Experiments with Hardware-based Transactional Memory in Parallel Simulation

Joshua Hay

hayja@mail.uc.edu

(513) 607-4929

June 7, 2014

Abstract

Acknowledgements

Contents

List of Figures

List of Tables

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 [20]. For example, a multi-threaded application 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 grid. The critical section reads a single element, performs a calculation, and updates the element. Once a thread enters the critical section, it locks all other threads out of the entire grid until it has completed its task, thus forcing the threads to essentially execute sequentially upon entering the critical section. 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 [18]; this approach requires the programmer to maintain a separate lock for each row or each element respectively. Unfortunately, programmers are limited to using static information to determine when threads must execute a critical section sequentially regardless of whether coarse-grained or fine-grained locking is used.

However, untapped concurrency could be exposed if the determination to execute sequentially was made dynamically [1]. 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. However, transactional memory systems can dynamically determine when such a scenario arises and allow both threads to execute concurrently.

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 safely executed concurrently [18]. In the previous example, the programmer identifies the critical section as a

transactional region; these terms will be used interchangeably. As threads enter the transactional region, they attempt to transactionally execute the critical section concurrently. 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 safely execute the critical section concurrently by the TM system. Transactional memory is analogous to traffic roundabouts whereas conventional sycnhronization mechanisms are analogous to conventional traffice lights [15].

Transactional memory operates on the same principles as database transactions [11]. The processor atomically commits *all* memory operations of a successful transaction or discards *all* memory operations if the transaction should fail. In order for a transaction to execute successfully, it must be executed in isolation, i.e. without conflicting with other tranasctions/threads memory operations. This is the key principle that allows transactional memory to expose untapped concurrenty 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 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 a logical process are executed concurrently. To further exploit concurrency, optimistic PDES aggressively schedules events instead of strictly enforcing causal ordering of event execution [9, 8], meaning events will continue to be scheduled regardless of order until an error is *detected*. More importantly, the events must be retrieved from a global 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, using the optimistic Time Warp synchronization protocol. 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, specifically, WARPED: a Time Warp synchronized Parallel Discrete Event Simulation application.

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 nameing 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. However, two other

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

Chapter 6 discusses the accomplishments of this research. It also briefly discusses some areas of future research.

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 critical section concurrently [18]. The programmer identifies a transactional region, typically a critical section. As threads enter the transactional region, the underlying processor core attempts to execute transactionally. As it does so, it tracks the memory accesses within the transactioanl region to determine whether or not two or more threads 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 safely execute concurrently. If they do conflict, the process must abort the transaction and execute the critical section non-transactionally, i.e. sequentially with conventional synchronization mechanisms.

As previously mentioned, a transaction builds a set of memory addresses it has read from and a set of memory addresses it has written to, referred to as the read-set and write-set respectively [1] as it executes. Note that These memory operations are buffered until the transaction completes. If the memory operations within the local transaction do not conflict with any memory operations within any other thread's execution path, the transaction can safely complete execution. Upon completion, the transaction will atomically commit all of the buffered memory operations, henceforth referred to simply as a commit.

However, if any memory operation within the local transaction happens to conflict with any memory operation within any other thread's execution path, the transaction cannot safely continue execution. This is referred to as a data conflict and only occurs if: 1) another thread attempts to read a location that is part of the local transaction's write-set, or 2) another thread attempts to read a location that is part of the local transaction's write-set [1]. Once a data conflict is detected, the transactions will abort execution, henceforth referred to simply

as an abort.

Revisiting the example from before, the programmer uses transactional memory synchronization mechanisms to access the 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.

EXAMPLE OF ABORT

TM operates on the principles of database transactions. A transaction is a series of actions with four key attributes: 1) atomicity, 2) consistency, 3) isolation, and 4) durability [11]. The two key attributes for TM are atomicity and isolation; consistency and durability must hold for all multi-threaded operations in multi-threaded applications. If atomicity and isolation be can guaranteed for all memory operations performed within a critical section, that "critical section" can be executed concurrently [18].

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 commited, 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; another property of transactions is 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 as all memory operations must be commited or none can be commited.

2.2 Related Studies

There have been many implementations of TM systems since its conception, mostly in software. Software Transactional Memory (STM) offers better portability but at the cost of performance. Gajinov et al. performed a

study with STM by developing a parallel version of the Quake multiplayer game server from the ground up useing OpenMP parallelizations pragmas and atomic blocks [10]. Their results showed that the 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 reasons discussed above.

Hardware Transactional Memory (HTM) provides the physical resources necessary to implement transacational 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 [22]. 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 news regarding future releases and Sun's Rock processor was cancelled after Sun was acquired by Oracle.

With the release of Intel's Haswell generation processors, Intel's Transactional Synchronization Extensions (TSX) is the most widely commercially available HTM system. Numerous studies have already been done 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 Transactioanl 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 [23]. 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 [24]. 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 Extenions (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 the locking instruction to mark the beginning of a transaction. XRELEASE is placed before the 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. Specifcally, 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 transaction aborts and explicitly writes 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 for the application [1].

A general implementation for the HLE software interface is shown in Figure 1. All the programmer has to do is add the HLE memory model parameters in the locking function intrinsics. GCC 4.8 and above includes

support for the __ATOMIC_HLE_ACQUIRE and __ATOMIC_HLE_RELEASE memory models [21].

```
/* 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 1: Generic HLE Software Interface

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 even read 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 atomically 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 parth 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. 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 ultimiately uses explicit synchronization and guarantees forward progress. GCC 4.8 and above includes support for the _xbegin, _xabort, and _xend intrinsics [21].

```
/* 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: Generic RTM Software Interface

While RTM is a much more flexible interface than HLE, it can only be used on supported Haswell platforms. If a legacy device attemps 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].

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.

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 becase 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.

A set of benchmarks was developed to access a shared array of custom structures. Each structure is allocated to occupy an entire cache line, i.e. 64 bytes, and aligned to the nearest cache line boundary using the GCC align attribute. This ensures that memory is optimally organized as previously mentioned. Benchmarks were run with varying transaction sizes and number of threads.

The first benchmark was run with a single thread to avoid data conflicts. Figure 3 shows the abort rate, i.e. # aborts / # operations, as the number of cache lines accessed within the transaction increases. The read-set data points respresent strictly read operations while the write-set data points represent strictly write opreations.

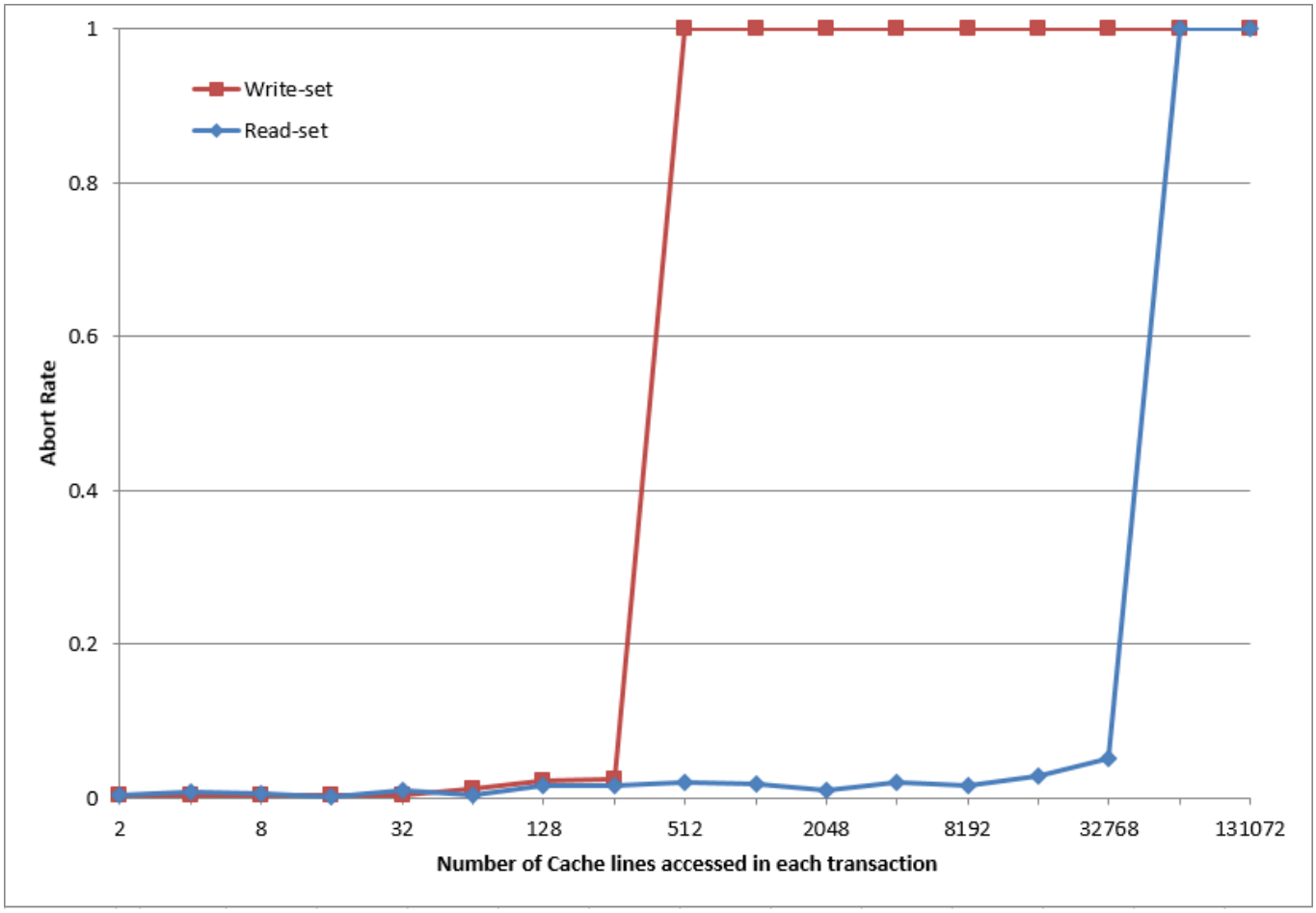

Figure 3: TSX RTM abort rate versus cache lines accessed for a single thread on one core

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. Furthermore, the cache is split into 64 8-way sets; if memory is not organized properly, the total write-set size will be reduced.

However, 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 tranasctional abort, evicition 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 second benchmark is run with two threads on the same core to evaluate the effects of hyper-threading on TSX. Each thread accesses the same number of cache lines, but at different memory locations to prevent any data conflicts. Figure 4 shows the abort rate for one of the threads as the number of cache lines accessed within each transaction increases.

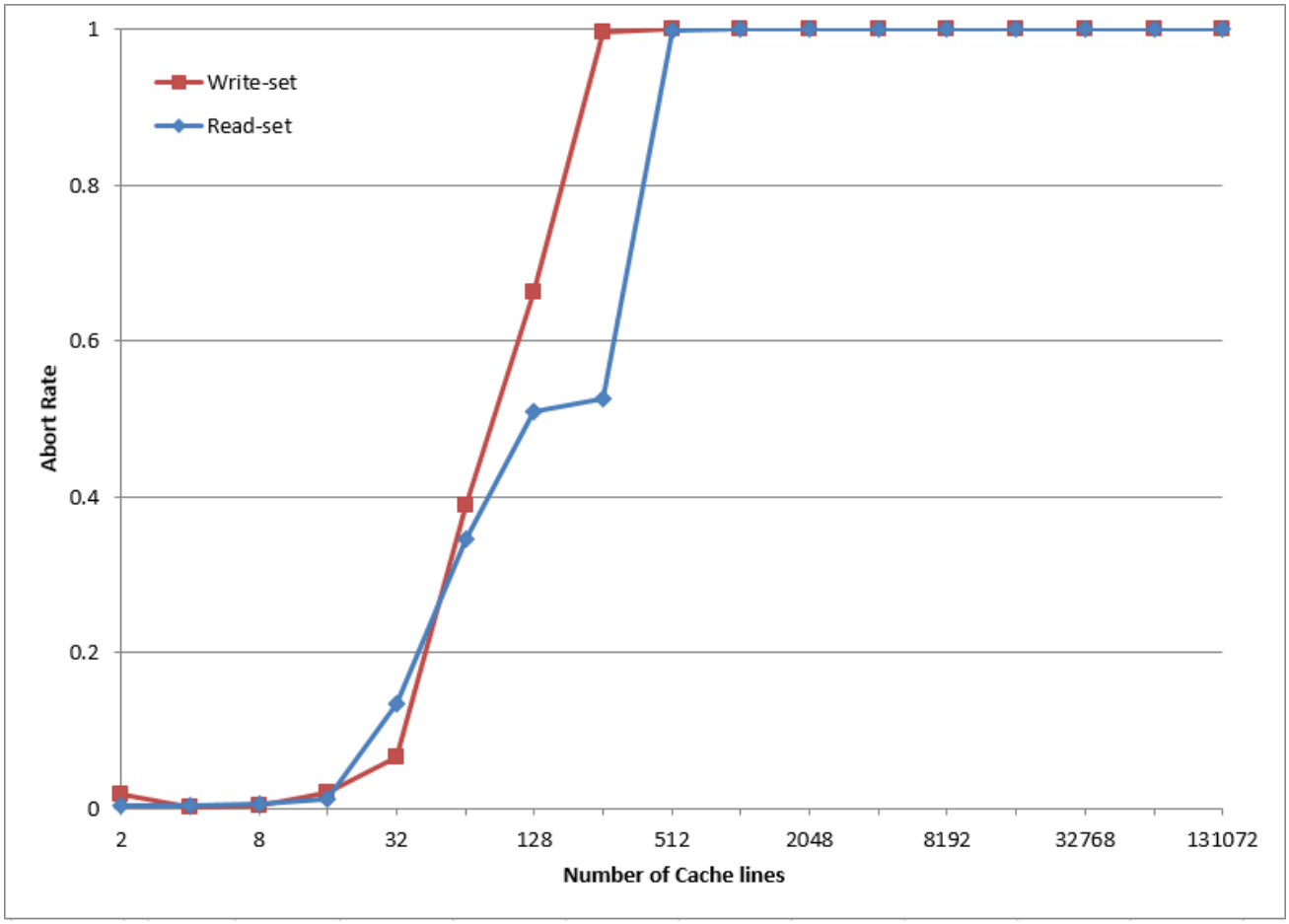

Figure 4: TSX RTM abort rate versus cache lines accessed for a two threads on hyper-threaded core

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.

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 [16].

To evaluate how long a transaction can safely execute for, a third benchmark was developed to perform 100 to 1000000 increment operations on a single element. This benchmark was executed with a single thread to minimize the possibility of data conflicts. Figure 5 shows the transaction abort rate as the duration of the transaction is increased, i.e. the number of operations performed within the transaction.

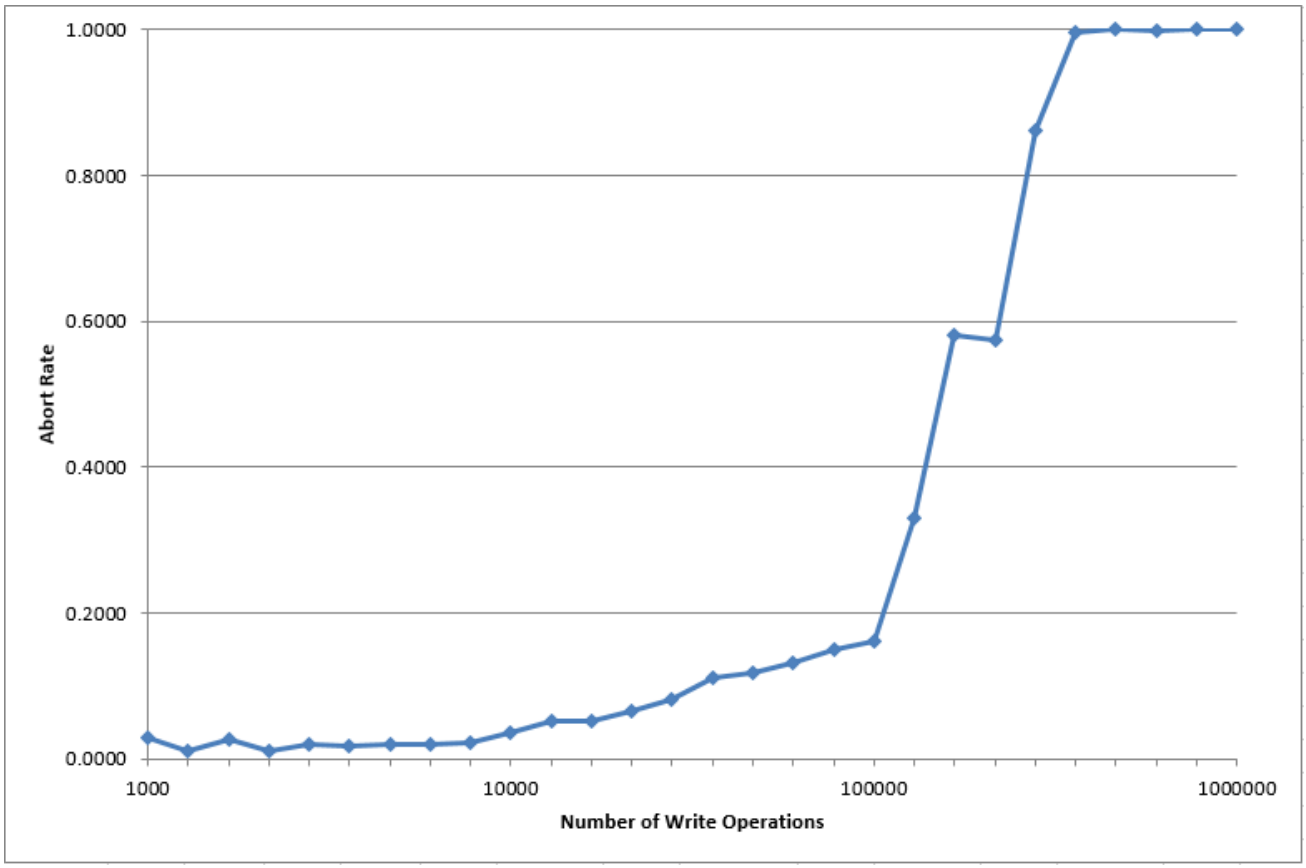

Figure 5: TSX RTM abort rate versus cache lines accessed for a two threads on hyper-threaded core

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 that shorter transactions are more likely to succeed than longer transactions.

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 goal of this benchmark is to compare the TSX latencies to conventional synchronization mechanism latencies. To be clear, this benchmark merely demonstrates how long TSX synchronization mechanisms take to enter transactional regions 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, and calls the release function the data 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 6. Clearly the HLE and RTM mechanisms take longer to actually complete the synchronization process. This can most likely be attributed to the extra actions the hardware must perform to initiate transactional execution.

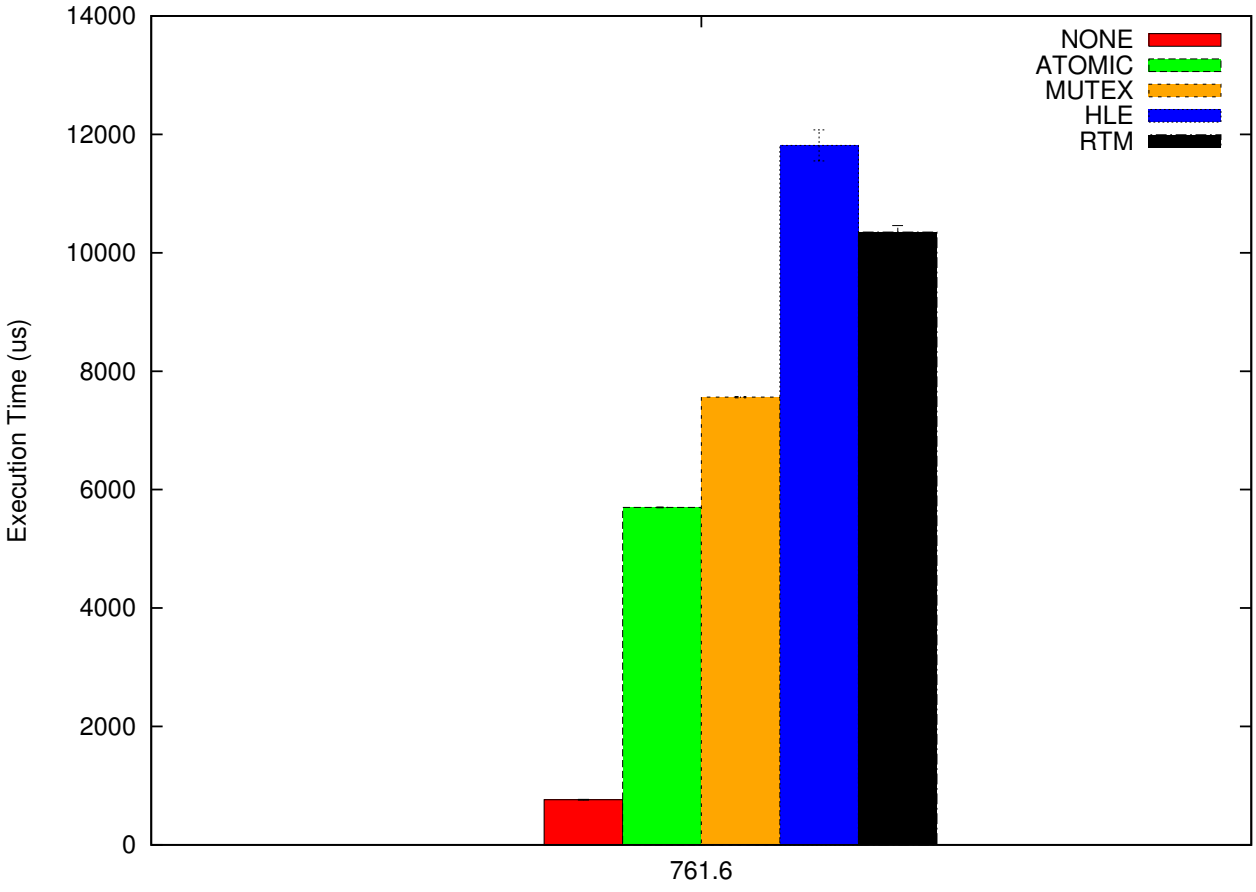

Figure 6: Synchronization Latency

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].

4 The Problem Space

4.1 Parallel Discrete Event Simulation 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 timestamped event from the pending event set, 2) Processes the event, 3) Updates the LPs state, and 4) inserts generated events into the pending event set.

Physical processes are represented by Logical Processes (LPs) in the simulation [13]. For example, in an epidemic simulation, LPs represent geographical locations containing a subset of the total population. The LPs state represents the diffusion of the disease within the location and the status of the occupants within the location. Events in this simulation represent the arrival or departure of individuals to or from locations, the progression of a diseased individual within a location, the diffusion of a disease within a geographical location, etc. [17]. To effectively model epidemics, a significant population size and number of locations needs to be simulated. 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 seaparate 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 PDES being the most susceptible. Optimistic approaches, such as the Time Warp protocol which is used in the WARPED simulation kernel, detect rather than prevent causal errors. This allows for increased concurrency as events are continually executed until a causal error is detected. In contrast, conservative approaches do not execute events until the system has determined it is safe to do so [9].

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 timestamp smaller than the timestamp 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 timestamp 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 timestamp 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 7 shows the scheduling state of the LP as a straggler event is received. Figure 8 shows the scheduling state of the LP after the rollback is processed [7].

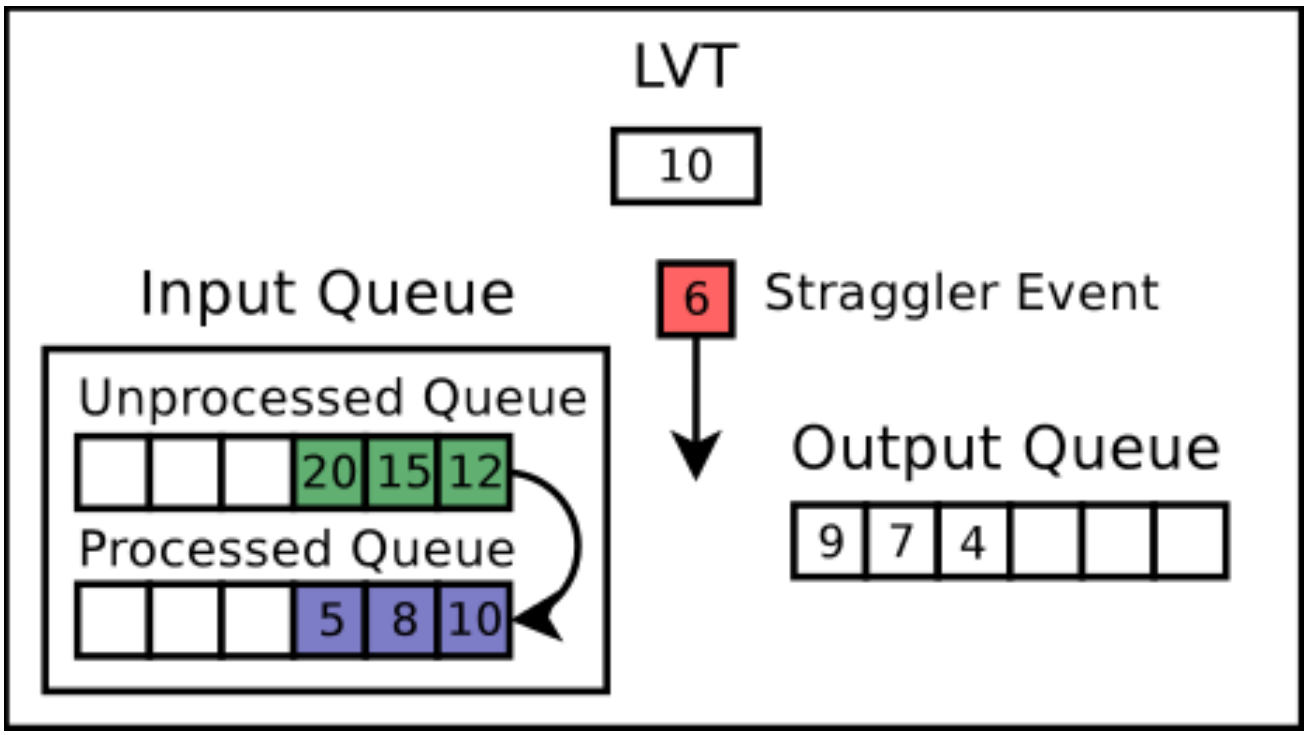

Figure 7: LP at the time of a straggler event is received

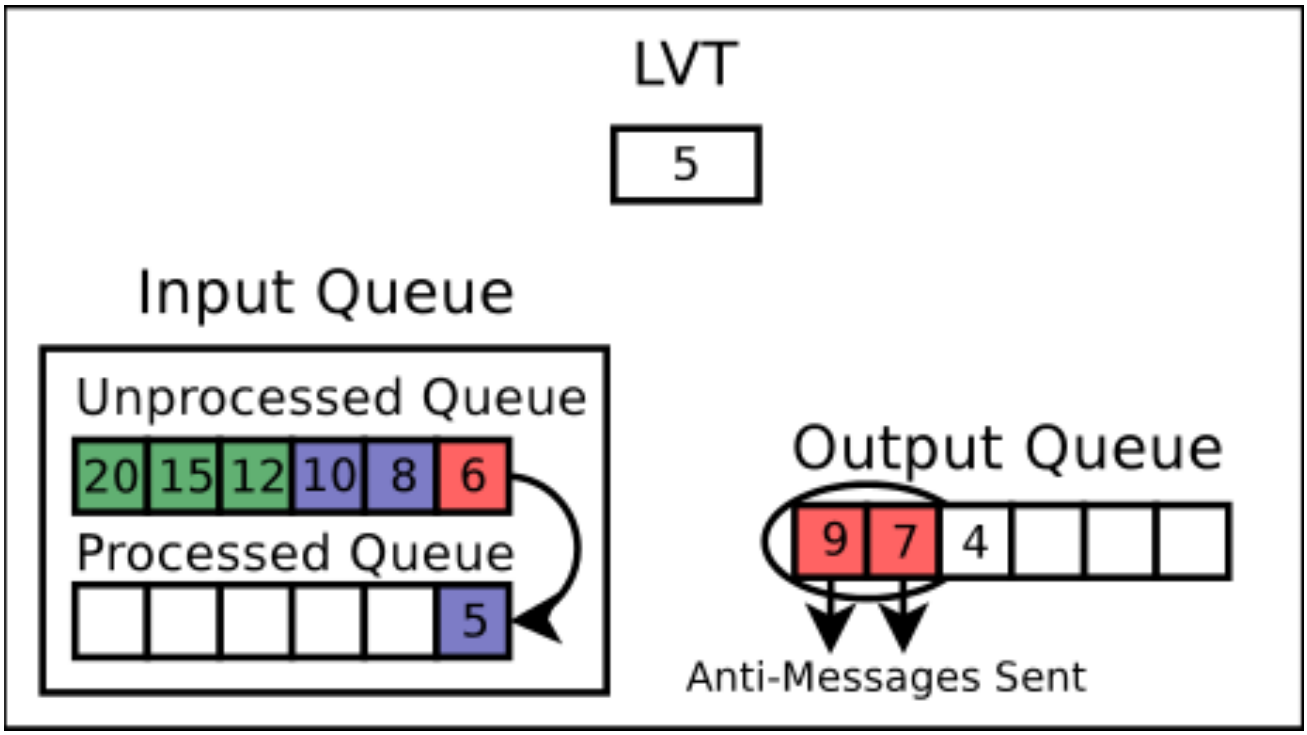

Figure 8: LP after a rollback is processed

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 [19].

4.2 WARPED and the Pending Event Set

WARPED is a publicly available Discrete Event Simulation (DES) kernel implementing the Time Warp protocol [12, 9]. It was recently redesigned for parallel execution on multi-core processing nodes [14]. 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 timestamp 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 timestamp order so that worker threads can simply retrieve an event from he head of the queue. This guarantees the worker thread retrieves the least timestamped 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 [19]. The WARPED simulation kernel has two fully functional implementations: 1) using the STL multiset data structure, and 2) using a splay tree data structure.

4.2.1.1 STL MultiSet

4.2.1.2 Splay Tree

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 timestamped event.

```
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 9: Generalized event execution loop for the worker threads. Many details have been omitted for clarity.

If the worker thread succesfully 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 timestamped event. The dequeued event is inserted into the LTSF queue, which resorts itself based on the event timestamps. An abstract event processing algorithm is shown in Figure 9 [7]. Note that the worker threads perform many other functions. The entire pending event set implementation can be seen in Figure 10 [7].

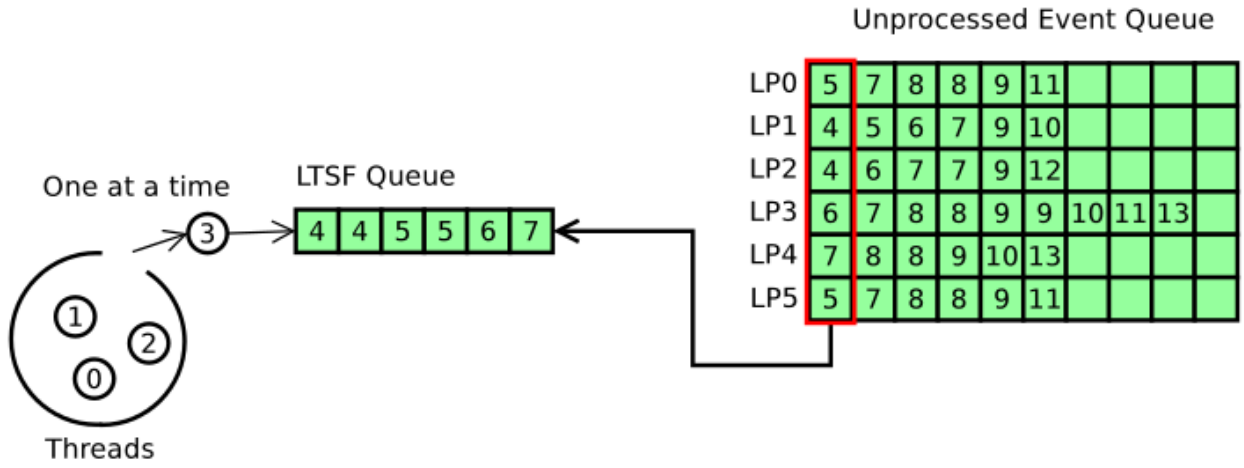

Figure 10: Pending Event Set Scheduling

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 11. 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.

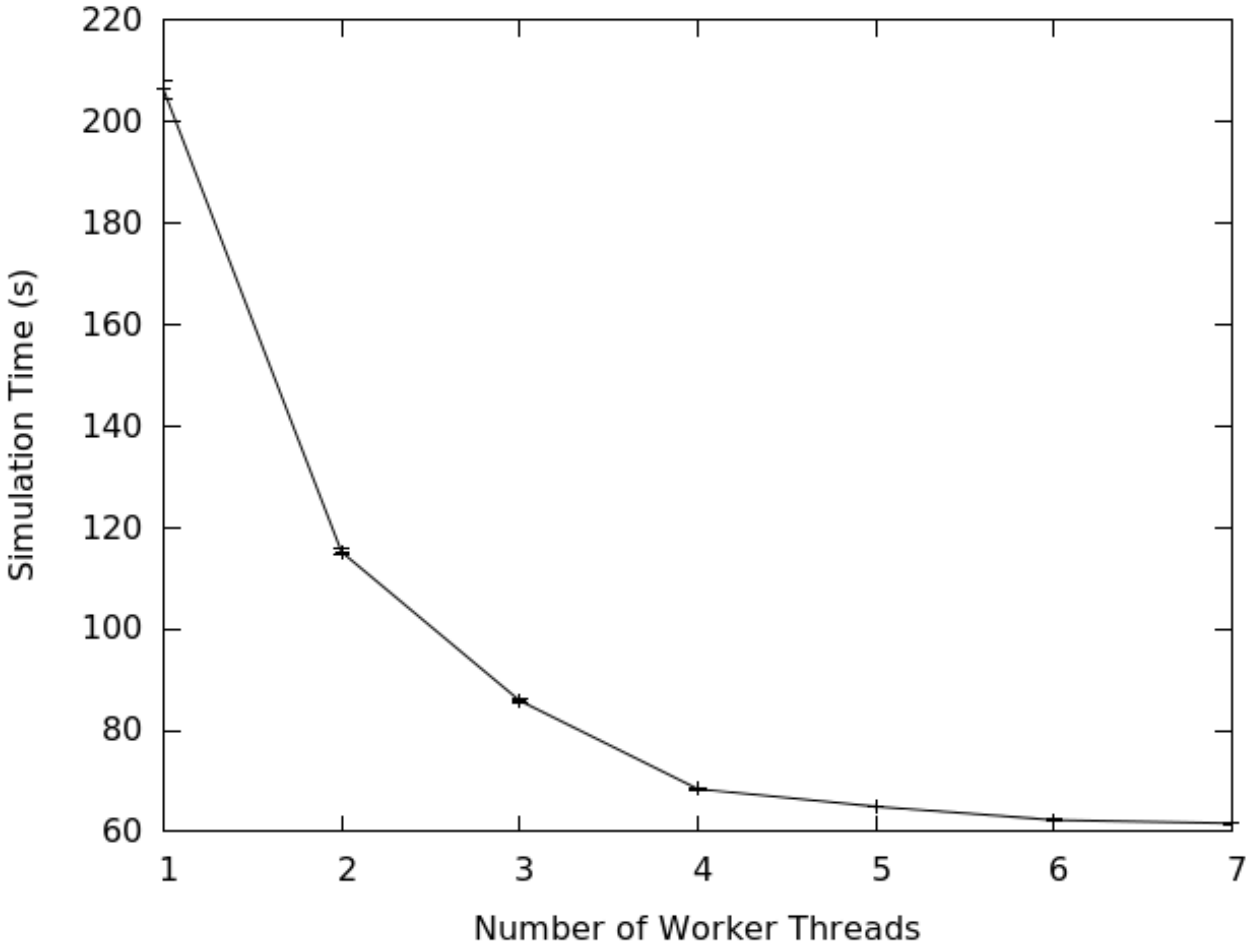

Figure 11: WARPED Simulation Time versus Worker Thread Count for Epidemic Model

4.3 Previous Solutions to Contention

Dickman et al. explored the explored the use of various data structures in the WARPED pending event set implementation, specifically, the STL multiset, splay tree, and ladder queue data structures [7]. A secondary focus of this study will expand upon the use of splay tree versus STL multiset 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 12. 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 reassing LPs to LTSF queues. This study expands the previous multiple LTSF queue to evaluate if contention can be reduced even further with TSX.

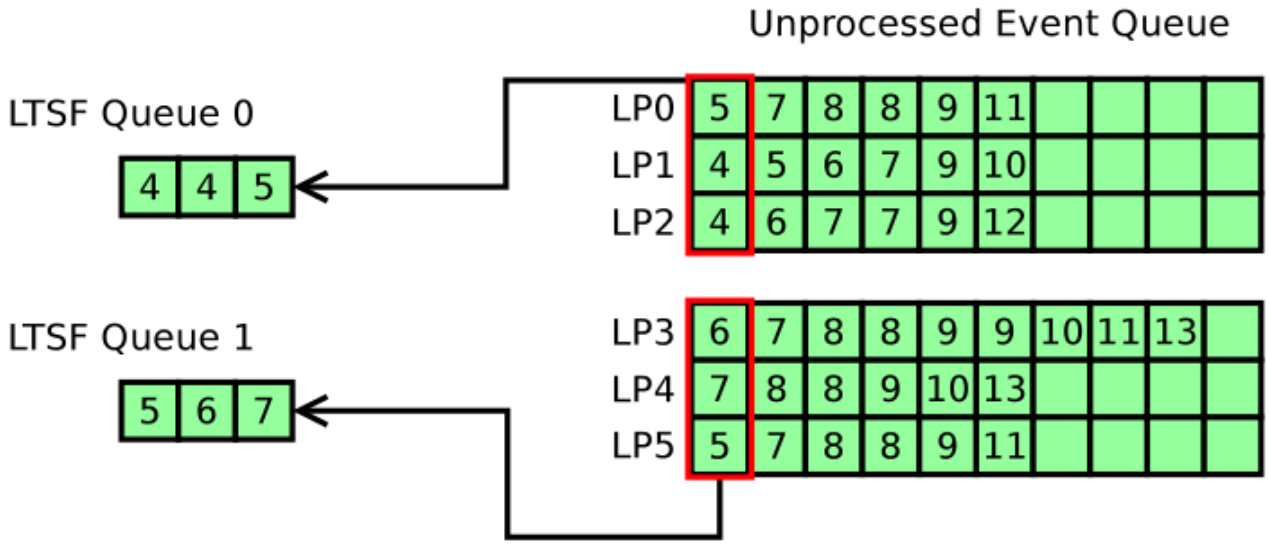

Figure 12: Pending Event Set Scheduling with Multiple LTSF Queues

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, the worker thread would insert the next event into the same LTSF it had just scheduled from as seen in Figure **??**. In this implementation, the worker thread inserts the next event into a different LTSF queue, based on a circularly incrementing counter.

It was discovered that this implementation resulted in poor performance on NUMA architectures. Jingjing Wang et al. noticed similar performance degredation, which they attributed to poor memory locality due to the movement of LPs to different threads [**?**]. 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

```
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 13: Generalized event execution loop for migrating worker threads. Many details have been omitted for clarity.

certain number of events.

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 alwasy 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 timestamped event from a specific LP's unprocessed queue.
- `getEvent()` - dequeue and return the least timestamped 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 timestamp 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 timestamp 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 interefere can execute concurrently. TSX tracks read and write memory operations separately in the transaction's read-set and write-set resepctively. 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 timestamp 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 sycnhronization. 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 scenarious 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 runtime events that can cause transcations to abort, as well as physical limitations.

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 reuiring 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 Lock Contention

The first part of this study compares the performance of the original WARPED pending event set implementation using: 1) atomic locks, 2) HLE, and 3) RTM with 1 retry, 4) RTM with 4 retries, and 5) RTM with 9 retries. It should be noted that the original implementation of the WARPED pending event set only allows for a number of worker threads that is eveny divisble by the number of worker threads due to the way in which LP's are partitioned to LTSF queues. These results are shown in Figures 14 and Tables 1, 2, and 3.

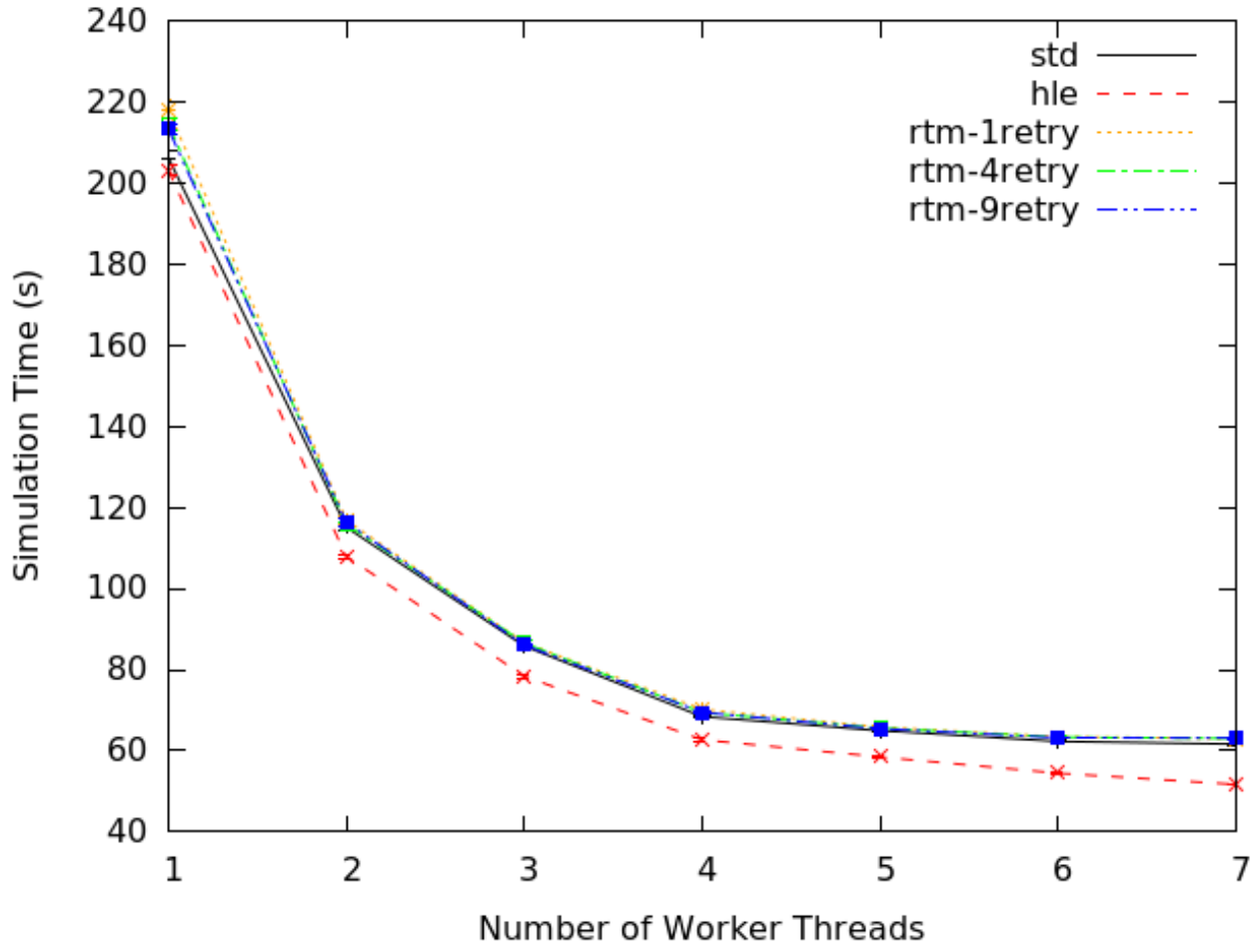

Figure 14: Simulation Time as Number of Worker Threads is Increased Using Various Synchronization Mechanisms for 1 STL Multiset LTSF Queue

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 4-retry	RTM-9retry
1	115.1073	107.93	116.8573	115.8511	116.3141
2	111.6427	104.7166	113.0621	121.884	130.4556

Table 1: Simulation Times for 2 Worker Threads with X LTSF Queues

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 4-retry	RTM-9retry
1	68.40767	62.70367	70.23204	69.44761	69.46226
2	73.62335	69.23382	74.28421	74.5701	74.40304
4	100.74668	97.87896	100.70637	98.65185	97.00871

Table 2: Simulation Times for 4 Worker Threads with X LTSF Queues

# LTSF Queues	Lock	HLE	RTM 1-retry	RTM 4-retry	RTM-9retry
1	62.32475	54.4856	63.46503	63.38292	63.22529
2	61.03439	56.03277	61.72473	61.38556	61.23396
3	65.605	63.3307	65.24451	64.92354	64.93433
6	73.50022	69.54849	72.83046	71.01408	64.13042

Table 3: Simulation Times for 6 Worker Threads with X LTSF Queues

It is evident from Figure 14 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.

6.1.1 Continuous Thread Migration

Based on the less than desirable results above, a roaming thread migration technique was implemented to alleviate contention on the LTSF queues. It distributes threads that attempt to access the same LTSF queue to separate LTSF queues. Pseudocode for this implementation can be seen in Figure 15 **TODO:get psuedocode figure**. Many details are omitted for clarity.

Figure 15: Generalized event execution loop for migration worker threads

6.1.2 Thread Migration for X Events

7 Conclusions

References

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

[2] *Intel 64 and IA-32 Architectures Optimization Reference Manual. Chapter 12: Intel TSX Recommendations*, 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?, 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*, page 103.

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

[12] 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.

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

[14] 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.

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

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

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

[18] 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.

[19] 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*, page 101.

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

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

[22] 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)*, page 127.

[23] 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.

[24] 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.