Experiments with Hardware-based Transactional Memory in Parallel Simulation

A thesis submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

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

August xx, 2010

by

Joshua Hay
BSxx, University of Cincinnati, 20xx

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

Abstract

Transactional memory is a concurrency control mechanism that dynamically determines when threads may safely execute critical sections of code. It does so by tracking memory accesses performed within a transactional region, or critical section, and detecting when memory operations conflict with other threads. Transactional memory provides the performance of fine-grained locking mechanisms with the simplicity of coarse-grained locking mechanisms.

Parallel Discrete Event Simulation is a problem space that has been studied for many years, but still suffers from significant lock contention on SMP platforms. The pending event set is a crucial element to PDES, and its management is critical to simulation performance. This is especially true for optimistically synchronized PDES, such as those implementing the Time Warp protocol. Rather than prevent causality errors, events are continually scheduled and executed until a causality error is detected.

This thesis explores the use of transactional memory as an alternative to conventional synchronization mechanisms for managing the pending event set in a time warp synchronized parallel simulator. In particular, this thesis examines the use of Intel's hardware transactional memory, TSX, to manage shared access to the pending event set by the simulation threads. In conjunction with transactional memory, other solutions to contention are explored such as the use of multiple queues to hold the pending event set and the binding of threads to these multiple queues. The latter is an attempt to distribute worker threads that attempt to access the same pending event queue. Each configuration is compared with conventional locking mechanisms to evaluate the performance of each in WARPED to reduce contention for the pending event set.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The advent of multi-core processors introduced a new avenue for increased software performance and scalability through multi-threaded programming. However, this avenue came with a toll: the need for synchronization mechanisms between multiple threads of execution, especially during the execution of critical sections. By definition, a critical section is a segment of code accessing a shared resource that can only be executed by one thread at any given time [22]. For example, consider a multi-threaded application that is designed to operate on a shared two-dimensional array. For the sake of simplicity, the programmer uses coarse-grained locking mechanisms to control access to the critical section, *e.g.,* a single atomic lock for the entire structure. The critical section reads a single element, performs a calculation, and updates the element of the array. Once a thread enters the critical section, it locks all other threads out of the entire array until it has completed its task, thus forcing the collection threads to essentially execute sequentially through the critical section even when they are accessing completely independent subparts of the array. This results in lock contention, and consequently negatively impacts performance, as threads must now wait for the currently executing thread to relinquish access to the shared resource. Programmers can employ more fine-grained locking mechanisms to expose concurrency, such as locking individual rows or even individual elements in the previous example. However, this approach is vastly more complicated and error prone [20]; this approach requires the programmer to define and maintain a separate lock for each row or each element. Unfortunately, programmers are limited to using static information to decide when threads must execute a critical section sequentially regardless of whether coarse-grained or fine-grained locking is used.

In the previous example, a scenario arises where one thread will access one element of the two dimensional array, while another thread will access a element in an entirely different row. The programmer only knows that any thread can access any given element at any given time, and thus locks all elements when one thread is executing the critical section. Untapped concurrency can be exposed if the decision to execute a critical section or wait is made dynamically [1].

Transactional memory (TM) is a concurrency control mechanism that attempts to eliminate the static sequential execution of a critical section by dynamically determining when accesses to shared resources can be executed concurrently [20]. In the previous example, instead of using locks, the programmer identifies the critical section as a transactional region (these terms, critical region and transaction, will be used interchangeably hereafter). As the threads enter the transactional region, they attempt to transactionally execute the critical section. The TM system records memory accesses as the transactions execute and finds that the transactions operate on independent regions of the data structure, *i.e.,* there are no conflicting memory accesses. Instead of being forced to execute sequentially by the programmer, the threads are allowed to concurrently and safely execute the critical section. Transactional memory is analogous to traffic roundabouts whereas conventional synchronization mechanisms are analogous to conventional traffic lights [17].

Transactional memory operates on the same principles as database transactions [12]. The processor atomically commits *all* memory operations of a successful transaction or discards *all* memory operations if the transaction should fail (a collision to the updates by the multiple threads occurs). In order for a transaction to execute successfully, it must be executed in isolation, *i.e.,* without conflicting with other transactions/threads memory operations. This is the key principle that allows transactional memory to expose untapped concurrency in multi-threaded applications.

One problem space that could benefit from transactional memory is that of Parallel Discrete Event Simulation (PDES). In Discrete Event Simulation (DES) applications, a physical system is modeled as a collection of Logical Processes (LPs) representing the physical processes of the system. The system being modeled can only change state at discrete points in simulated time and only changes state upon execution of an event [9]. Large simulations, such as those in economics, engineering, and military tactics, require enormous resources and computational time, making it infeasible to execute them on sequential machines. The necessity to perform such large simulations has sparked considerable interest in the parallelization of these simulations. In PDES, the events of the LPs are executed concurrently. To further exploit concurrency, optimistic PDES aggressively schedules events instead of strictly enforcing causal ordering of event execution [8, 9]. This means that events will continue to be scheduled without strict enforcement of their causal order until a causal violation is *detected*. More importantly, the events must be retrieved from a global (and shared) pending event set by one of multiple execution threads, resulting in non-trivial contention for this structure. A key challenge area in PDES is the need for contention-free pending event set management solutions [7]; this will be the primary focus of this research. Transactional memory can help alleviate contention for this shared structure and expose untapped concurrency in the simulation's execution.

Researchers at the University of Cincinnati have developed a PDES kernel called WARPED, that implements the optimistic Time Warp synchronization protocol [9, 13]. In WARPED, events to be scheduled are sorted into a global Least-Time-Stamp-First (LTSF) queue. When a worker thread schedules an event, it locks the LTSF queue and retrieves the event from the head of the queue. Thus, the LTSF becomes the primary source of contention in the WARPED kernel.

1.1 Research Statement

The goal of this thesis is to explore the use of transactional memory in a parallel discrete event simulator. In particular, experiments with transactional memory to manage access to the pending event set data structures of the WARPED parallel discrete event simulation enging are examined.

The primary objective of this research is to *modify the* WARPED *pending event set locking mechanisms to utilize the underlying hardware support for transactional memory on Intel's Hardware Transactional Memory (HTM) supported Haswell platform.* The principal hypothesis is that making the aforementioned modifications will exposed untapped concurrency during simulation execution, thereby improving the performance of WARPED on the Haswell platform.

Due to the wide availability of Intel's HTM supported platforms, it was selected as the focus of this research. Intel's HTM implementation is aptly named Transactional Synchronization Extensions (TSX). This naming will be used to refer to Intel's HTM implementation for the remainder of this study.

While WARPED uses many shared data structures, the focus of this thesis is on the pending event set. It is the primary bottleneck in PDES applications, and hence the primary motivation for this study.

1.2 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 provides a general overview of transactional memory. It gives some examples of other TM implementations and discusses why they do not work as well as TSX. It provides examples of related studies. Finally, it provides an overview of how TSX works and how it is implemented in software.

Chapter 3 discusses practical considerations for the programmer when programming TSX enabled multi-threaded applications. It discusses optimizations to ensure TSX performs optimally, as well as physical limitations of the hardware.

Chapter 4 provides a background of the PDES problem space. It introduces WARPED and some of the implementation details relevant to this study. Previous studies with the WARPED pending event set are also briefly discussed.

Chapter 5 discusses how TSX is implemented in WARPED. It also provides a brief overview of the critical sections utilizing TSX and why TSX will be beneficial.

Chapter 6 provides and discusses the experimental results of this research for several different simulation configurations.

Chapter 7 discusses the accomplishments of this research. It also briefly discusses some areas of future research.

Chapter 2

Background

This section provides a high level explanation of how transactional memory operates. It then introduces other implementations, as well as reasons why they were not explored in this study. Next, it provides some examples of related studies with transactional memory, specifically the implementation used in this study. Finally, it provides an overview of Intel's implementation, Transactional Synchronization Extensions (TSX) and how the programmer can develop TSX enabled multi-threaded applications.

2.1 Transactional Memory Overview

Transactional memory (TM) is a concurrency control mechanism that dynamically determines when two or more threads can safely execute a critical section [20]. The programmer identifies a transactional region, typically a critical section, for monitoring. When the transaction executes, the TM system, whether it is implemented in hardware or software, tracks memory operations performed within the transactional region to determine whether or not two or more transactions conflict with one another, *i.e.,* if any memory accesses conflict with one another. If the threads do not conflict with one another, the transactions can be safely and concurrently executed. If they do conflict, the process must abort the transaction and execute the critical section non-transactionally, *i.e.,* by serializing execution of the critical section with conventional synchronization mechanisms.

As a transaction is executed, the memory operations performed within the transaction are buffered, specifically write operations. Write operations will only be fully executed when the transaction is complete and safe access has been determined. Safe access is determined by comparing the set of addresses each transaction reads from (called the *read-set* and the set of addresses each transation writes to (called the *write-set*). Each transaction builds its own read-set and write-set as it executes. The TM system compares the read-sets and write-sets of all concurrently executing transactions to determine if any memory operations conflict. If the transaction completes execution and the TM system has not detected any conflicting memory operations, the transaction atomically commits all of hte buffered memory operations, henceforth referred to simply as a *commit*.

Whenever safe access does not occur, the transaction cannot safely continue execution. This is referred to as a *data conflict* and only occurs if: (i) one transaction attempts to read a location that is part of another transaction's write-set, or (ii) a transaction attempts to write a location that is part of another transaction's read-set [1]. Once a memory location is written to by a transaction, it cannot be accessed in any way by any other transaction; any access by any other transaction results in a race condition. If such a situation arises, all concurrently executing transactions will abort execution, henceforth referred to simply as an *abort*.

Revisiting the example from Chapter 1, assume that a programmer uses transactional memory synchronization mechanisms to access a shared two dimensional array. Recall that any thread can access any element at any given time. One thread enters the transactional region and begins transactional execution. It adds the element's memory location to its read-set. At the same time, another thread enters the transactional region; however, it accesses a different element. As the first thread continues execution, it adds the element's memory location to its write-set. The second thread adds its element's memory location to its read-set at the same time. However, because the memory location is not part of the first thread's read-set, the threads continue executing concurrently. No memory conflicts are detected in this case and the transactions execute successfully and commit.

Now, assume that another thread enters the transactional region. It begins its read operation on a specific element and adds the element memory address to the read-set. However, another thread is is already writing to that memory location. Because the memory address is tracked in the second transaction's write-set, a data conflict occurs. The two threads cannot execute concurrently and be guaranteed to produce the correct output. Therefore, the transactions must abort. Typically, the threads will retry execution with explicit synchronization. Although, as will be shown later, various retry options are possible.

By definition, a transaction is a series of actions that appears instantaneous and indivisible possessing four key attributes: (1) atomicity, (2) consistency, (3) isolation, and (4) durability [12]. TM operates on the principles of database transactions. The two key attributes for TM are atomicity and isolation; consistency and durability must hold for all multi-threaded operations in multi-threaded applications. Atomicity is guaranteed if: (1) all memory operations performed within the transaction are completed successfully, or (2) it appears as if the performed memory operations were never attempted [12]. Isolation is guaranteed by tracking memory operations as the transactions execute and aborting if any memory operations conflict. If both atomicity and isolation can be guaranteed for all memory operations performed within a critical section, that "critical section" can be executed concurrently [20].

In the case of a commit, the transaction has ensured that its memory operations are executed in isolation from other threads and that *all* of its memory operations are committed, thus satisfying the isolation and atomicity principles. Note that only at this time will the memory operations performed within the transaction become visible to other threads, thus satisfying the appearance of instantaneousness. In the case of an abort due to a data conflict, it is clear that the isolation principle has been violated. It should be noted that transactions can abort for a variety of reasons depending on the implementation [2, 5], but the primary cause is data conflicts. Upon abort, all memory operations are discarded to maintain atomicity.

2.2 Related Studies

There have been many implementations of TM systems since its conception [3–6, 10, 25, 28], mostly in software. Software Transactional Memory (STM) systems implement the memory tracking, conflict detection, write buffering etc. in software, as the name suggests. Most systems are implementation specific, but memory tracking is typically done through some form of logging. While this allows transactional memory enabled applications to be executed on a variety of platforms, performance usually suffers. Gajinov *et al* performed a study with STM by developing a parallel version of the Quake multi-player game server from the ground up using OpenMP parallelizations pragmas and atomic blocks [10]. Their results showed that the logging overhead required for STM resulted in execution times that were 4 to 6 times longer than the sequential version of the server. STM in general has been found to result in significant slowdown [3]. Although STM is more widely available than HTM, this study dismissed it as a potential solution due to the performance reasons discussed above.

Hardware Transactional Memory (HTM) provides the physical resources necessary to implement transactional memory effectively. Many chip manufacturers have added, or at least sought to add, support for HTM in recent years. IBM released one of the first commercially available HTM systems in their Blue Gene/Q machine [25]. Even though they found that this implementation was an improvement over STM, it still incurred significant overhead. AMD's Advanced Synchronization Facility and Sun's Rock processor included support for HTM [5, 6]. However, AMD has not released any HTM enabled processors and Sun's Rock processor was canceled after Sun was acquired by Oracle.

With the release of Intel's Haswell generation processors, Intel's Transactional Synchronization Extensions (TSX) is the currently the only widely available commercial HTM-enabled system. Numerous studies have already been performed with TSX, primarily evaluating its performance capabilities. Chitters *et al* modified Google's write optimized persistent key-value store, LevelDB, to use TSX based synchronization instead of a global mutex. Their implementation shower 20-25% increased throughput for write-only workloads and increased throughput for 50% read / 50% write workloads [4]. Wang *et al* studied the performance scalability of a concurrent skip-list using TSX Restricted Transactional Memory (RTM). They compared the TSX implementation to a fine-grain locking implementation and a lock-free implementation, and found that the performance was comparable to the lock-free implementation without the added complexity [27]. Yoo *et al* evaluated the performance of TSX using high-performance computing (HPC) workloads, as well as in a user-level TCP/IP stack. They measured an average speed up of 1.41x and 1.31x respectively [28]. The decision to use Intel's TSX for this research was based on its wide availability and the performance improvements observed in other studies.

2.3 Transactional Synchronization Extensions (TSX)

Intel's Transactional Synchronization Extensions (TSX) is an extension to the x86 instruction set architecture that adds support for HTM. TSX operates in the L1 cache using the cache coherence protocol [2]. It is a best effort implementation, meaning it does not guarantee transactions will commit [1]. TSX has two interfaces: 1) Hardware Lock Elision (HLE), and 2) Restricted Transactional Memory (RTM). While both operate on the same principles of transactional memory, they have subtle differences. This section discusses some of the implementation details of TSX as well as how the programmer utilizes TSX.

2.3.1 Hardware Lock Elision (HLE)

The Hardware Lock Elision (HLE) interface is a legacy-compatible interface introducing two instruction prefixes: 1) XACQUIRE and 2) XRELEASE. XACQUIRE is placed before a locking instruction to mark the beginning of a transaction. XRELEASE is placed before an unlocking instruction to mark the end of a transaction.

These prefixes tell the processor to elide the write operation to the lock variable during lock acquisition/release. When the processor encounters an XACQUIRE prefixed lock instruction, it transitions to transactional execution. Specifically, it adds the lock variable to the transaction's read-set instead of issuing any write requests to the lock [1]. To other threads, the lock will appear to be free, thus allowing those threads to enter the critical section and execute concurrently. All transactions can execute concurrently as long as no transactions abort and explicitly write to the lock variable. If that were to happen, a data conflict technically occurs - one transaction writes to a memory location that is part of another transaction's read-set.

The XRELEASE prefix is placed before the instruction used to release the lock. It also attempts to elide the write associated with the lock release instruction. If the lock release instruction attempts to restore the lock to the value it had prior to the XACQUIRE prefixed locking instruction, the write operation on the lock is elided [1]. It is at this time that the processor attempts to commit the transaction.

However, if the transaction aborts for any reason, the region will be re-executed non-transactionally. If the processor encounters an abort condition, it will discard all memory operations performed within the transaction, return to the locking instruction, and resume execution without lock elision, *i.e.,* the write operation will be performed on the lock variable. This guarantees forward progress [1].

A general implementation for the HLE software interface is shown in Figure 2.1. All the programmer has to do is add the HLE memory model parameters in the locking function intrinsic. GCC 4.8 and above includes support for the `__ATOMIC_HLE_ACQUIRE` and `__ATOMIC_HLE_RELEASE` memory models [24].

HLE is legacy compatible. Code utilizing the HLE interface can be executed on legacy hardware, but the HLE prefixes will be ignored [1], *i.e.,* the processor will always perform the write operation on the locking variable and execute the critical section non-transactionally. While this interface does nothing for multi-threaded applications on legacy hardware, it does allow for easier cross-platform code deployment.

2.3.2 Restricted Transactional Memory (RTM)

The Restricted Transactional Memory (RTM) interface introduces four new instructions: 1) XBEGIN, 2) XEND, 3) XABORT, and 4) XTEST. XBEGIN marks the start of a transaction, while XEND makes the end of a transaction. XABORT is used by the programmer to manually abort a transaction. XTEST can be used to test if the processor is executing transactionally or non-transactionally.

The XBEGIN instruction transitions the processor into transactional execution [1]. Note that the XBEGIN instruction does not jead the locking variable as HLE does. Therefore, the programmer should manually add the locking variable to the transaction's read-set by checking if the lock is free at the start of the transaction. If it is free, the transaction can execute safely. Once execution reaches the XEND instruction, the processor will attempt to commit the transaction.

As previously mentioned, the transaction can abort for many reasons. One case specific to RTM occurs when the lock is not free upon entering the transaction. In this case, the programmer uses the XABORT instruction to explicitly abort the transaction. But no matter the reason for the abort, execution jumps to the fallback instruction address [1]. This address is an operand of the XBEGIN instruction.

It is this fallback path that makes RTM a much more flexible interface than HLE because it is entirely at the discretion of the programmer. Even so, the programmer must still provide an abort path that guarantees forward progress [1]. Therefore, the abort path should use explicit synchronization, *e.g.,* acquire the lock, to ensure forward progress. However, the programmer can use this abort path to tune the performance of RTM enabled applications. For instance, a retry routine can be used to specify how many times the processor should attempt to enter transactional execution before using explicit synchronization. Furthermore, the EAX register reports information about the condition of an abort [1], such as whether or not the abort was caused by the XABORT instruction, a data conflict, etc. The programmer can use this information to make more informed decisions regarding reattempting transactional execution.

The general algorithm for the RTM software interface is shown in Figure 2.2. The programmer moves the existing locking mechanism inside an else clause of the xbegin if statement, which will determine if the processor transitions to transactional execution or takes the abort path. As previously mentioned, the processor will also return to this point should the transaction abort in the middle of execution. Moving the locking mechanism into the RTM abort path ensures that the abort path ultimately uses explicit synchronization and guarantees forward progress. GCC 4.8 and above includes support for the `_xbegin`, `_xabort`, and `_xend` intrinsics [24].

While RTM is a much more flexible interface than HLE, it can only be used on supported Haswell platforms. If a legacy device attempts to execute one of the RTM instructions, it will throw a General Protection Fault. It should be noted that execution of the XEND instruction outside of a transaction will result in a General Protection Fault as well [2].

```
/* Acquire lock with lock elision */
while(__atomic_exchange_n(&lock, 1, __ATOMIC_HLE_ACQUIRE|__ATOMIC_ACQUIRE));

/* Begin Critical Section transactionally or with lock */
...
/* End Critical Section */

/* Free lock with lock elision */
__atomic_store_n(&lock, 0, __ATOMIC_HLE_RELEASE|__ATOMIC_RELEASE);
```

Figure 2.1: **Generic HLE Software Interface**

```
/* Start transactional region.  Return here on abort */
if (_xbegin() == _XBEGIN_STARTED) {
     /* Add lock to read-set */
     if (lock is not free) {
           /* Abort if lock is already acquired */
           _xabort(_ABORT_LOCK_BUSY);
     }
} else {
     /* Abort path */
     acquire lock;
}

/* Begin Critical Section transactionally or with lock */
...
/* End Critical Section */

if (lock is free) {
     /* End transaction */
     _xend();
} else {
     release lock
}
```

Figure 2.2: **Generic RTM Software Interface**

Chapter 3

Practical Programming with TSX

Before implementing TSX in the WARPED simulation kernel, a more in depth evaluation of its capabilities needed to be performed. One of the disadvantages of HTM is the physical limitations of the hardware. This section evaluates practical programming techniques to consider when using TSX to ensure optimal performance. Custom benchmarks were developed to evaluate these various constraints. All benchmarks were run on a system with an Intel i7-4770 running at 3.4GHz with 32 GB RAM. Each core has a 32KB 8-way, set associative L1 cache and a 256 L2 cache. Each cache line is 64 bytes. This information was verified using common Unix commands. All measurements were performed ten times and averaged.

3.1 Memory Organization

TSX maintains a read-set and a write-set with the granularity of a cache line [1]. During transactional execution, TSX constructs a record of memory addresses read from and a record of memory addresses written to. A data conflict occurs if another thread tries to read an address in the write-set or tries to write an address in the read-set. This definition can be expanded to state that *a data conflict occurs if: 1) another thread attempts to read a memory address that occupies the same cache line as a memory address to be written, or 2) another thread attempts to write a memory address that occupies the same cache line as a memory address that has been read from.*

Therefore, aborts can be caused by data occupying the same cache line, especially false sharing, *i.e.,* the occupation of the same cache line by unrelated data [2]. To mitigate the effects of shared cache line data conflicts, the programmer must be conscientious of how data is organized in memory. For instance, the data in the previously discussed benchmarks is optimally organized by allocating individual elements to 64 bytes, *i.e.,* a single cache line.

Furthermore, data elements should be aligned to cache line boundaries to ensure that each element is limited to exactly one cache line. If a data element crosses a cache line boundary, the probability of shared cache line data conflicts increases as the data access now has to check against two cache lines.

3.2 Transaction Size

TSX maintains a transaction read-set and write-set in the L1 cache [2]. The size of these memory sets is therefore limited by the size of the L1 cache. Hyper-threading further restricts the size of the transaction data sets because the L1 cache is shared between two threads on the same core [2]. Based on granularity of the read-set and write-set stated above, transaction size is defined as the number of cache lines accessed within a transaction.

The first two benchmarks evaluate the size restrictions of a transaction's read-set and write-set, *i.e.,* how many cache lines can a transaction track in each set during transactional execution. The benchmarks access a shared array of custom structures. Each structure is allocated to occupy an entire cache line and aligned to the nearest cache line boundary using the GCC align attribute. This ensures that memory is optimally organized as previously discussed.

The objective of the first benchmark is to evaluate the read and write-set sizes for a transaction being executed by a single thread on a single core. Furthermore, only one thread is used to avoid data conflicts. The critical section performs a certain number of strictly read or strictly write operations in the body of a loop. The loop increments an array index to a limit specified by the transaction size being tested, and the body of the loop accesses every single element in that range. This is repeated one hundred times. The number of elements, or cache lines, accessed during the critical section is doubled with every iteration of the main test loop. Figure 3.1 shows the abort rate, (# aborts / 100 operations), as the number of cache lines accessed within the transaction is increased. The read-set data points represent strictly read operations while the write-set data points represent strictly write operations.

It is clear that transactions abort 100% of the time once the thread tries to write to 512 or more cache lines within the transaction. This is consistent with the size of the L1 cache, 32KB of 64 bytes caches lines equates to 512 cache lines; it is unrealistic to expect that no other process will use the cache while the transaction is executing and thus the transaction cannot occupy the cache in its entirety. Note that the cache is split into 64 8-way sets; if memory is not organized properly, the total write-set size will be reduced.

It is evident that the same size limitations do not hold for the read-set size. While eviction of a cache line containing a write-set address will always cause a transactional abort, eviction of a cache line containing a read-set address may not cause an immediate transactional abort; these cache lines may be tracked by a second-level structure in the L2 cache [2].

The objective of the second benchmark is to evaluate the read and write-set sizes for a transaction being executed by a single thread on a shared core, *i.e., a hyper-threaded core*. This benchmark uses the same procedure as above, but with two threads bound to the same core. Each thread accesses the same number of cache lines, but at different memory locations to prevent any data conflicts. Figure 3.2 shows the abort rate for one of the threads as the number of cache lines accessed within each transaction increases.

It is evident that the write-set is strictly limited to half of the L1 cache. However, the probability of an abort is non-trivial for any write-set size between 32 and 128 cache lines. It is also evident that the read-set size is limited to a similar size as the write-set on a hyper-threaded core.

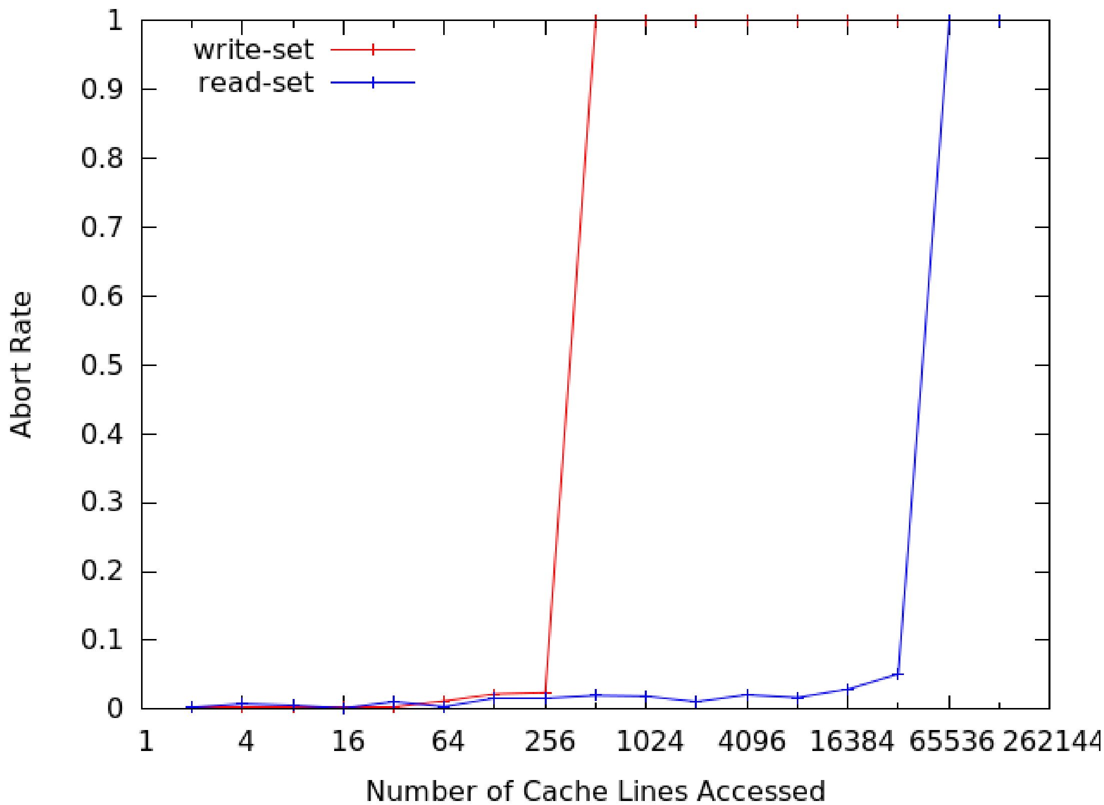

Figure 3.1: **TSX RTM abort rate versus cache lines accessed for a single thread on one core**

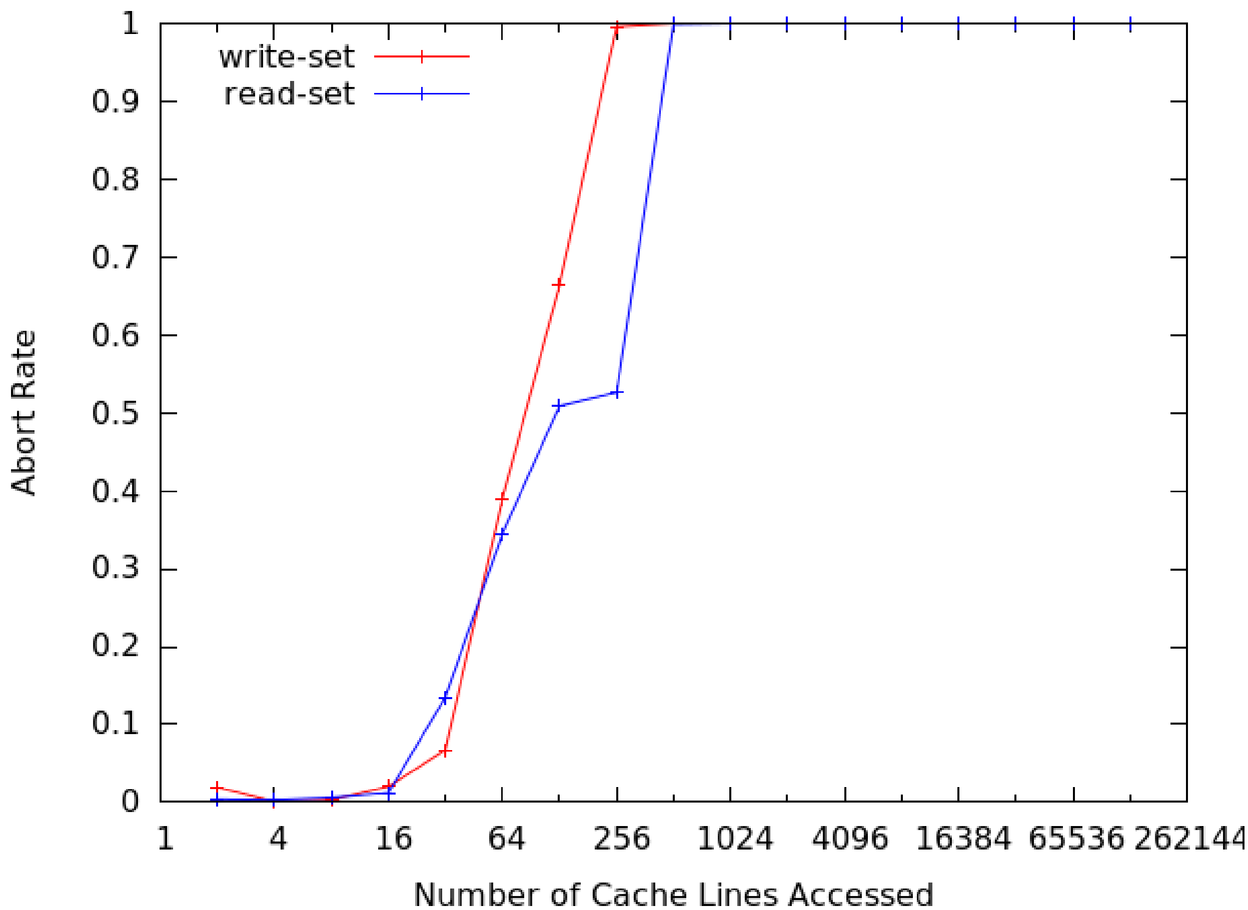

Figure 3.2: **TSX RTM abort rate versus cache line accesses for two threads on hyper-threaded core**

3.3 Transaction Duration

Transaction aborts can be caused by a number of run-time events [1], including but not limited to: interrupts, page faults, I/O operations, context switches, illegal instructions, etc. This is due to the inability of the processor to save the transactional state information [18].

The objective of the third benchmark is to evaluate the running time restrictions for a transaction, *i.e.*, how long can a transaction safely execute without failing. The duration of each transaction is increased by increasing the number of operations performed within the transaction. The critical section performs a certain number of increment operations on a single data element in the body of loop. The loop increments to a limit specified by the duration being tested. The operation count or duration is increased logarithmically from 1000 to 1000000 every iteration of the main test loop. Figure 3.3 shows the transaction abort rate as the duration of the transaction is increased.

It is clear that the longer a transaction executes, the higher the probability is that it will abort. Practical applications will perform a varying number of operations that take varying amounts of time. This benchmark simply demonstrates there is a limit to how long a transaction can be executed. Shorter transactions are more likely to succeed than longer transactions.

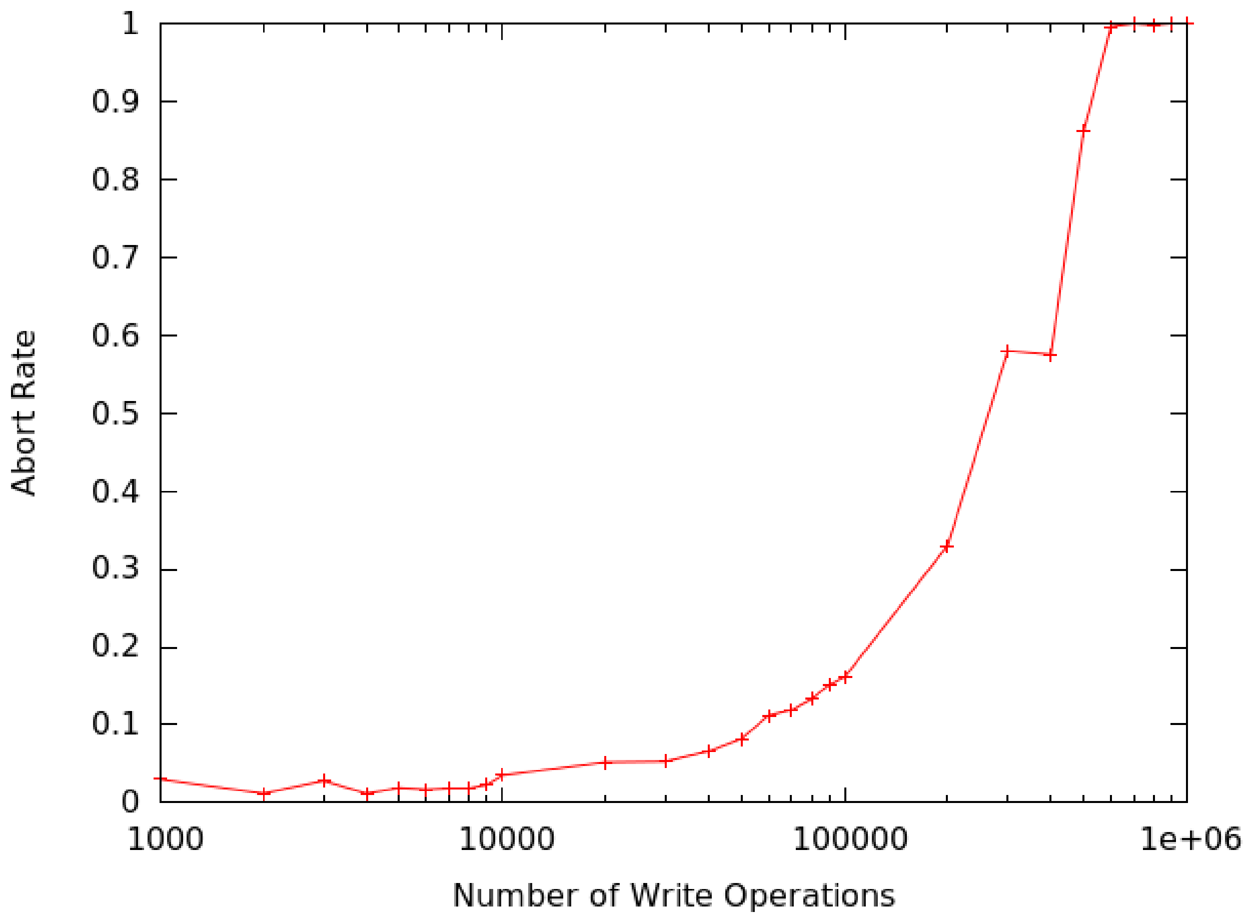

Figure 3.3: **TSX RTM abort rate versus number of operations performed during transaction**

3.4 Synchronization Latency

Conventional synchronization mechanisms have varying latencies, therefore TSX most likely also has varying latencies. While it is incredibly difficult to obtain accurate measurements, the objective of this benchmark is to compare the TSX latencies to conventional synchronization mechanism latencies. This benchmark merely demonstrates how long TSX synchronization mechanisms take to enter a transactional region relative to how long conventional synchronization mechanisms take to enter a critical section.

Each synchronization mechanism is used to perform a simple increment operation. The thread calls the locking function, increments the data, and calls the release function 100000 times. The execution time of the entire loop is measured using the gettimeofday functionality in Linux. The locking/release functions use one of the following depending on the configuration:

1. no synchronization
2. an atomic compare and exchange lock
3. a mutex lock
4. HLE
5. RTM

The results are shown in Figure 3.4. Clearly the HLE and RTM mechanisms take longer to actually complete the synchronization process. This can most likely be attributed to the extra actions performed by the hardware to initiate transactional execution.

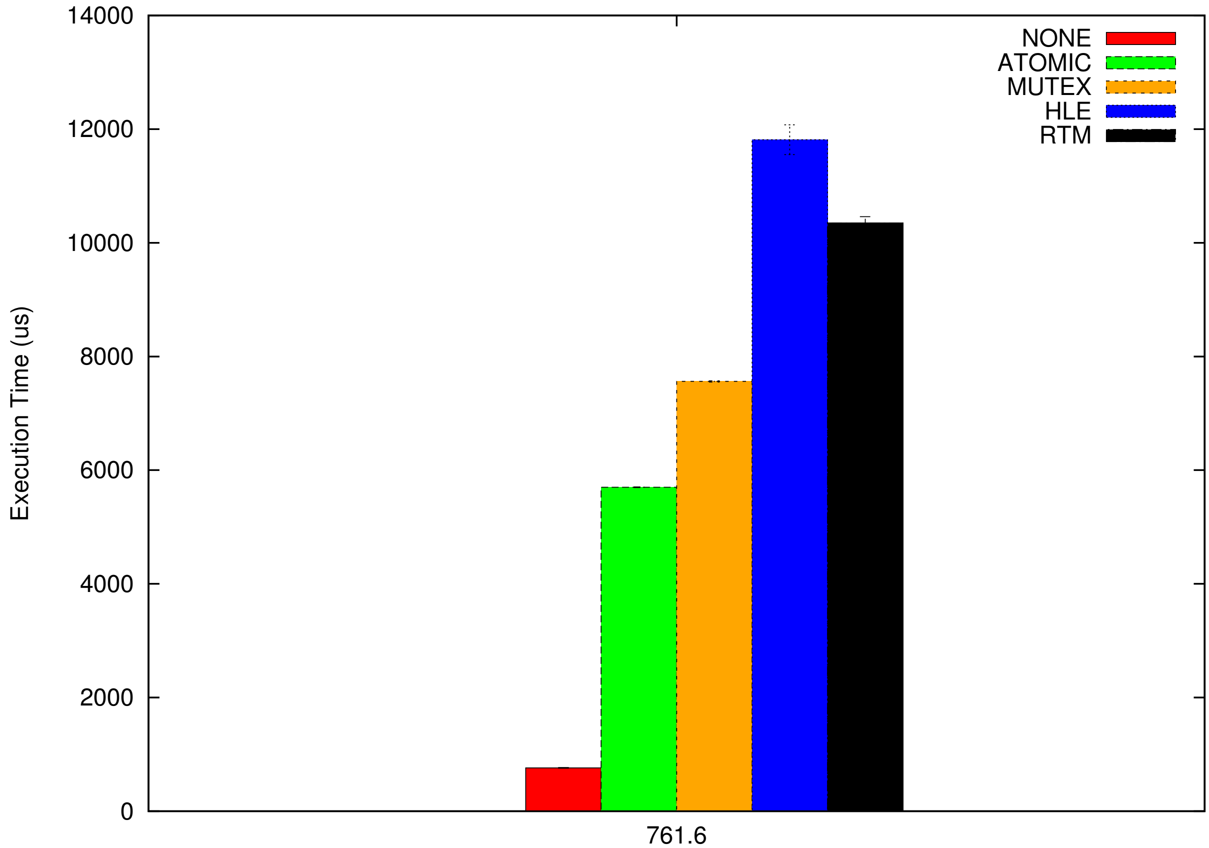

Figure 3.4: **Synchronization Latency**

3.5 Nesting Transactions

When developing larger TSX enabled multi-threaded applications, it is possible for critical sections to be nested within one another. TSX supports nested transactions for both HLE and RTM regions, as well as a combination of the two. When the processor encounters an XACQUIRE instruction prefix or an XBEGIN instruction, it increments a nesting count. Note that the processor transitions to transactional execution when the nesting count goes from 0 to 1 [1]. When the processor encounters an XRELEASE instruction prefix or an XEND instruction, the nesting count is decremented. Once the nesting count returns to 0, the processor attempts to commit the transactions as one monolithic transaction [1].

The total nesting depth is still limited by the physical resources of the hardware. If the nesting count exceeds this implementation specific limit, the transaction may abort. Upon abort, the processor transitions to non-transactional execution as if the first lock instruction was executed without elision [1].

Scenarios may arise where different locks may be nested within the same critical section. For instance, one critical section may reside within a separate critical section. While this is not a concern for RTM regions, it can become a concern for HLE regions, as the processor can only track a fixed number of HLE prefixed locks. However, any HLE prefixed locks executed after this implementation specific limit has been reached will simply execute without elision; consequently, the secondary lock variable will be added to the transaction's write-set [1].

Chapter 4

PDES and WARPED

4.1 Background

Discrete Event Simulation (DES) models a system's state changes at discrete points in time. In general, simulators consist of the following data structures [9]:

- **Pending Event Set:** contains events that have been scheduled, but not processed. Events are retrieved from this structure to be executed.
- **Clock:** denotes how far the simulation has progressed.
- **State:** describes the state of the system.

The state of the simulation can only change upon execution of an event. During the execution of an event, the simulation: 1) retrieves the least time-stamped event from the pending event set, 2) processes the event, 3) updates the LP's state, and 4) if necessary, inserts generated events into the pending event set.

Physical processes are represented by Logical Processes (LPs) in the simulation [15]. For example, in an epidemic simulation, LPs represent geographical locations containing a subset of the total population. The LP's state represents the diffusion of the disease within the location and the status of the occupants within that location. Events in this simulation represent the arrival or departure of individuals to or from that location, the progression of a diseased individual within that location, the diffusion of a disease within that location, etc. [19]. To effectively model epidemics, a significant population size and number of locations needs to be simulated. With so many LPs, it becomes infeasible to perform this simulation on a sequential machine.

The need for such large simulations has energized research in Parallel Discrete Event Simulation (PDES). Events from separate LPs are executed concurrently by one of N threads. Each LPs' events execute in chronological order to ensure local causality constraints are met [9]. However, PDES is susceptible to other causality errors. Optimistically synchronized simulators are the most susceptible to these causality errors. While conservatively synchronized simulators do not execute events until the system has determined it is safe to do so [9], optimistic approaches, such as the Time Warp protocol, detect rather than prevent causal errors. This does, however, allow for increased concurrency as events are continually executed until a causal error is detected.

One of the most well-known optimistic protocols is the Time Warp mechanism [9]. In addition to the standard DES data structures, each LP in a simulation implementing the Time Warp protocol consists of the following data structures:

- **Unprocessed Queue:** contains events that have been scheduled, but have yet to be executed. This structure acts as the pending event set for the LP.
- **Processed Queue:** records previously executed events.
- **Output Queue:** contains event messages sent to other LPs.

In Time Warp, a causality error occurs if an event message is received containing an event time-stamp smaller than the time-stamp of the previously executed event. Such an event is known as a straggler event. When a straggler event is received by an LP, that LP must undo all effects of all events executed with a time-stamp greater than that of the straggler event, henceforth referred to as a rollback. During a rollback, prematurely executed events are removed from the processed queue and reinserted into the unprocessed queue after the straggler event, For every event message in the output queue with an event time-stamp greater than that of the straggler event, an anti-message is generated. Anti-messages are sent to the associated LP of that event and remove the prematurely generated event from the LP's queue. Figure 4.1 shows the scheduling state of the LP as a straggler event is received. Figure 4.2 shows the scheduling state of the LP after the rollback is processed [7].

While rollbacks are a problem in themselves, rollbacks represent another issue relevant to this study; the need to access the pending event set. When a rollback modifies an LP's local pending event set, the global pending event set must be updated as well. Any access to the global pending event set is a possible point of contention as only one thread can access this structure at a time. The implementation and management of the pending event set is crucial to the overall performance of PDES [21].

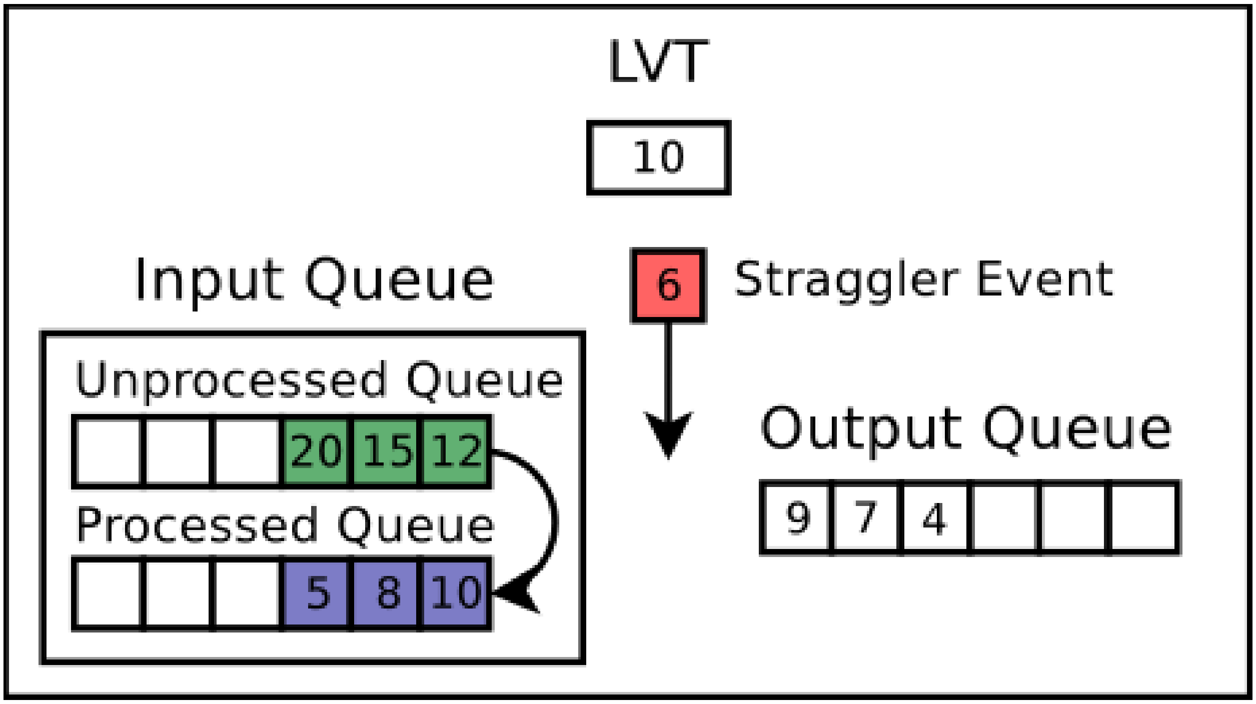

Figure 4.1: **LP at the time of a straggler event is received**

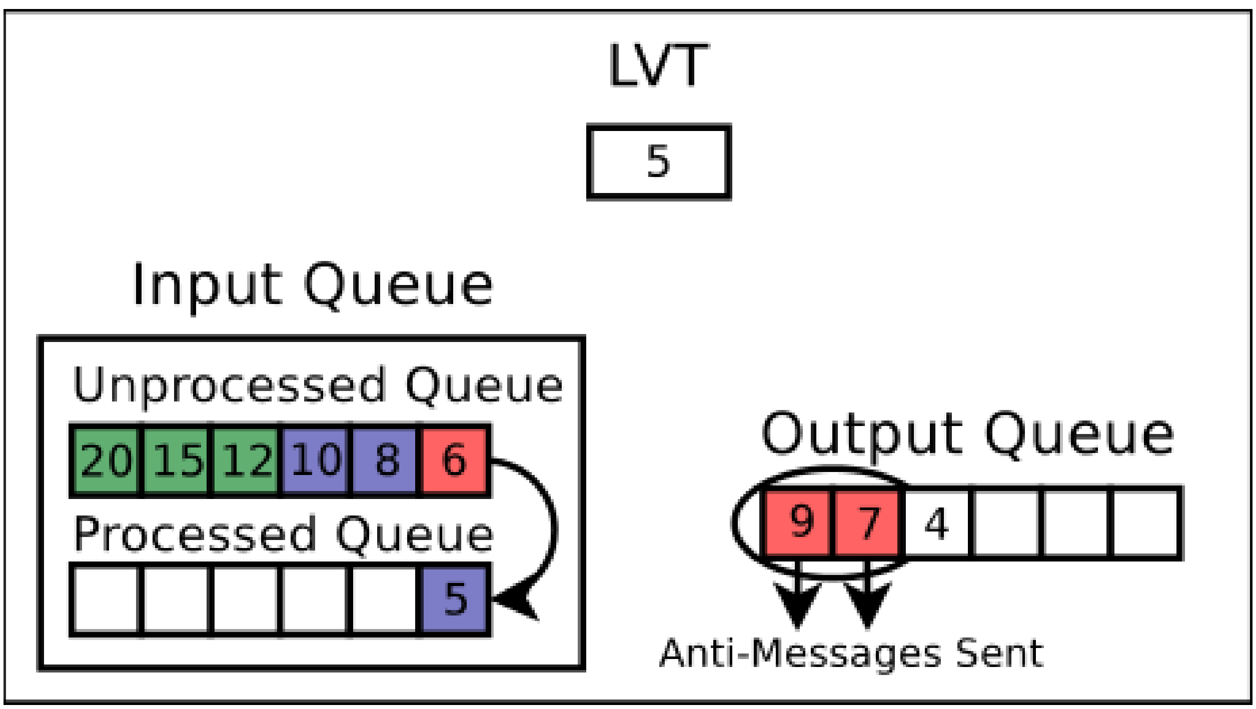

Figure 4.2: **LP after a rollback is processed**

4.2 The WARPED Pending Event Set

WARPED is a publicly available Discrete Event Simulation (DES) kernel implementing the Time Warp protocol [9, 14]. It was recently redesigned for parallel execution on multi-core processing nodes [16]. It has many configuration options and utilizes many different algorithms of the Time Warp protocol [9].

The pending event set is maintained as a two-level structure in WARPED [7]. Each LP maintains its own event set as a time-stamp ordered queue. As previously mentioned, each LP maintains an unprocessed queue for scheduled event to be executed and a processed queue to store processed events. A common Least Time-Stamped First queue is populated with the least time stamped event from each LP's unprocessed queue. As the name suggests, the LTSF queue is automatically sorted in increasing time-stamp order so that worker threads can simply retrieve an event from he head of the queue. This guarantees the worker thread retrieves the least time-stamped event without having to search through the queue, The LTSF queue is also referred to as the schedule queue in WARPED; these terms will be used interchangeably.

4.2.1 Pending Event Set Data Structures

The implementation of the pending event set is a key factor in the performance of the simulation [21]. The WARPED simulation kernel has two fully functional implementations: 1) the C++ Standard Template Library (STL) multi-set data structure, and 2) the splay tree data structure. The way in which these data structures are accessed and, more importantly, self-adjust will be relevant to how effectively TSX can be used to access these structures.

STL Multi-set The STL multi-set data structure , specifically the sorted STL multi-set data structure, is an abstract data structure implemented as a self-balancing, red-black binary search tree [11]. Look-up, insertion, and deletion operations performed in a red-black tree with n elements are performed in average O(log n) time. When insertion or deletion operations are performed, the tree is rebalanced by a tree rearrangement algorithm and a "painting" algorithm taking average O(1) and O(log n) time respectively.

The STL multi-set is a self sorting data structure. The lowest value element will be the left most child node of the tree. To access the least time-stamped event at the head of the LTSF queue, multi-set red-black tree must be traversed to the left most child node. Any insertion or removal of events requires the red-black tree to rebalance itself.

Splay Tree The splay tree is a self-adjusting binary search tree in which recently accessed elements are moved to the root of the tree for quicker access [23]. Look-up, insertion, and deletion operations performed in a splay tee with *n* elements are performed in average O(log n) time. When an element is inserted or looked up, a splaying operation is to move that element to the root of the tree.

When an event is inserted into the LTSF queue, it becomes the root of the splay tree. This is advantageous tf the inserted event is the next least time-stamped event; when a worker thread schedules the next event, it just has to look at the root of the splay tree. Furthermore, there are several functions in WARPED that access the least time-stamped event in the pending event without removing it. This moves the event to the root of the splay tree and results in quick retrieval when the event is finally removed.

4.2.2 Worker Thread Event Execution

A manager thread initiates *n* worker threads at the beginning of the simulation. It can also suspend inactive worker threads if they run out of useful work. When a worker thread is created, or resumes execution after being suspended by the manager thread, it attempts to lock the LTSF queue and dequeue the least time-stamped event. If the worker thread successfully retrieved an event, it executes that event as specified by the simulation model. It then attempts to lock the unprocessed queue for the LP associated with the executed event, and dequeue the next least time-stamped event. The dequeued event is inserted into the LTSF queue, which resorts itself based on the event time-stamps. An abstract event processing algorithm is shown in Figure 4.3 [7]. Note that the worker threads perform many other functions. The entire pending event set implementation can be seen in Figure 4.4 [7].

4.2.3 Contention

Only one worker thread can access the LTSF queue at a time. This creates a clear point of contention during event scheduling as each thread must first retrieve an event from the LTSF queue. The LTSF queue must also be updated when events are inserted into any of the LP pending event sets. This occurs when new events are generated or the simulation encounters a causality error and must rollback.

Contention increases with the number of worker threads used to perform the simulation. The initial

WARPED implementation execution time was measured and analyzed for 1 to 7 worker threads. These results can be seen in Figure 4.5. It is evident that performance begins to plateau once the number of worker threads used surpasses four. This is attributed to the increased contention for the LTSF queue; with more threads, each thread has to wait longer for access to the LTSF queue. The multi-core processor trend continues to increase the number of simultaneous execution threads available, consequently increasing the contention problem.

```
worker_thread()

  lock LTSF queue
  dequeue smallest event from LTSF
  unlock LTSF queue

  while !done loop

    process event (assume from LPi)

    lock LPi queue

    dequeue smallest event from LPi

    lock LTSF queue

    insert event from LPi
    dequeue smallest event from LTSF

    unlock LTSF queue
    unlock LPi queue
  end loop
```

Figure 4.3: **Generalized event execution loop for the worker threads. Many details have been omitted for clarity.**

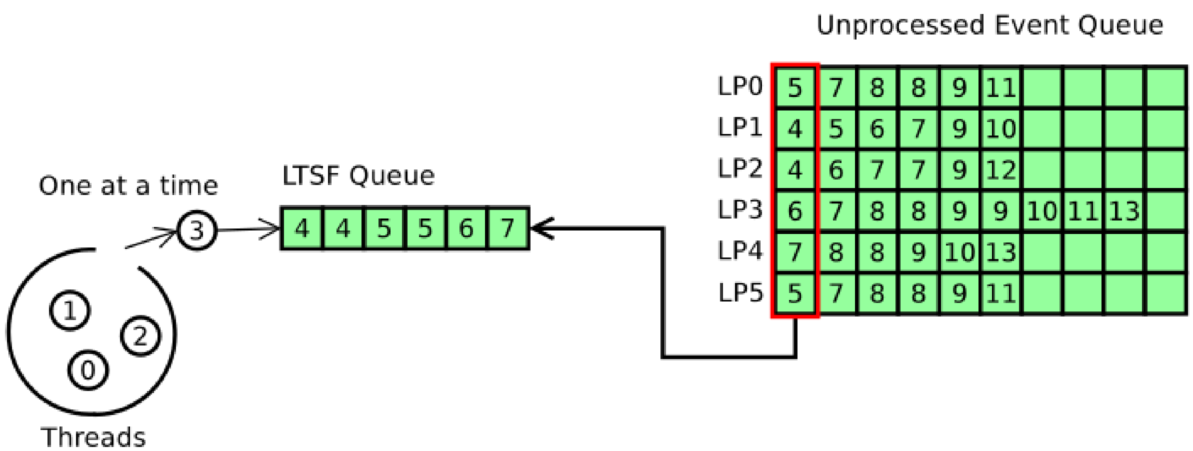

Figure 4.4: **Pending Event Set Scheduling**

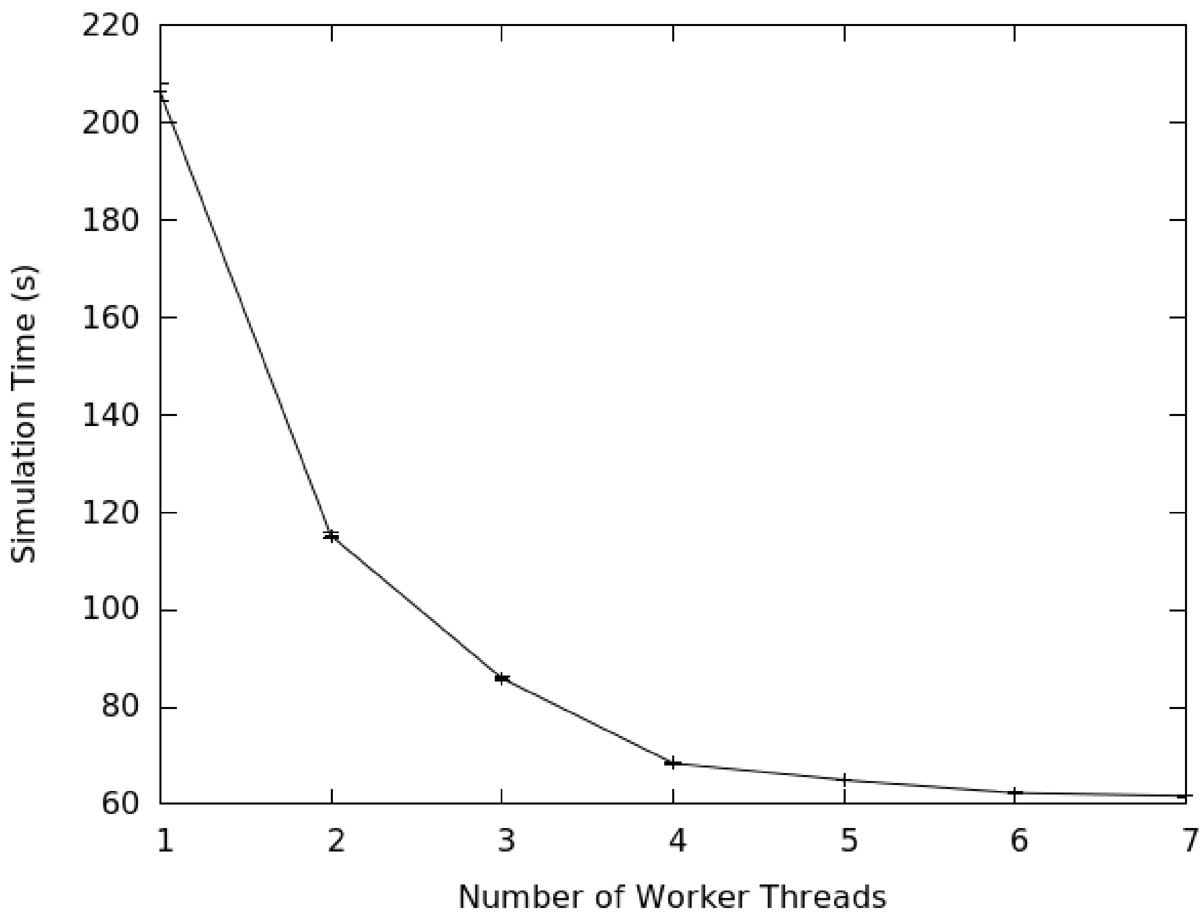

Figure 4.5: **WARPED Simulation Time versus Worker Thread Count for Epidemic Model**

4.3 Previous Solutions to Contention

Dickman *et al* explored the explored the use of various data structures in WARPED pending event set implementation, specifically, the STL multi-set, splay tree, and ladder queue data structures [7]. A secondary focus of this study will expand upon the use of splay tree versus STL multi-set data structures; at the time of this study, the ladder queue implementation was being heavily modified and could not be included in this study.

Another focus of their study was the utilization of multiple LTSF queues [7]. Multiple LTSF queues are created at the beginning of the simulation. Each LP is assigned to a specific LTSF queue as shown in Figure 4.6. In a simulation configured with four LPs, two worker threads, and two LTSF queues, two LPs and one thread are assigned to each queue. This significantly reduced contention as each thread could access separate LTSF queues concurrently. The initial implementation statically assigned LPs to LTSF queues. This resulted in an unbalanced load distribution, leading to an increased number of rollbacks and reduced simulation performance. This was corrected using a load balancing algorithm to dynamically reassign LPs to LTSF queues. This study expands the previous multiple LTSF queue to evaluate if contention can be reduced even further with TSX.

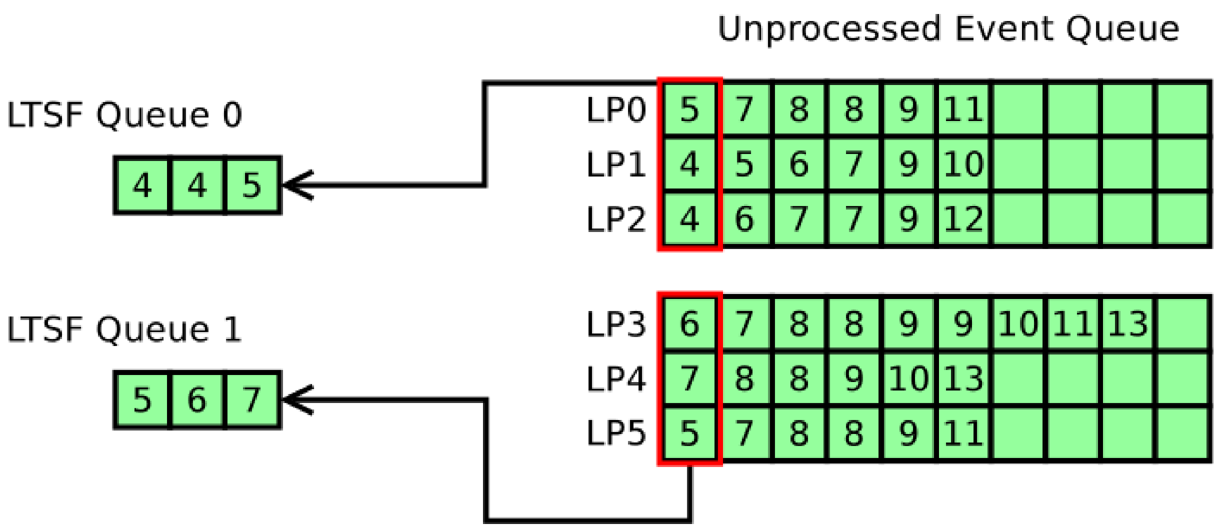

Figure 4.6: **Pending Event Set Scheduling with Multiple LTSF Queues**

4.4 Thread Migration: Another Solution to Contention

Another potential solution to contention is to distribute worker threads that try to simultaneously access the same LTSF queue to different LTSF queues. In the original scheduling scheme, worker threads are assigned to a specific LTSF queue. The worker thread would insert the next event into the same LTSF it had just scheduled from as seen in Figure 4.3. In this implementation, the worker thread inserts the next event into a different LTSF queue, based on a circularly incremented counter. The worker threads are dynamically reassigned to LTSF queues. The number of LTSF queues is specified in a configuration file, and has no restrictions as in the static assignment.

It was discovered that this implementation resulted in poor performance on NUMA architectures. Jingjing

```
worker_thread()

  i = fetch-and-add LTSF queue index
  lock LTSF[i]
  dequeue smallest event from LTSF[i]
  unlock LTSF[i]

  while !done loop

    process event (assume from LPi)

    lock LPi queue

    dequeue smallest event from LPi

    i = fetch-and-add LTSF queue index

    lock LTSF[i]

    insert event from LPi into LTSF[i]
    dequeue smallest event from LTSF[i]

    unlock LTSF queue
    unlock LPi queue
  end loop
```

Figure 4.7: **Generalized event execution loop for migrating worker threads. Many details have been omitted for clarity.**

Wang et al. noticed similar performance degradation, which they attributed to poor memory locality due to the movement of LPs to different threads [26]. To offset these performance hits, a migration count was implemented in this scheme. Instead continuous migration, threads would be statically assigned to one LTSF after executing a certain number of events.

Chapter 5

WARPED with TSX

This section analyzes the various critical sections that use the TSX mechanism. As previously mentioned, the primary focus of this study is the shared LTSF queue. The per LP unprocessed and processed queues also use the TSX mechanism.

5.1 Shared Data Structure Critical Sections

First, it is important to understand the operations performed in a critical section. If a critical section always writes to the entire shared data structure, TSX will most likely not be useful. Functions are only explained in terms of the operations pertaining to the specific data structure they operate on for the sake of clarity.

5.1.1 LTSF Queue Functions

The following functions require synchronization to access the LTSF queue:

- `insert()` - insert an event into the LTSF queue if an event was inserted at the beginning of a specific LP's unprocessed queue.
- `updatedScheduleQueueAfterExecute()` - inserts the dequeued event from a specific LP's unprocessed queue into the LTSF queue.
- `nextEventToBeScheduledTime()` - returns the time of the event at the beginning of the LTSF queue.
- `clearScheduleQueue()` - clears the LTSF queue.

- `setLowestObjectPosition()` - **not clear on this function**
- `peek()` - retrieves the next event for execution.

5.1.2 Unprocessed Queue Functions

The following functions require synchronization to access a specific LPs unprocessed queue:

- `insert()` - insert an event into a specific LP's unprocessed queue.
- `updatedScheduleQueueAfterExecute()` - dequeue the next least time-stamped event from a specific LP's unprocessed queue.
- `getEvent()` - dequeue and return the least time-stamped event in the unprocessed queue; insert event into processed queue.
- `getEventIfStraggler()` - same as `getEvent()` but does not insert the event into the processed queue as the getEvent function above does.
- `peekEvent()` - return a reference to the next event in the LP's unprocessed queue.
- `peekEventCoastForward()` - same as `peekEvent()`.
- `handleAntiMessage()` - delete an event in a specific LP's unprocessed queue for which the LP received an anti-message.
- `ofcPurge()` - removes all events from the unprocessed queue; used for optimistic fossil collection, which is beyond the scope of this study.
- `peekEventLockUnprocessed()` - peek the first event of a specif LP's unprocessed queue, but leave the queue locked.
- `getMinEventTime()` - get the time-stamp of the first event in a specific LP's unprocessed queue.

5.1.3 Processed Queue Functions

The following functions require synchronization to access a specific LPs processed queue:

- `getEvent()` - insert the dequeued event from a specific LP's unprocessed queue into that LP's processed queue.
- `getEventWhileRollback()` - same as getEvent(), except the unprocessed queue is already locked.
- `rollback()` - traverse a specific LP's entire processed queue and remove any events with a time-stamp greater than or equal to the rollback time; the removed events are placed in the LP's unprocessed queue.
- `fossilCollect()` - remove events satisfying a certain criteria from a specific LP's processed queue.
- `ofcPurge()` - same as the ofcPurge() function for the unprocessed queue.

5.2 Shared Data Structure Transactional Regions

The functions described above perform a variety of memory operations, and any thread can execute any critical section at any time. Based on static analysis, there's no way of knowing which threads will access what structure in what way, hence the need for synchronization. But with TSX, functions that do not interfere can execute concurrently. TSX tracks read and write memory operations separately in the transaction's read-set and write-set respectively. Transactions only interfere if a data conflict occurs, *i.e.,* a thread attempts to write to a memory location in another transaction's read-set, or a thread attempts to read a memory location in another transaction's write-set.

For example, one worker thread calls `nextEventToBeScheduleTime` to get the time-stamp of the event at the head of the LTSF queue. There is a possibility that a different worker thread is currently updating the LTSF queue or will attempt to update the LTSF queue with the first worker thread is in the middle of executing `nextEventToBeScheduleTime`. This scenario necessitates synchronization. However, instead of the second worker thread writing to the LTSF queue, it also calls `nextEventToBeScheduleTime`. Both are read operations and do not interfere with each other. TSX recognizes this scenario and allows the worker threads to execute concurrently, whereas locks force one worker thread to wait until the other is done with the LTSF queue.

Several similar scenarios can arise during simulation execution. While there are too many possible scenarios to identify specifically where TSX can be beneficial, the potential to expose concurrency through dynamic synchronization is too great to be dismissed. Note, there is also no guarantee that TSX will work 100% of the time; there are several run-time events that can cause transactions to abort, as well as physical limitations.

The process of scheduling requires a significant amount of write operations to the queues listed above. As long as executing threads do not write to an entire queue, there is a good chance that the write operations will not interfere. TSX dynamically determines if the write operations are performed on different memory locations and allows the threads to execute concurrently. The data structures used to implement the respective queues is a significant factor in determining if two or more threads perform conflicting memory operations on the same structure. Not only is the performance of simulation dependent on the pending event set implementation, but the performance of TSX is also dependent on the data structures implementing the pending event set.

For example, a worker thread is scheduling the next least time-stamped event from the LTSF queue and needs to remove that event from the queue. In the STL multi-set schedule queue implementation, the tree must be traversed to the lowest value element, which adds all tree nodes in that path to the transactions read-set. The node is removed, but the STL multi-set must go through the process of rebalancing itself before the transaction ends. Before the multi-set can complete the rebalancing process, another worker thread attempts to schedule the next least time-stamped event. It traverse the tree to the lowest value element and removs it. This involves a write operation to the parent node of the event. But the node was already addded to the first transaction's read-set. A data conflict results and both transactions must abort.

In the splay tree schedule queue implementation, the next least time-stamped event is already the root of the splay tree, either because it was just peeked at or inserted. A worker thread enters the transction to schedule the event and removes it from the queue. The parent of the removed node then has to be splayed to the root of the splay tree

. Before that process is completed, another

thread enters the transactional region to schedule the next least time-stamped event.

Both of these data structures are some form of binary tree, meaning that few nodes are accessed for certain operations. This is advantageous for TSX as the memory locations will not be added to the transaction read-sets thus making it less likely for data conflicts to occur. It is possible, though highly unlikely, that either implementation could take the form of a single linked list, depending on what order events are inserted in. For instance, events inserted into the multi-set red-black tree in increasing chronological order create a singly linked list. This situation poses a threat to TSX if the entire list is traversed in a transaction. Not only because access to any element becomes a possible data conflict, but also because of size limitations. If the LTSF queue contains too many events, TSX might not be able to track all the memory locations involved in the queue, thus resuling in aborts.

5.3 TSX Implementation

This section discusses how both TSX interfaces were implemented in WARPED.

5.3.1 Hardware Lock Elision (HLE)

The generic algorithm presented in Figure 2.1 in Section 2.3.1 only works for locks with a binary value, *i.e.,* the lock is free or not free. The WARPED locking mechanism assigns the thread number to the lock value to indicate which thread currently holds the lock. To comply with this implementation, custom HLE lock acquire and lock release functions were implemented. GCC inline assembly functions were developed appending the appropriate HLE prefixes to a lock cmpxchg instruction.

5.3.2 Restricted Transactional Memory (RTM)

As previously explained in Section 2.3.2, RTM allows the programmer to specify an abort path to be executed upon a transactional abort. This allows better tuning of RTM performance. The RTM algorithm implemented in WARPED includes a retry algorithm described below in Figure 5.1. Instead of immediately retrying transactional execution, the algorithm decides when and if the transaction should be retried based on the condition of the abort. If the transaction was explicitly aborted for reasons other than another thread owning the lock, do not retry transactional execution. The programmer used the `_xabort()` function to explicitly abort the transaction. If the lock was not free upon entering the transaction, wait until it is free to retry transactional execution. If a data conflict occurred, wait an arbitrary amount of time before retrying. This offsets the execution of the conflicting threads in hopes that the conflicting memory operations will be performed at different times on the next retry.

The RTM retry limit is specified at compile time. Each data structure maintains its own retry limit initially set to the global limit. A backoff algorithm is used to reduce the retry limit for a specific data structure. If the transactions for this data structure abort more often than not, the retry limit is reduced.

Furthermore, if transactions for the data structure consistently abort for an extended period of time with no successfull commits, transactional execution is not attempted for the remainder of the simulation.

```
while retry count is less than retry limit
    status = _xbegin()

    if status == XBEGIN
        if lock is free
            execute transactional region
        else
            _xabort

    update abort stats

    if transaction will not succeed on retry or
        _xabort was called due to reasons other than the lock not being free

        break

    else if _xabort was used because the lock was not free

        wait until the lock becomes free to retry

    else if a data conflict occurred

        wait an arbitrary amount of time before retrying

    increment retry count
end loop

acquire lock

execute critical section
```

Figure 5.1: **RTM Retry Algorithm**

Chapter 6

Experimental Analysis

This study compares the performance of the WARPED simulation kernel using conventional synchronization mechanisms, Hardware Lock Elision, and Restricted Transactional memory. All simulations were performed on a system with an Intel i7-4770 running at 3.4 GHz with 32GB of RAM. Average execution time and standard deviation were calculated from a set of 10 trials for each simulation configuration.

The simulation model used to obtain the following results is an epidemic model. It consists of 110998 geographically distributed people in 119 separate locations requiring a total of 119 LPs. The epidemic is modeled by reaction processes to model progression of the disease within an individual entity, and diffusion processes to model transmission of the disease among individual entities.

6.1 Static Thread Assignment

In the original WARPED implementation, threads are statically assigned to an LTSF queue. Contention will clearly be a problem if the simulation only schedules from one LTSF queue as every worker thread is assigned to that queue.

The first part of this study compares the performance of the WARPED pending event set static thread scheduling implementation using one LTSF queue synchronized with : 1) atomic locks, 2) HLE, and 3) RTM with 1 retry, 4) RTM with 9 retries, and 5) RTM with 19 retries. These results are shown in Figure 6.1.

It is evident from Figure 6.1 that contention is increasing as the number of worker threads increases. Furthermore, it would appear that utilizing more than one LTSF queue is no longer producing consistent or desirable results. It was also observed that the number of rollbacks was non-trivial for these simulations.

These poor performance could be attributed to recent changes made to the WARPED kernel.

Varying schedule queue countdData is limited for static thread migration as the number of LTSF queues has to be evenly divisible by the number of worker threads. It should be noted that the static assignment of worker threads to LTSF queues only allows for a number of worker threads that is evenly divisible by the number of worker threads due to the way in which LP's are partitioned to LTSF queues.

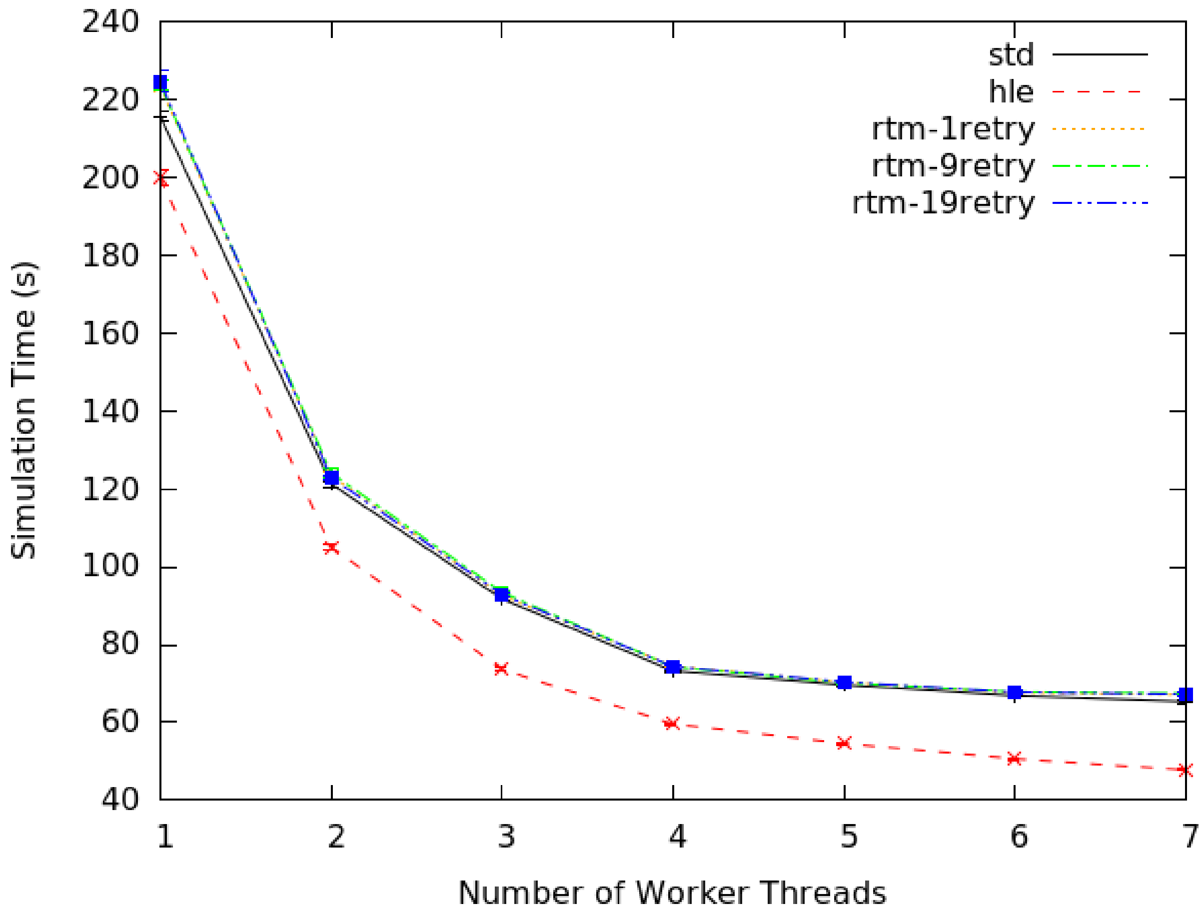

Figure 6.1: **Simulation Time versus Number of Worker Threads 1 STL Multi-set LTSF Queue**

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 9-retry	RTM-19retry
1	121.2255	105.0569	123.4132	124.0336	122.8344
2	118.1558	101.1242	120.085	137.5637	153.0293

Table 6.1: **Simulation Times for 2 Worker Threads with X LTSF Queues**

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 4-retry	RTM-9retry
1	91.68472	73.58687	92.58229	93.42086	92.83141
3	89.50474	70.24289	89.48378	105.8095	125.8599

Table 6.2: **Simulation Times for 3 Worker Threads with X LTSF Queues**

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 4-retry	RTM-9retry
1	69.62144	54.59489	70.33108	70.1835	70.42886
5	93.21429	76.85048	93.23045	84.19164	76.23965

Table 6.3: **Simulation Times for 5 Worker Threads with X LTSF Queues**

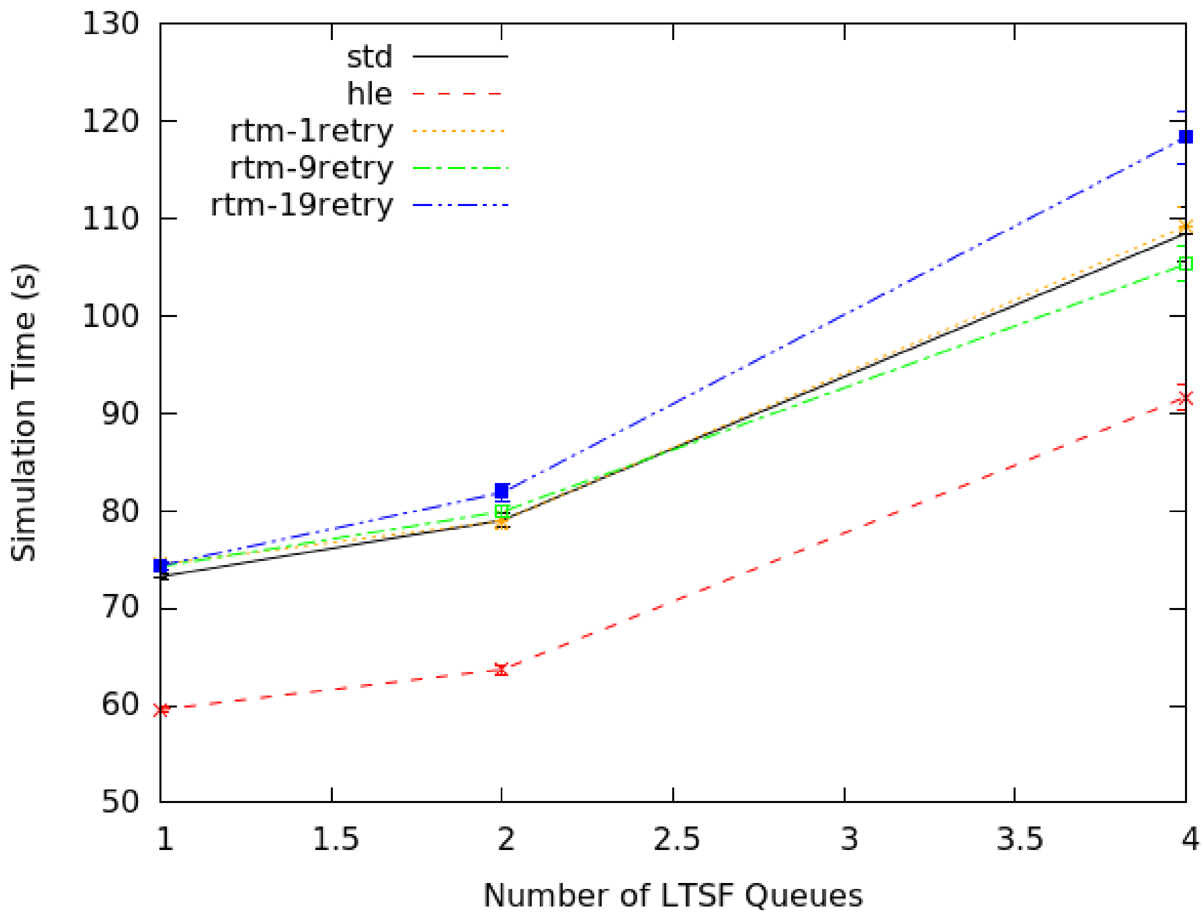

Figure 6.2: **Simulation Time versus Number of STL Multi-set LTSF Queues for 4 Worker Threads**

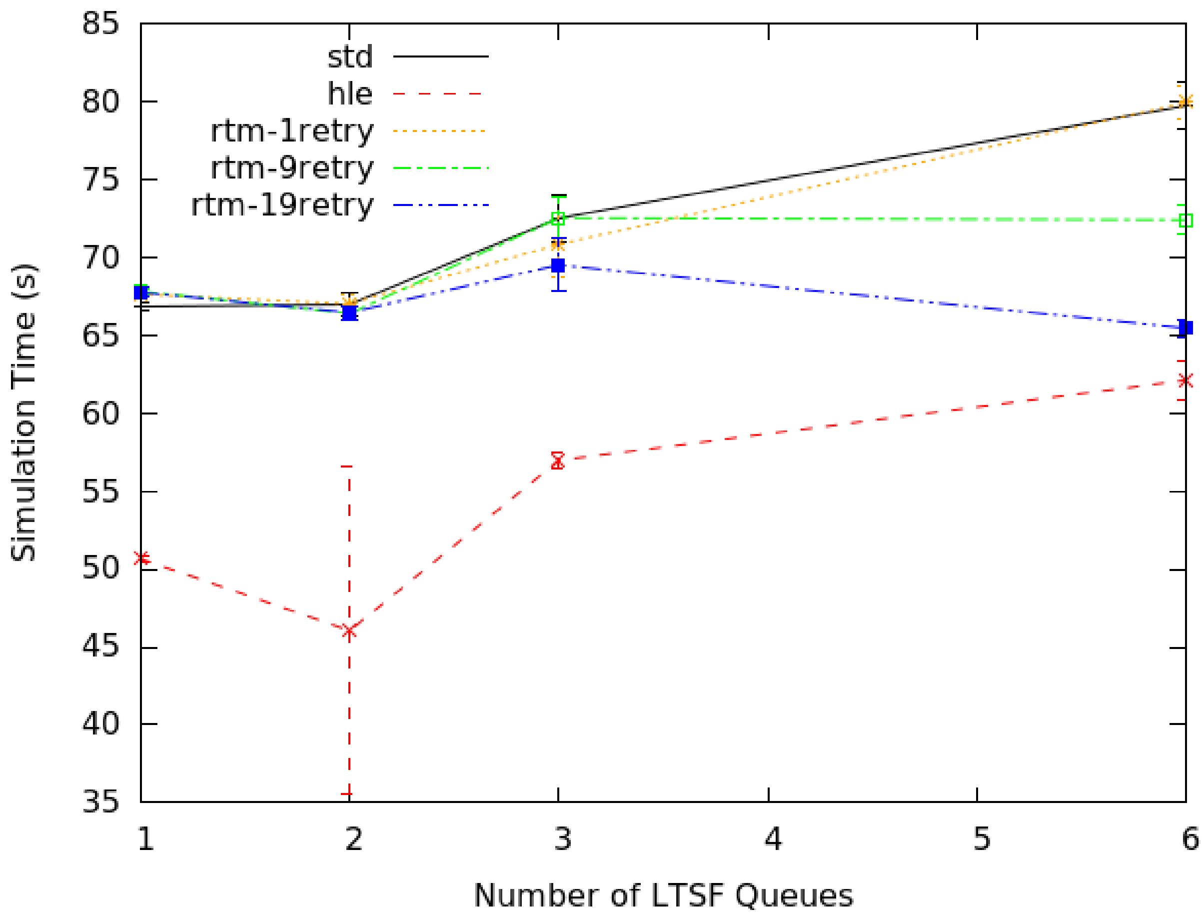

Figure 6.3: **Simulation Time versus Number of STL Multi-set LTSF Queues for 6 Worker Threads**

6.2 Dynamic Thread Assignment

6.2.1 Continuous Thread Migration

6.2.2 Event Limited Thread Migration

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 9-retry	RTM-19retry
1	121.4532	105.2221	123.2083	124.1073	124.1445
2	122.5693	105.7831	123.6679	129.8969	134.2494

Table 6.4: **Simulation Times for 2 Worker Threads with X LTSF Queues**

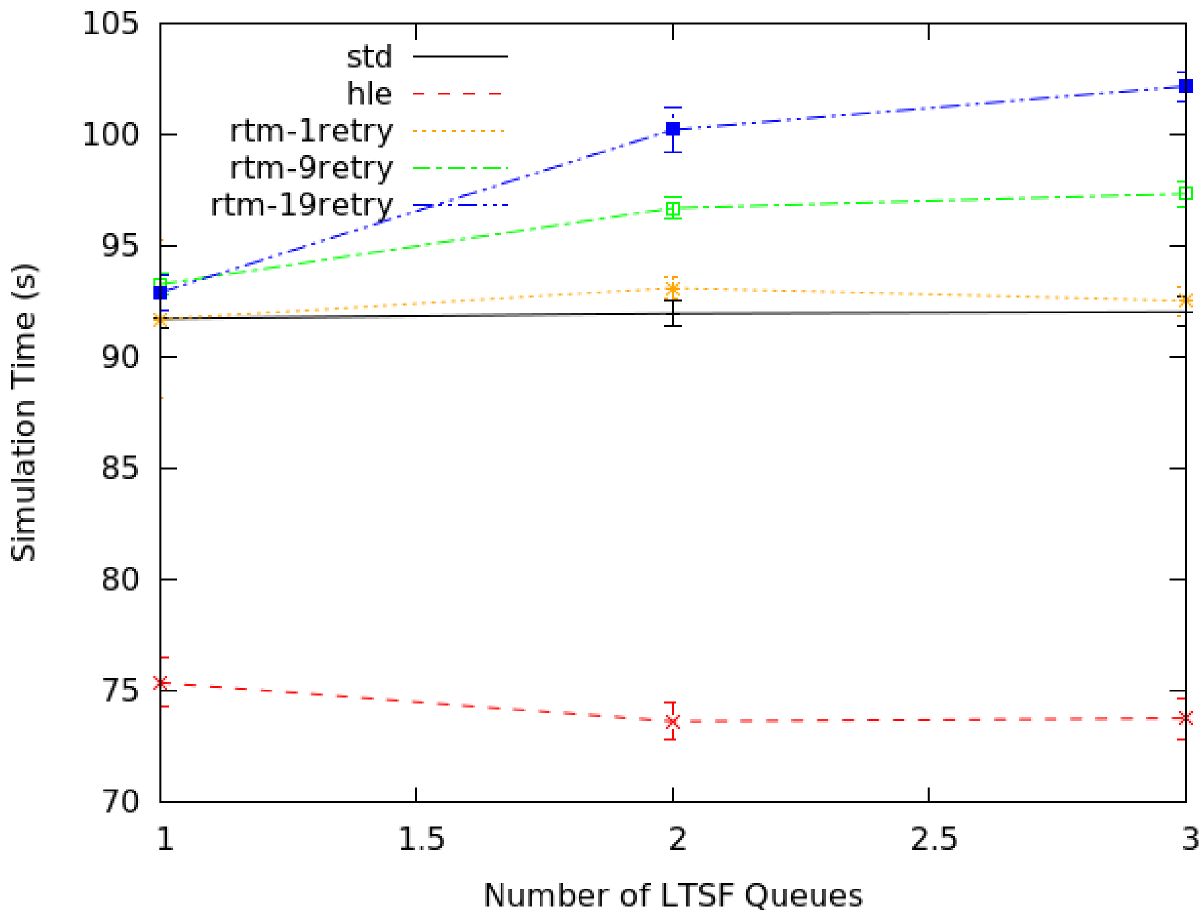

Figure 6.4: **Simulation Time versus Number of STL Multi-set LTSF Queues for 3 Worker Threads**

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 9-retry	RTM-19retry
1	121.2448	105.6107	124.1983	123.6149	123.0078
2	117.6654	102.0753	121.0667	137.9824	154.3929

Table 6.5: **Simulation Times for 2 Worker Threads with X LTSF Queues**

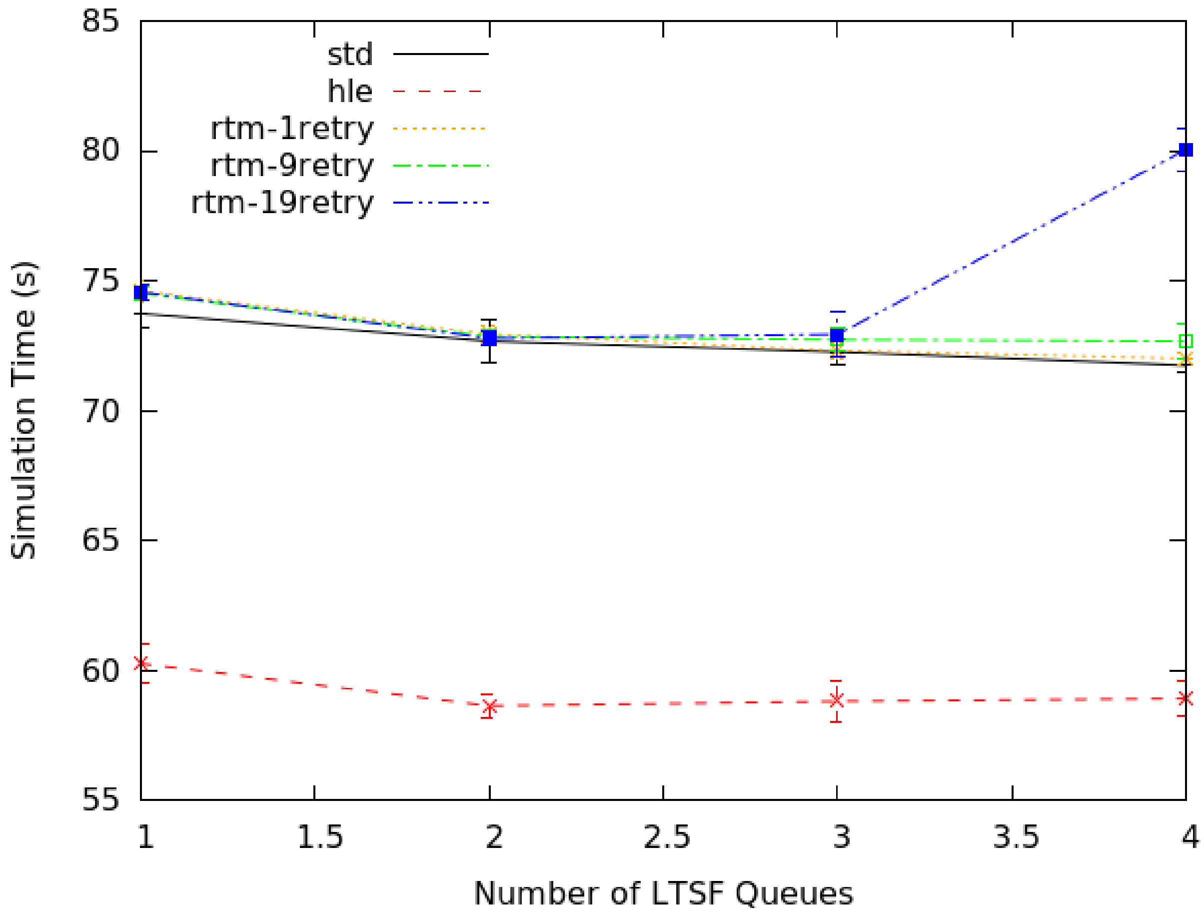

Figure 6.5: **Simulation Time versus Number of STL Multi-set LTSF Queues for 4 Worker Threads**

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 9-retry	RTM-19retry
1	92.36592	74.22306	93.23531	92.67724	92.6402
3	88.91337	70.67712	89.53974	107.369	126.557

Table 6.6: **Simulation Times for 3 Worker Threads with X LTSF Queues**

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 9-retry	RTM-19retry
1	69.30301	54.40232	70.85832	70.46488	70.4784
5	95.31338	77.55966	93.71796	85.63724	76.2384

Table 6.7: **Simulation Times for 5 Worker Threads with X LTSF Queues**

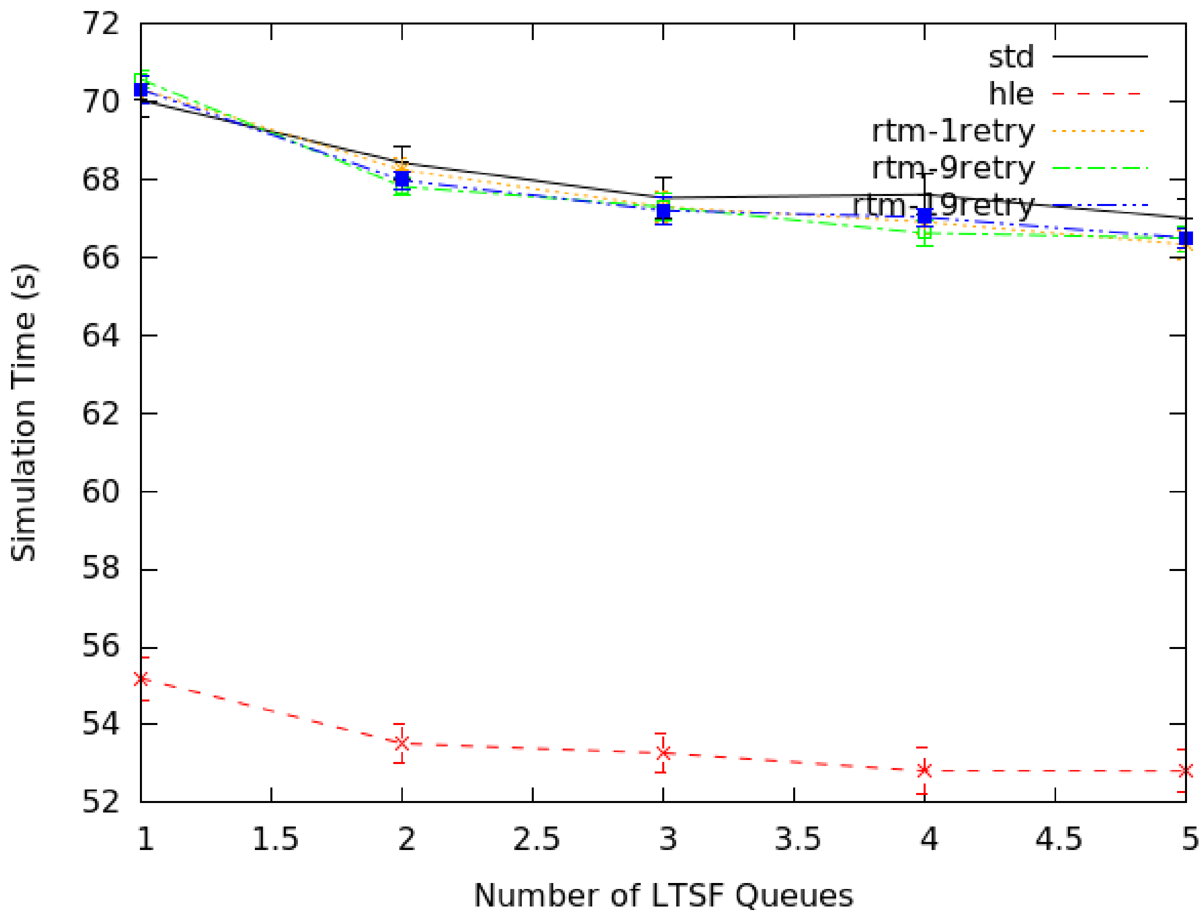

Figure 6.6: **Simulation Time versus Number of STL Multi-set LTSF Queues for 5 Worker Threads**

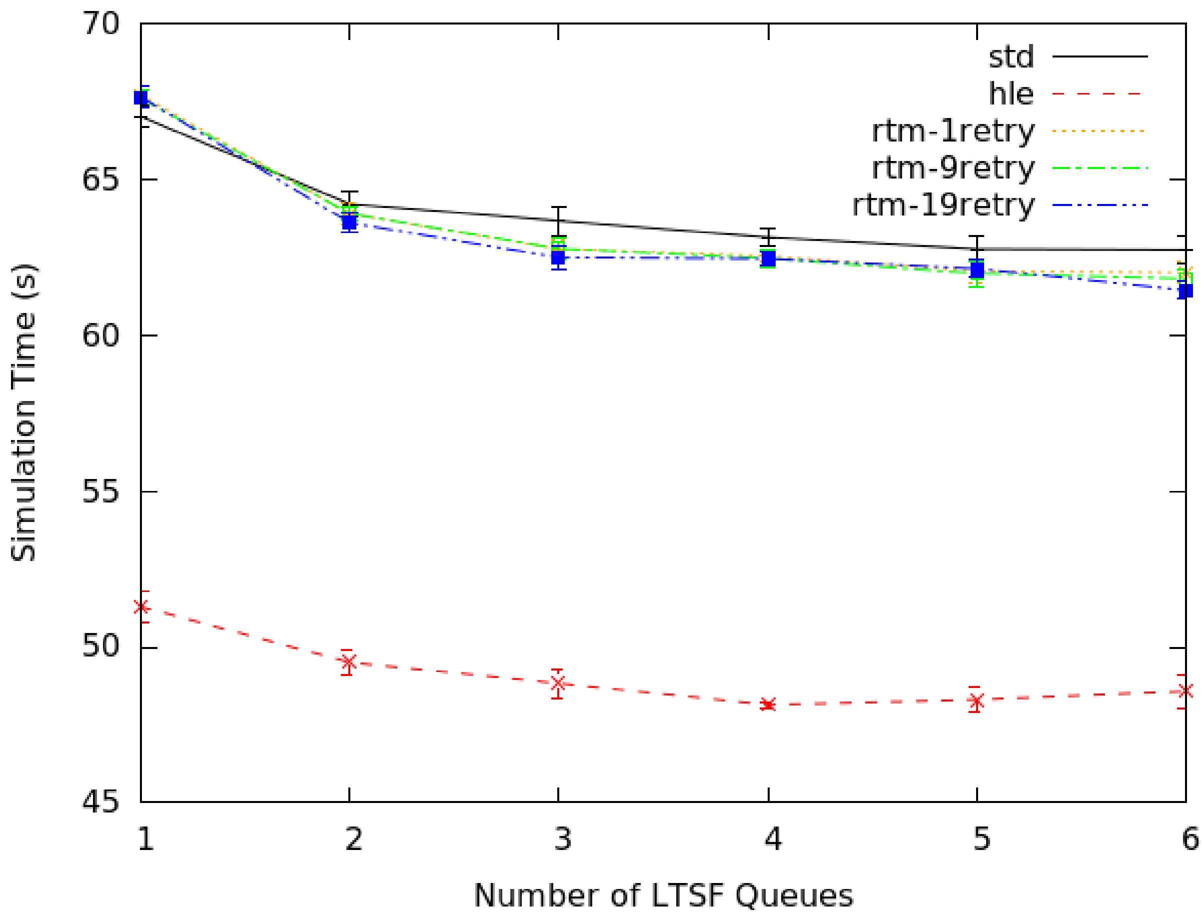

Figure 6.7: **Simulation Time versus Number of STL Multi-set LTSF Queues for 6 Worker Threads**

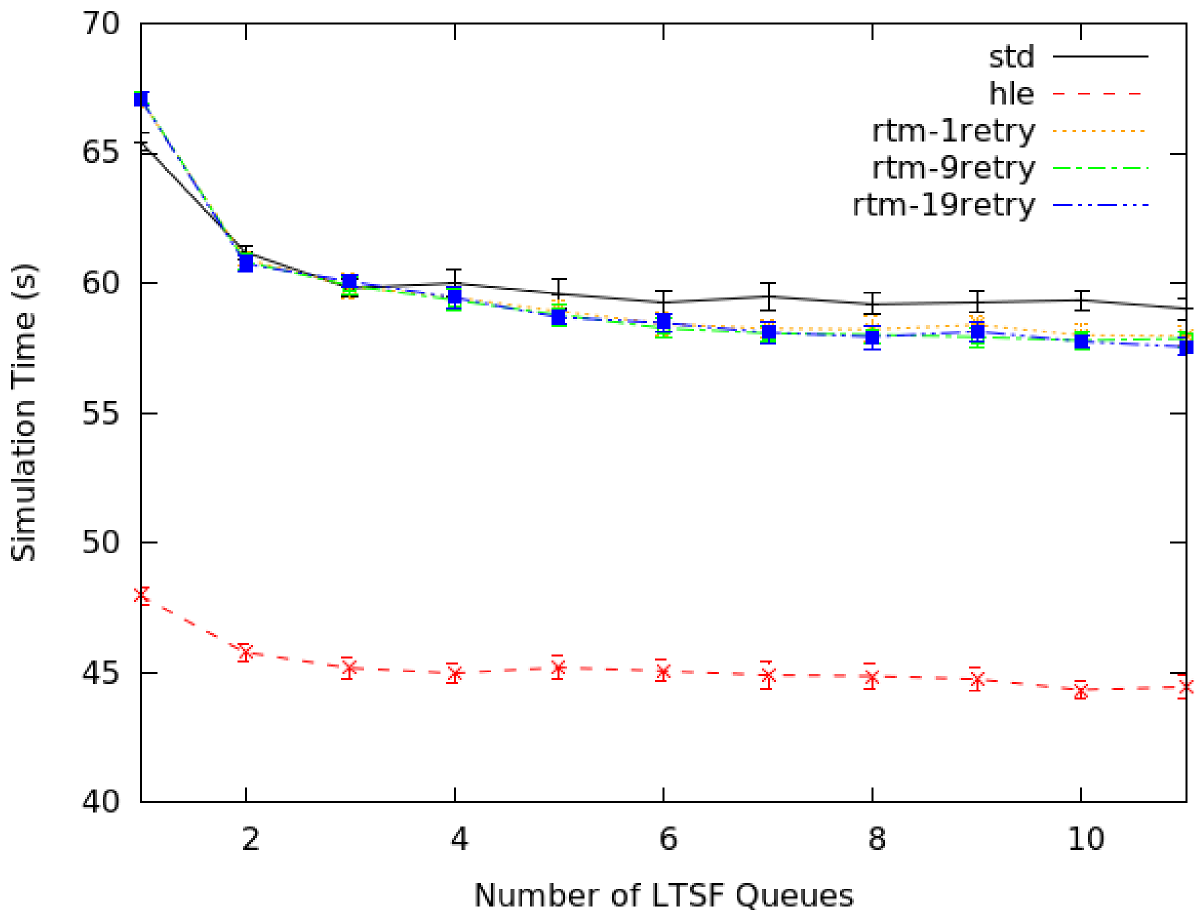

Figure 6.8: **Simulation Time versus Number of STL Multi-set LTSF Queues for 7 Worker Threads**

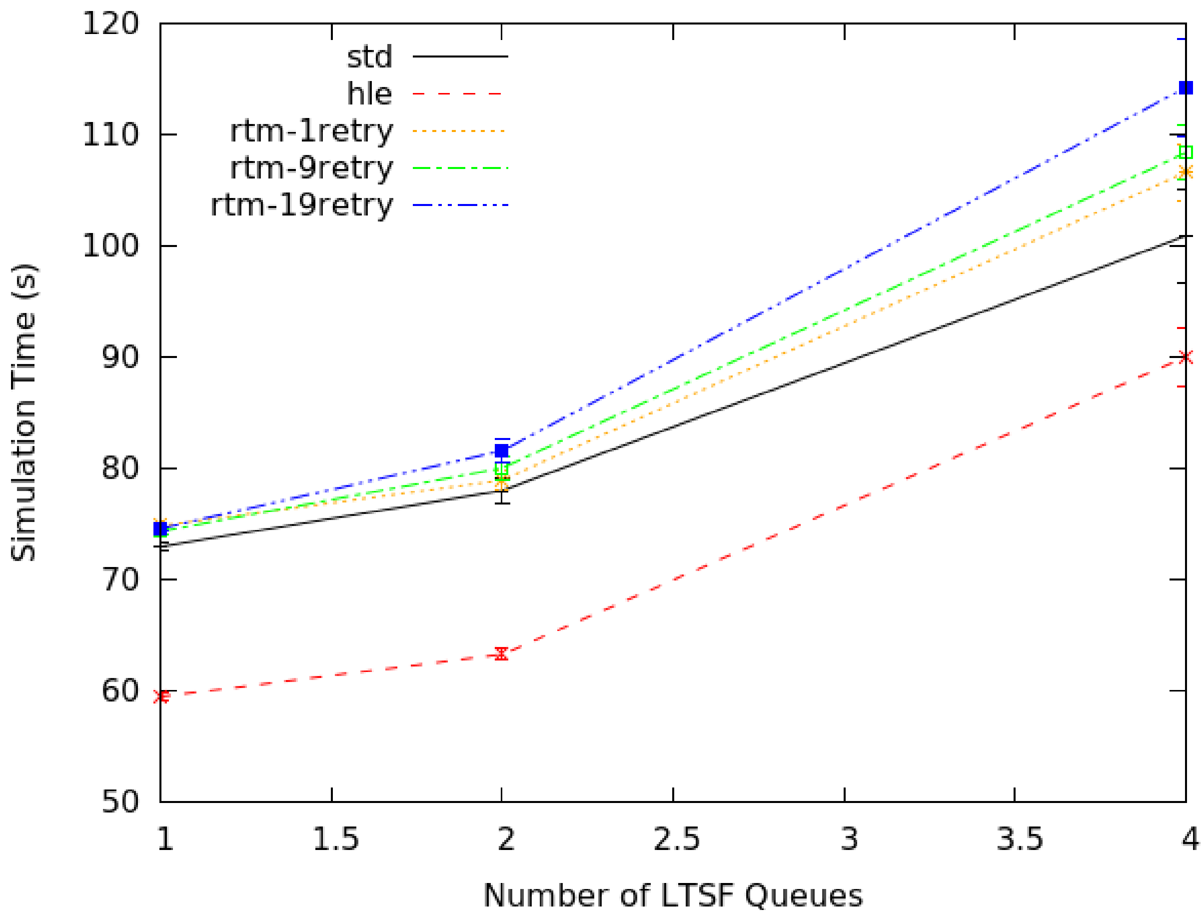

Figure 6.9: **Simulation Time versus Number of STL Multi-set LTSF Queues for 4 Worker Threads**

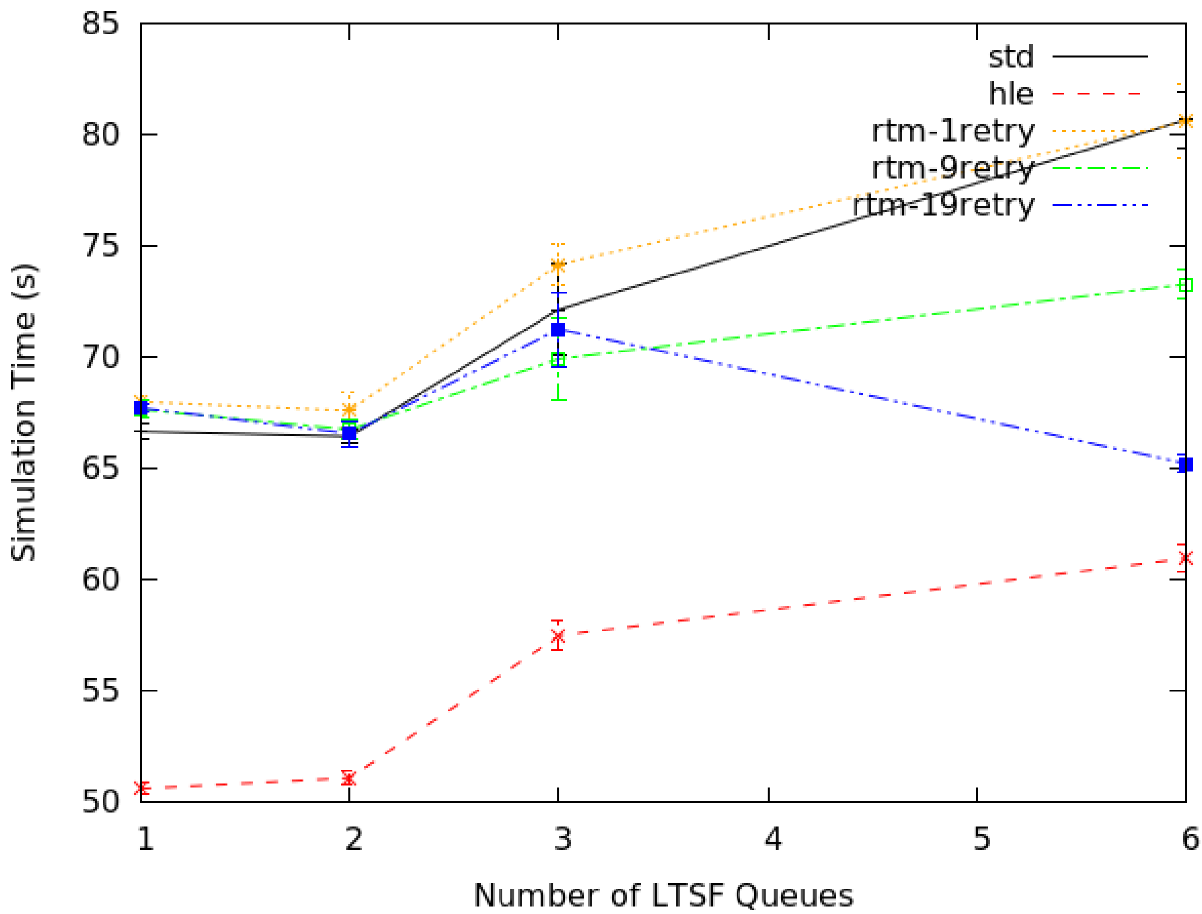

Figure 6.10: **Simulation Time versus Number of STL Multi-set LTSF Queues for 6 Worker Threads**

Chapter 7

Conclusions

Bibliography

[1] *Intel Architecture Instruction Set Extensions Programmer Reference. Chapter 8: Intel Transactional Synchronization Extensions*, Feb. 2012.

[2] *Intel 64 and IA-32 Architectures Optimization Reference Manual. Chapter 12: Intel TSX Recommendations*, July 2013.

[3] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? In *Communications of the ACM - Remembering Jim Gray*, volume 51, pages 40–46, 2008.

[4] V. Chitters, A. Midvidy, and J. Tsui. Reducing synchronization overhead using hardware transactional memory, 2013.

[5] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. Asf: Amd64 extension for lock-free data structures and transactional memory. In *MICRO*, pages 39–50, 2010.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.

[7] T. Dickman, S. Gupta, and P. A. Wilsey. Event pool structures for pdes on many-core beowulf clusters. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 103–114, 2013.

[8] T. J. Dickman. Event list organization and management on the nodes of a many-core beowulf cluster. Master's thesis, University of Cincinnati, 2013.

[9] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[10] V. Gajinov, F. Zyulkyarov, U. Osman S, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Quaketm: Parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*, pages 126–135, 2009.

[11] S. Hanke. The performance of concurrent red-black tree algorithms. Technical report, Institut fur Informatik, Universitat Freiburg, 1998.

[12] T. Harris, J. R. Laurus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2nd edition, 2010.

[13] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):405–425, July 1985.

[14] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. Warped: A time warp simulation kernel for analysis and application development. In H. El-Rewini and B. D. Shriver, editors, *29th Hawaii International Conference on System Sciences (HICSS-29)*, volume Volume I, pages 383–386, Jan. 1996.

[15] J. Misra. Distributed discrete-event simulation, 1986.

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

[17] M. Neuling. What's the deal with hardware transactional memory?, 2014.

[18] T. M. A. Overview. Oliver schwahn, 2011.

[19] K. S. Perumalla and S. K. Seal. Discrete event modeling and massively parallel execution of epidemic outbreak phenomena. *Simulation*, 88, 2012.

[20] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, pages 294–305, 2001.

[21] R. Ronngren, R. Ayani, R. M. Fujimoto, and S. R. Das. Efficient implementation of event sets in time warp. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, pages 101–108, 1993.

[22] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley and Sons, Inc., 8th edition, 2009.

[23] D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. *Journal of the ACM*, 32(3):652–686.

[24] R. M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. Free Software Foundation, Inc., 2013.

[25] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gener/q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT-12)*, pages 127–136, 2012.

[26] J. Wang, N. Abu-Ghazaleh, and D. Ponomarev. Interference resilient pdes on multi-core systems: Towards proportional slowdown. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 115–126, 2013.

[27] Z. Wang, H. Qian, H. Chen, and J. Li. Opportunities and pitfalls of multi-core scaling using hardware transactional memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, 2013.

[28] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.