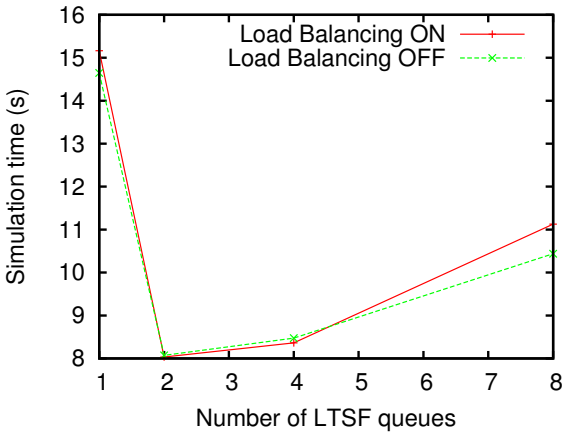

Figure 6.13: Performance of Load Balancing with ISCAS c2670 with 8 Threads

simulation time, but the reduction is not as significant as with the c2670 simulation. The c7552 simulation has significantly more LPs assigned to each LTSF queue, resulting in even slower load balancing, leading to less performance benefit when using load balancing.

All previous simulations were performed using a *simulateUntil* value of 100,000. This shortens the simulations such that it is feasible to run all possible configurations for comparison. The following simulation is a full RAID5 simulation to explore the differences that occur with a full simulation.

Load balancing reevaluates the system load balance periodically, and reassigns LPs between LTSF queues when this balance exceeds a certain threshold. Most rebalancing occurs during the first 50,000 steps

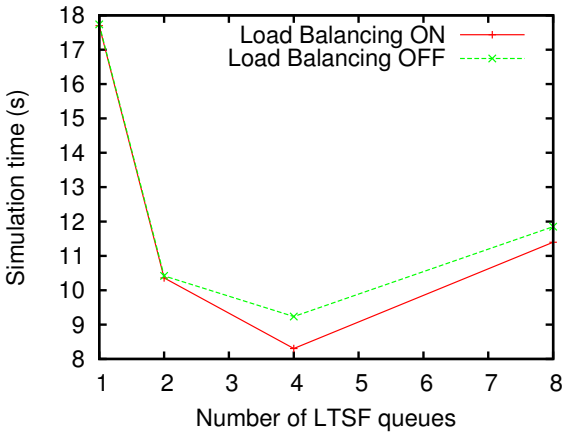

Figure 6.14: Performance of Load Balancing with ISCAS c2670 with 16 Threads

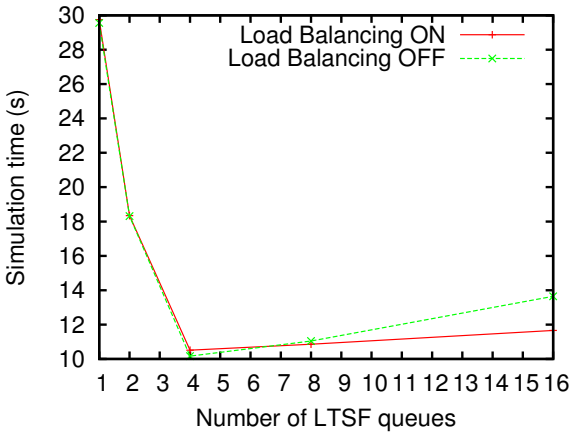

Figure 6.15: Performance of Load Balancing with ISCAS c2670 with 32 Threads

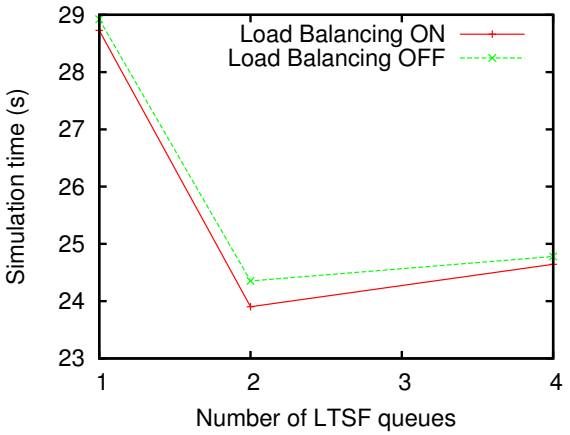

Figure 6.16: Performance of Load Balancing with ISCAS c7552 with 4 Threads

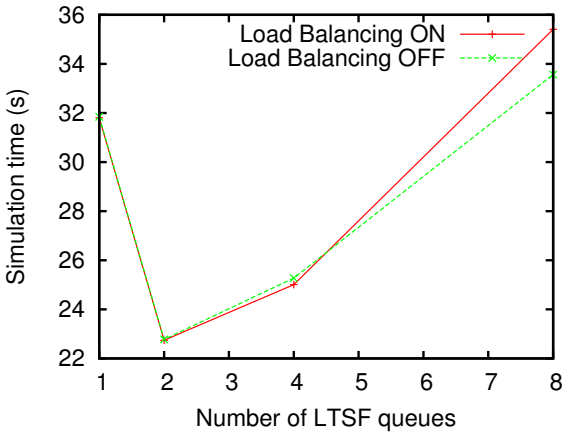

Figure 6.17: Performance of Load Balancing with ISCAS c7552 with 8 Threads

of the simulation, with small reassignments occurring periodically after this point as needed. The simulation shown in Figure 6.20 is performed using 16 threads and a varying number of LTSF queues. The simulation with load balancing results in noticeably better performance than the simulation without load balancing for one to four LTSF queues. In addition, when these results are compared with the partial simulation shown in Figure 6.10, it can be seen that load balancing increases performance much more substantially with a full simulation. In a full simulation, the the system runs for longer after stabilizing, resulting in better overall performance.

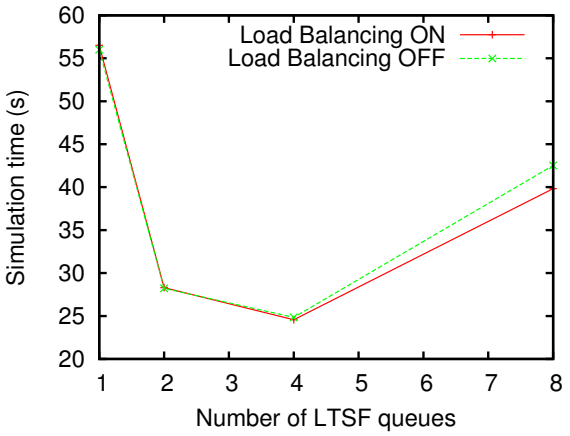

Figure 6.18: Performance of Load Balancing with ISCAS c7552 with 16 Threads

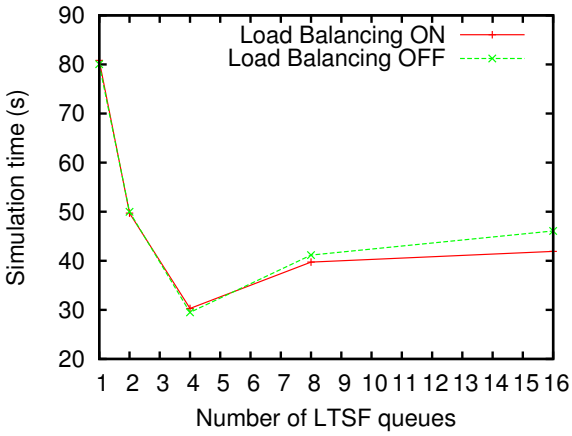

Figure 6.19: Performance of Load Balancing with ISCAS c7552 with 32 Threads

6.5.5 Load Balancing Backoff Performance

Backoff parameters are introduced into the load balancing system in an attempt to further reduce the simulation time. The load balancer mostly reassigns LPs between LTSF queues at the beginning of each simulation, and performs slight modifications later to restabilize the simulation as needed. Backoff parameters reduce the frequency of load balance measurements and reassignments as less reassignments are needed. This causes less CPU time to be dedicated to measurements, and allows the system to stabilize more, and hopefully perform better. Three new system parameters are introduced to enable backoff to be used:

LoadBalancingNormalThresh: The normal threshold determines the number of measurements that must occur without a reassignment for the system to enter the *relaxed* mode of operation.

LoadBalancingRelaxedInterval: The relaxed interval determines the frequency that measurements are taken at (in seconds) after the system enter the relaxed mode of operation.

LoadBalancingRelaxedThresh: The relaxed threshold determines the number of rebalances that must occur in a row for the system to reenter the normal mode of operation.

The system has two separate modes of measurement, namely: *normal* and *relaxed*. Normal mode is used at the beginning when frequent rebalances are occurring. Relaxed mode is used once the simulation has stabilized, and few reassignments are occurring. Several criterion are used to determine when the mode is shifted between normal and relaxed.

Several simulations were performed to determine the optimal configuration settings, which were found to be:

Configuration	**Value**
LoadBalancingNormalThresh	5 measurements
LoadBalangingRelaxedInterval	15 seconds
LoadBalancingRelaxedThresh	2 measurements

Figure 6.21 shows a full RAID5 simulation with 16 threads and a varying number of LTSF queues. One line shows the simulation performance with back off enabled, and one shows the performance with

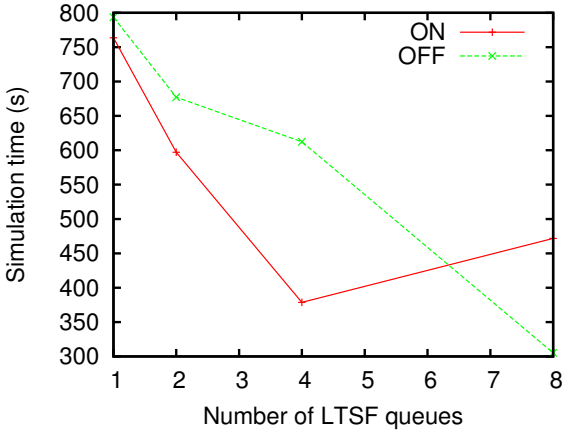

Figure 6.20: Load Balancing for a Full Simulation on RAID5 with 16 Threads

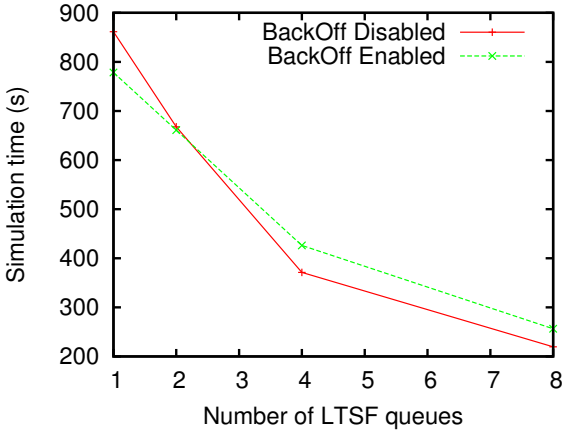

Figure 6.21: Load Balancing with Backoff on RAID5 with 16 Threads

back off disabled. Both configurations result in very similar simulation times, within each others' margin of error. This shows that both rebalance measurements and reassignments have a negligible negative impact on simulation performance.

6.6 Summary

These experiments show that using multiple LTSF queues has a positive impact on simulation performance. This performance improves as more LTSF queues are used, up to some maximum. Load balancing is able to slightly improve performance even more when load balancing is also used to distribute the load between LTSF queues. Configurations with two or four LTSF queues are found to perform the best. Anything beyond that results in a simulation that is too distributed. 16 threads is the optimal configuration, with the shortest simulation times, while still receiving consistent results. A load balancing configuration with a `LoadBalancingVarianceThresh` value of 0.3 and a `LoadBalancingNormalInterval` of 1 second also results in the best performance results.

Chapter 7

Conclusions and Suggestions for Future Research

7.1 Detailed Conclusions

This thesis explored approaches for organizing the Threaded WARPED LTSF queue to improve performance. First, it explored creating multiple LTSF queues to reduce thread contention and increase performance. This improved performance as expected, but when larger numbers of LTSF queues were used, the queues became imbalanced, resulting in a slower simulation. The use of load balancing to dynamically move LPs between LTSF queues was successful in improving performance in almost all configurations. Backoff parameters were also explored to reduce the load balancing frequency as the system stabilized.

7.2 Suggestions for Future Work

The performance results from this research show several places for improvement. This section details several possible paths for exploration of future work.

7.2.1 Lock Free Data Structures

Originally, upcoming events were stored in a single queue, called the LTSF queue. This data structure was protected by a single atomic lock, creating contention when a lot of threads were used. This thesis

created multiple LTSF queues, and distributed the threads between them, reducing the contention. Lock-free data structures offer an interesting alternative queue structure to hold these events. Other researchers at the University of Cincinnati are exploring the use of the LadderQ data structure for the LTSF queue [13]. This structure has faster inserts and retrievals, but also offers some alternative locking structures to optimize performance. Other data structures can also be explored that completely remove the need for locks when accessing and updating the LTSF queue.

7.2.2 Dynamic Load Balancing

Load balancing was introduced in this thesis to improve simulation performance by reassigning threads to maintain a balanced simulation. This showed significant performance improvements when simulations with a low number of LPs were used, such as the RAID5 simulation. Other simulations such as the ISCAS benchmarks use a larger number of LPs. Because single LPs are reassigned, load balancing is slower to react, typically resulting in a slower simulation. Reassigning multiple threads when the load imbalance exceeds a threshold would allow faster rebalances, resulting in lower simulation times.

7.2.3 More Simulation Models

RAIDSim models a RAID controller reading and writing to disk drives. This offers a reasonably large simulation that is based on a real world system. Recently, several of the ISCAS-85 benchmark circuits were ported over to the WARPED kernel, which offers more real world systems with large numbers of LPs that can be used to test kernel improvements. More, larger circuit models would be helpful for further testing, including other simulation models such as a weather simulation.

Appendix A

Configuration File for Large RAID-5 Model

```
32 8 96
Disk1 FUJITSU
Disk2 FUJITSU
Disk3 FUJITSU
Disk4 FUJITSU
Disk5 FUJITSU
Disk6 FUJITSU
Disk7 FUJITSU
Disk8 FUJITSU
Disk9 FUJITSU
Disk10 FUJITSU
Disk11 FUJITSU
Disk12 FUJITSU
Disk13 FUJITSU
Disk14 FUJITSU
Disk15 FUJITSU
Disk16 FUJITSU
Disk17 FUJITSU
```

```
Disk18 FUJITSU
Disk19 FUJITSU
Disk20 FUJITSU
Disk21 FUJITSU
Disk22 FUJITSU
Disk23 FUJITSU
Disk24 FUJITSU
Disk25 FUJITSU
Disk26 FUJITSU
Disk27 FUJITSU
Disk28 FUJITSU
Disk29 FUJITSU
Disk30 FUJITSU
Disk31 FUJITSU
Disk32 FUJITSU
Fork1 4 Disk1 Disk2 Disk3 Disk4 1
Fork2 4 Disk5 Disk6 Disk7 Disk8 1
Fork3 4 Disk9 Disk10 Disk11 Disk12 1
Fork4 4 Disk13 Disk14 Disk15 Disk16 1
Fork5 4 Disk17 Disk18 Disk19 Disk20 1
Fork6 4 Disk21 Disk22 Disk23 Disk24 1
Fork7 4 Disk25 Disk26 Disk27 Disk28 1
Fork8 4 Disk29 Disk30 Disk31 Disk32 1
Source1 Fork1 FUJITSU 20000 1
Source2 Fork2 FUJITSU 20000 1
Source3 Fork3 FUJITSU 20000 1
Source4 Fork4 FUJITSU 20000 1
Source5 Fork5 FUJITSU 20000 1
```

```
Source6 Fork6 FUJITSU 20000 1
Source7 Fork7 FUJITSU 20000 1
Source8 Fork8 FUJITSU 20000 1
Source9 Fork1 FUJITSU 20000 1
Source10 Fork2 FUJITSU 20000 1
Source11 Fork3 FUJITSU 20000 1
Source12 Fork4 FUJITSU 20000 1
Source13 Fork5 FUJITSU 20000 1
Source14 Fork6 FUJITSU 20000 1
Source15 Fork7 FUJITSU 20000 1
Source16 Fork8 FUJITSU 20000 1
Source17 Fork1 FUJITSU 20000 1
Source18 Fork2 FUJITSU 20000 1
Source19 Fork3 FUJITSU 20000 1
Source20 Fork4 FUJITSU 20000 1
Source21 Fork5 FUJITSU 20000 1
Source22 Fork6 FUJITSU 20000 1
Source23 Fork7 FUJITSU 20000 1
Source24 Fork8 FUJITSU 20000 1
Source25 Fork1 FUJITSU 20000 1
Source26 Fork2 FUJITSU 20000 1
Source27 Fork3 FUJITSU 20000 1
Source28 Fork4 FUJITSU 20000 1
Source29 Fork5 FUJITSU 20000 1
Source30 Fork6 FUJITSU 20000 1
Source31 Fork7 FUJITSU 20000 1
Source32 Fork8 FUJITSU 20000 1
Source33 Fork1 FUJITSU 20000 1
```

```
Source34 Fork2 FUJITSU 20000 1
Source35 Fork3 FUJITSU 20000 1
Source36 Fork4 FUJITSU 20000 1
Source37 Fork5 FUJITSU 20000 1
Source38 Fork6 FUJITSU 20000 1
Source39 Fork7 FUJITSU 20000 1
Source40 Fork8 FUJITSU 20000 1
Source41 Fork1 FUJITSU 20000 1
Source42 Fork2 FUJITSU 20000 1
Source43 Fork3 FUJITSU 20000 1
Source44 Fork4 FUJITSU 20000 1
Source45 Fork5 FUJITSU 20000 1
Source46 Fork6 FUJITSU 20000 1
Source47 Fork7 FUJITSU 20000 1
Source48 Fork8 FUJITSU 20000 1
Source49 Fork1 FUJITSU 20000 1
Source50 Fork2 FUJITSU 20000 1
Source51 Fork3 FUJITSU 20000 1
Source52 Fork4 FUJITSU 20000 1
Source53 Fork5 FUJITSU 20000 1
Source54 Fork6 FUJITSU 20000 1
Source55 Fork7 FUJITSU 20000 1
Source56 Fork8 FUJITSU 20000 1
Source57 Fork1 FUJITSU 20000 1
Source58 Fork2 FUJITSU 20000 1
Source59 Fork3 FUJITSU 20000 1
Source60 Fork4 FUJITSU 20000 1
Source61 Fork5 FUJITSU 20000 1
```

```
Source62 Fork6 FUJITSU 20000 1
Source63 Fork7 FUJITSU 20000 1
Source64 Fork8 FUJITSU 20000 1
Source65 Fork1 FUJITSU 20000 1
Source66 Fork2 FUJITSU 20000 1
Source67 Fork3 FUJITSU 20000 1
Source68 Fork4 FUJITSU 20000 1
Source69 Fork5 FUJITSU 20000 1
Source70 Fork6 FUJITSU 20000 1
Source71 Fork7 FUJITSU 20000 1
Source72 Fork8 FUJITSU 20000 1
Source73 Fork1 FUJITSU 20000 1
Source74 Fork2 FUJITSU 20000 1
Source75 Fork3 FUJITSU 20000 1
Source76 Fork4 FUJITSU 20000 1
Source77 Fork5 FUJITSU 20000 1
Source78 Fork6 FUJITSU 20000 1
Source79 Fork7 FUJITSU 20000 1
Source80 Fork8 FUJITSU 20000 1
Source81 Fork1 FUJITSU 20000 1
Source82 Fork2 FUJITSU 20000 1
Source83 Fork3 FUJITSU 20000 1
Source84 Fork4 FUJITSU 20000 1
Source85 Fork5 FUJITSU 20000 1
Source86 Fork6 FUJITSU 20000 1
Source87 Fork7 FUJITSU 20000 1
Source88 Fork8 FUJITSU 20000 1
Source89 Fork1 FUJITSU 20000 1
```

```
Source90 Fork2 FUJITSU 20000 1
Source91 Fork3 FUJITSU 20000 1
Source92 Fork4 FUJITSU 20000 1
Source93 Fork5 FUJITSU 20000 1
Source94 Fork6 FUJITSU 20000 1
Source95 Fork7 FUJITSU 20000 1
Source96 Fork8 FUJITSU 20000 1
```

Bibliography

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

[2] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*. McGraw-Hill, 3rd ed., 2000.

[3] R. King, "Warped redesigned: An api and implementation for discrete event simulation analysis and application development," Master's thesis, University of Cincinnati, Cincinnati, OH, 2010.

[4] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on Software Engineering*, vol. 5, pp. 440–452, Sept. 1979.

[5] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. Wiley Interscience, Jan. 2000.

[6] P. F. Reynolds Jr., "A spectrum of options for parallel simulation," in *Winter Simulation Conference*, pp. 325–332, Society for Computer Simulation, 1988.

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

[8] 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.

[9] 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.

[10] 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.

[11] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Di Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, "Distributed simulation and the Time Warp operating system," in *Proceedings of the 12^{th} SIGOPS — Symposium of Operating Systems Principles*, pp. 77–93, 1987.

[12] D. E. Martin, T. McBrayer, and P. A. Wilsey, "WARPED: A Time Warp simulation kernel for analysis and application development," 1995. (available on the www at `http://www.ece.uc.edu/~paw/warped/`).

[13] T. Dickman, S. Gupta, and P. A. Wilsey, "Event pool structures for pdes on many-core beowulf clusters," *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS 2013)*, May 2013.

[14] K. Muthalagu, "Threaded warped: An optimistic parallel discrete event simulator for clusters fo multicore machines," Master's thesis, School of Electronic and Computing Systems, University of Cincinnati, Cincinnati, OH, Nov. 2012.

[15] J. Wang, D. Ponomarev, and N. Abu-Ghazaleh, "Performance analysis of a multithreaded PDES simulator on multicore clusters," *Workshop on Principles of Advanced and Distributed Simulation (PADS '12)*, pp. 93–95, 2012.

[16] R. Nance and R. Sargent, "Perspectives on the evolution of simulation," *Operations Research*, vol. 50, pp. 161–172, Jan. 2002.

[17] J. Misra, "Distributed discrete-event simulation," *Computing Surveys*, vol. 18, pp. 39–65, Mar. 1986.

[18] R. M. Fujimoto, J. Tsai, and G. Gopalakrishnan, "Design and performance of special purpose hardware for Time Warp," in *Proc. of the 15th Annual International Symposium on Computer Architecture*, pp. 401–408, June 1988.

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

[20] R. E. Bryant, "Simulation on a distributed system," in *Proceedings of the 1st International Conference on Distributed Computing Systems*, (Washington, DC), pp. 544–552, IEEE Computer Society, 1979.

[21] H. Avril and C. Tropper, "The dynamic load balancing of clustered time warp for logic simulation," *SIGSIM Simul. Dig.*, vol. 26, pp. 20–27, July 1996.

[22] K. M. Chandy and R. Sherman, "Space-time and simulation," in *Distributed Simulation*, pp. 53–57, Society for Computer Simulation, 1989.

[23] Z. Xiao, F. Gomes, B. Unger, and J. Cleary, "A fast asynchronous gvt algorithm for shared memory multiprocessor architectures," in *Parallel and Distributed Simulation, 1995. (PADS'95), Proceedings., Ninth Workshop on (Cat. No.95TB8096)*, pp. 203–208, 1995.

[24] 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.

[25] L. M. D'Souza, X. Fan, and P. A. Wilsey, "pGVT: An algorithm for accurate GVT estimation," in *Proc. of the 8th Workshop on Parallel and Distributed Simulation (PADS 94)*, pp. 102–109, Society for Computer Simulation, July 1994.

[26] L. M. D'Souza, "Global virtual time estimation algorithms in optimistically synchronized distributed discrete event driven simulation," Master's thesis, University of Cincinnati, Cincinnati, Ohio, May 1994.

[27] M. Chetlur and P. A. Wilsey, "Causality representation and cancellation mechanism in time warp simulations," in *Proceedings of the fifteenth workshop on Parallel and distributed simulation*, PADS '01, (Washington, DC, USA), pp. 165–172, IEEE Computer Society, 2001.

[28] A. Gafni, "Rollback mechanisms for optimistic distributed simulation systems," in *Distributed Simulation*, pp. 46–53, Society for Computer Simulation, Jan. 1988.

[29] R. Radhakrishnan, L. Moore, and P. A. Wilsey, "External adjustment of runtime parameters in Time Warp synchronized parallel simulators," in *11th International Parallel Processing Symposium, (IPPS'97)*, IEEE Computer Society Press, Apr. 1997.

[30] P. Membrey, D. Hows, and E. Plugge, *Practical Load Balancing, Ride the Performance Tiger*. Apress, 2012.

[31] F. Sarkar and S. K. Das, "Design and implementation of dynamic load balancing algorithms for roll-back reduction in optimistic pdes," in *Proceedings of the 5th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '97, (Washington, DC, USA), pp. 26–, IEEE Computer Society, 1997.

[32] B. Kannikeswaran, "A comparison of load balancing algorithms on warped," Master's thesis, University of Cincinnati, Cincinnati, OH, 1998.

[33] R. Vitali, A. Pellegrini, and F. Quaglia, "Load sharing for optimistic parallel simulations on multi core machines," *AGM SIGMETRICS Performance Evaluation Review*, vol. 40, pp. 2–11, Dec. 2012.

[34] D. Jefferson and P. L. Reiher, "Supercritical speedup," in *Proceedings of the 24^{th} Annual Simulation Symposium* (A. H. Rutan, ed.), pp. 159–168, IEEE Computer Society Press, Apr. 1991.

[35] S. Srinivasan and P. F. Reynolds, Jr., "Non-interfering gvt computation via asynchronous global reductions," in *Proceedings of the 1993 Winter Simulation Conference*, Sept. 1993.

[36] J. O. Henriksen, R. M. O'Keefe, C. D. Pegden, R. G. Sargent, and B. W. Unger, "Implementations of time (panel)," in *Proceedings of the 18th conference on Winter simulation* (D. W. Jones, ed.), WSC '86, (New York, NY, USA), pp. 409–416, ACM, 1986.

[37] "Futuristic intel chip could reshape how computers are built, consumers interact with their pcs and personal devices," Dec. 2009.