# Experiments with Hardware-based Transactional Memory in Parallel Simulation

Joshua Hay
Intel Corporation
Portland, Oregon
hayjo813@gmail.com

Philip A. Wilsey
University of Cincinnati
Cincinnati, OH 45221-0030
wilseypa@gmail.com

## ABSTRACT

Transactional memory is a concurrency control mechanism that dynamically determines when threads may safely execute critical sections of code. It provides the performance of fine-grained locking mechanisms with the simplicity of coarse-grained locking mechanisms. With hardware based transactions, the protection of shared data accesses and updates can be evaluated at runtime so that only true collisions to shared data force serialization. This paper 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, we explore the application of Intel's hardware-based transactional memory (TSX) to manage shared access to the pending event set by the simulation threads. Comparison between conventional locking mechanisms and transactional memory access is performed to evaluate each within the WARPED Time Warp synchronized parallel simulation kernel. In this testing, evaluation of both forms of transactional memory found in the Intel Haswell processor, Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM), are evaluated. The results show that RTM generally outperforms conventional locking mechanisms and that HLE provides consistently better performance than conventional locking mechanisms, in some cases as much as 27%.

## Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming, distributed programming*
; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*parallel, distributed, discrete event*

## Keywords

Transactional memory, parallel simulation, optimistic synchronization, pending event set


*SIGSIM-PADS'15,* June 10 - 12, 2015, London, United Kingdom
ACM ISBN 978-1-4503-3583-6/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2769458.2769462 .

## 1. INTRODUCTION

Multi-core processors introduce an avenue for increased software performance and scalability through multi-threaded programming. However, this avenue comes with a toll: the need for synchronization 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, 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 of threads to essentially execute sequentially through the critical section even when they are accessing completely independent parts 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 [18]; 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 regardless of whether coarse-grained or fine-grained locking is used.

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 [18]. In the above example, instead of using locks, the programmer identifies the critical section as a transactional region (hereafter, the terms *critical region* and *transaction* will be used interchangeably). As the threads enter the transactional region, they attempt to "atomically" 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 conventional locking mechanisms, the threads are allowed to safely execute the critical

section concurrently. TM is analogous to traffic roundabouts whereas conventional synchronization mechanisms are analogous to conventional traffic lights [16].

Transactional memory operates on the same principles as database transactions [9]. 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). A key challenge area in PDES is the need for contention-free pending event set management solutions [5]. Transactional memory can help alleviate contention for this shared structure and potentially expose untapped concurrency in the simulation's execution.

This paper explores the use of transactional memory to manage the pending event set schedule queue in the WARPED parallel simulation kernel. In particular, we will integrate the hardware-based transactional memory primitives from the Intel Haswell platform to manage the pending event set data structures of the WARPED parallel discrete event simulation engine. While WARPED has multiple shared data structures in the kernel, the focus of this work is on the pending event set. It is the primary bottleneck in PDES applications, and hence the primary motivation for this study.

The remainder of this paper is organized as follows. Section 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. Section 8 provides some 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. Section 9 discusses how TSX is incorporated into the WARPED pending event set implementation. It also provides a brief overview of the critical sections utilizing TSX and why TSX will be beneficial. Section 10 presents the experimental results of this research for different simulation configurations. Finally, Section 10.2 contains some concluding remarks.

# 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 [18]. 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 committed 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 transaction writes to (called the *write-set*). Each transaction builds its own read-set and write-set as it executes. While a thread is executing transactionally, any memory operation performed by any other thread is checked against the read-set and write-set of the transactionally executing thread to determine if any memory operations conflict. The other threads can be executing either non-transactionally or transactionally. If the transaction completes execution and the TM system has not detected any conflicting memory operations, the transaction atomically commits all of the 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 or write-set [11]. 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*.

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 [9]. 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 [9]. 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 [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 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 instantaneous-

ness. 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 [12, 3], 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, mostly in software [25, 2, 4, 3, 22, 7, 1]. As the name suggests, Software Transactional Memory (STM) systems implement the memory tracking, conflict detection, write buffering and so on in software. 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 [7]. 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. In general, STM has been found to result in significant slowdown [1]. Although STM is more widely available than HTM, its use in this this study was dismissed due to the significant performance penalty.

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 [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 [3, 4]. However, AMD has not released any HTM enabled processors as of this study, 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 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 shows 20-25% increased throughput for write-only workloads and increased throughput for 50% read / 50% write workloads [2]. 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. They found that the performance was comparable to the lock-free implementation without the added complexity [24]. 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 [25]. 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 [12]. It is a best effort implementation, meaning it does not guarantee transactions will commit [11]. 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.

The ***Hardware Lock Elision*** (HLE) interface is a legacy-compatible interface introducing two instruction prefixes, namely: `XACQUIRE` and `XRELEASE`.

The `XACQUIRE` prefix 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 [11]. 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 (the lock) 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 [11]. 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. If another thread is executing the same transactional region, those transactions will also abort. The aborted transaction thread performs an explicit write on the lock, resulting in a data conflict for any other transaction as the lock variable is part of the other transaction's read-set. The re-execution of the critical section using conventional synchronization is necessary to guarantee forward progress [11].

To enable HLE synchronization, the programmer merely adds the HLE memory models to the existing locking intrinsics (Figure 1). The __ATOMIC_HLE_ACQUIRE tells the thread to execute an `XACQUIRE` prefixed lock acquire instruction when another thread releases the lock. The combination of memory models, `__ATOMIC_HLE_ACQUIRE|__ATOMIC_ACQUIRE)` allows for the locking instructions to be executed with or without elision. The local thread can be synchronized to a `XRELEASE` prefixed lock release instruction or a standard lock release instruction.

HLE is legacy compatible. Code utilizing the HLE inter-

```
/* Acquire lock with lock elision if possible */
/* Loop until the returned value
   indicates the lock was free */
while(__atomic_exchange_n(&lock, 1,
      __ATOMIC_HLE_ACQUIRE|__ATOMIC_ACQUIRE)):

/* Begin executing critical section/
   transactional region */
...
/* End critical section/transactional region */

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

**Figure 1: Generic HLE Software Implementation**

face can be executed on legacy hardware, but the HLE prefixes will be ignored [11] and the processor will 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.

The ***Restricted Transactional Memory*** (RTM) interface for HTM introduces four new instructions, namely: `XBEGIN`, `XEND`, `XABORT`, and `XTEST`.

The `XBEGIN` instruction marks the start of a transaction, while the `XEND` instruction makes the end of a transaction. The `XABORT` instruction is used by the programmer to manually abort a transaction. Finally, the `XTEST` instruction can be used to test if the processor is executing transactionally or non-transactionally.

The `XBEGIN` instruction transitions the processor into transactional execution [11]. Note that the `XBEGIN` instruction does not elide 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 abort the transaction. But no matter the reason for the abort, execution jumps to the fallback instruction address [11]. This address is specified as 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 to determine precisely what happens on failure of a transaction. Even so, the programmer must still provide an abort path that guarantees forward progress [11]. Therefore, the abort path should use explicit synchronization, *e.g.,* acquire a 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 [11], such as whether or not the abort was caused by the `XABORT` instruction, a data conflict, so on. The programmer can use this information to make more informed decisions

```
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/transactional region */
...
/* End critical section/transactional region */

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

**Figure 2: Generic RTM Software Implementation**

regarding reattempting transactional execution.

The RTM implementation is more involved because it uses entirely new instructions. 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 ultimately uses explicit synchronization and guarantees forward progress. GCC 4.8 and above includes support for the `_xbegin`, `_xabort`, and `_xend` intrinsics [21].

While RTM is a 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 [12].

# 3. PRACTICAL PROGRAMMING WITH TSX

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. This discussion of this chapter is directly related to the Intel Haswell i7-4770 processor implementation of HTM. Generalization of these results and the corresponding discussion to other processor implementations of HTM should not be made.

# 4. MEMORY ORGANIZATION

TSX maintains a read-set and a write-set with the granularity of a cache line [11]. 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 attempts to (i) read an address in the write-set or (ii) 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 or written to.*

Therefore, aborts can be caused by data occupying the same cache line, essentially reporting a false conflict on the shared data [12]. 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.

## 5. TRANSACTION SIZE

TSX maintains a transaction read-set and write-set in the L1 cache [12]. 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 [12]. Based on granularity of the read-set and write-set stated above, the 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 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

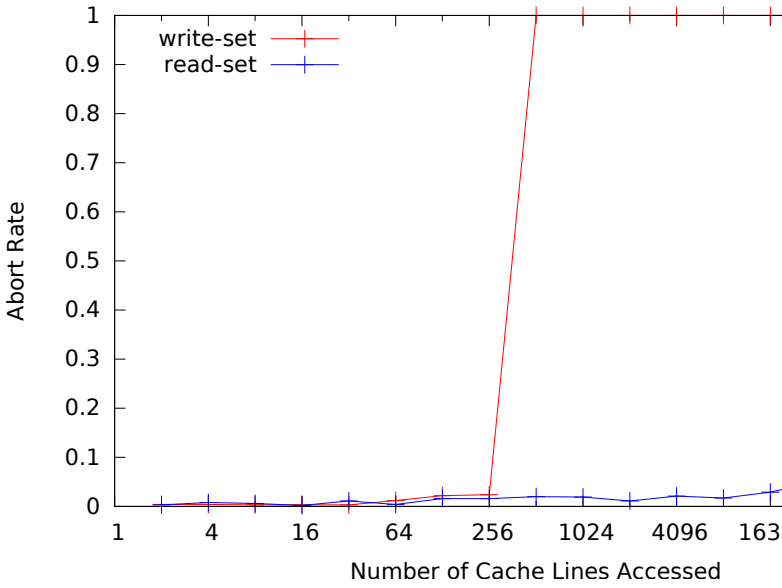


**Figure 3: TSX RTM abort rate versus cache lines accessed for a single thread on one core**

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 [12].

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 4 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.

## 6. TRANSACTION DURATION

Transaction aborts can be caused by a number of run-time events [11], 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 [**?**].

The objective of the third benchmark is to evaluate the running time restrictions for a transaction, *i.e.*, how long can

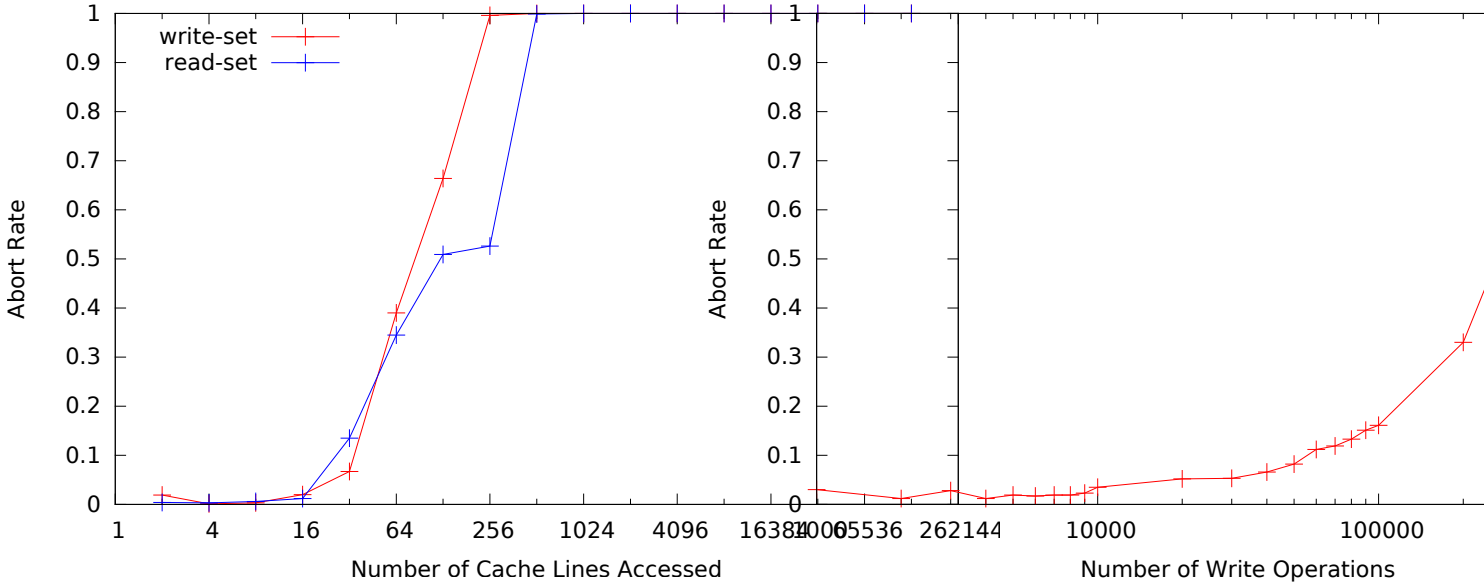


**Figure 4: TSX RTM abort rate versus cache line accesses for two threads on hyper-threaded core.**

**Figure 5: TSX RTM abort rate versus number of operations performed during transaction.**

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 5 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.

## 7. 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 [11]. 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 [11].

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 [11].

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 [11].

## 8. PDES AND WARPED

Discrete Event Simulation (DES) models a system's state changes at discrete points in time. In a DES model, physical processes are represented by Logical Processes (LPs) [14]. For example, in an example epidemic simulation (an example of which is used in this study), LPs can 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 at that location. Executed Events in this simulation represent the arrival or departure of individuals to or from that location, the progression of a disease within an individual at that location, the diffusion of a disease throughout that location, etc [17]. To effectively model epidemics, a significant population size and number of locations needs to be simulated.

WARPED is a publicly available Discrete Event Simulation (DES) kernel implementing the Time Warp protocol [13, 6]. It was recently redesigned for parallel execution on multi-core processing nodes [15]. It has many configuration options and utilizes many different algorithms of the Time Warp protocol [6].

The pending event set is maintained as a two-level structure in WARPED (Figure6) [5]. Each LP maintains its own event set as a time-stamp ordered queue. As previously men-

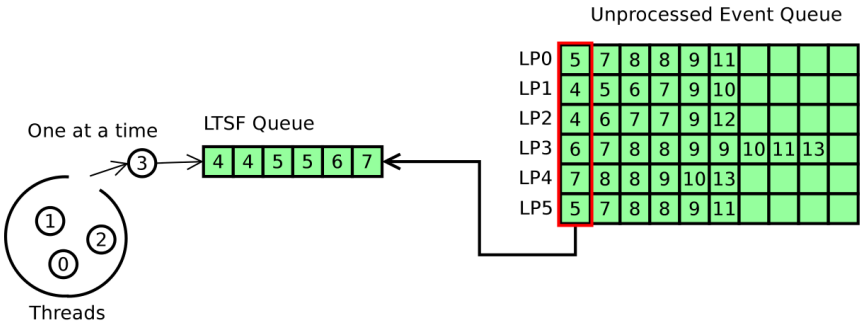


**Figure 6: Pending Event Set Scheduling**

tioned, each LP maintains an unprocessed queue for scheduled events yet to be executed and a processed queue to store previously executed events. A common Least Time-Stamped First (LTSF) 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 the 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.

## 8.1 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 functional implementations: (1) the C++ Standard Template Library (STL) multi-set data structure, and (2) the splay tree data structure. The way that these data structures are accessed and, more importantly, self-adjust will be relevant to how effectively TSX can be used to access these structures. Due to space considerations, only performance results with the multi-set data structure are shown. Results with splay trees are consistent with those described in this manuscript. Interested readers can find those details in [10].

The sorted STL multi-set data structure is an abstract data structure implemented as a self-balancing, red-black binary search tree [8]. 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.

In the STL multi-set, the lowest value element will always 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 that the red-black tree rebalance itself.

One concern with these data structures in relation to TSX is self-adjustment. When these data structures have to self-adjust, the read-set and write-set of the transaction can grow significantly; all transactions operating on that data structure may then need to abort. However, the self-adjustment is a necessary evil. Events need to retrieved from the pending event set and executed in least time-stamped order. If a thread had to search for the least time-stamped event every time it retrieved an event, execution of the simulation would be cripplingly slow. Instead, the pending event set is sorted in order of increasing time-stamp, and the thread can simply fetch the top event in the queue [15]. That being said, there are still opportunities where these data structures may try to self-adjust, but not actually need to write any changes to the structure, *i.e.,* the multi-set queue may already be sorted after insertion. In these situations, only the read-set of the transaction in question will grow, and all concurrently executing transactions may proceed.

We conducted some preliminary studies using TSX with simple data structures such as `std:list` and `std:forward_list` for the pending event set. However, the performance results were much worse due to sorting overheads. Perhaps there are alternate simple data structures or queue organizations that may uncover improved results with TSX, but as of yet we have not uncovered any. Therefore, the experiments reported in this manuscript are restricted to the more complex multi-set and splay tree data structures.

## 8.2 Worker Thread Event Execution

Within a WARPED simulation, a manager thread on each processing node initiates $n$ worker threads at the beginning of the simulation. It can also suspend inactive worker threads if they run out of useful work (events in the pending event set). 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 7 [5]. Note that the worker threads perform many other functions as well.

## 8.3 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.

```
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
    unlock LPi queue

    lock LTSF queue

    insert event from LPi
    dequeue smallest event from LTSF

    unlock LTSF queue
  end loop
```

**Figure 7: Generalized event execution loop for the worker threads. Many details have been omitted for clarity.**

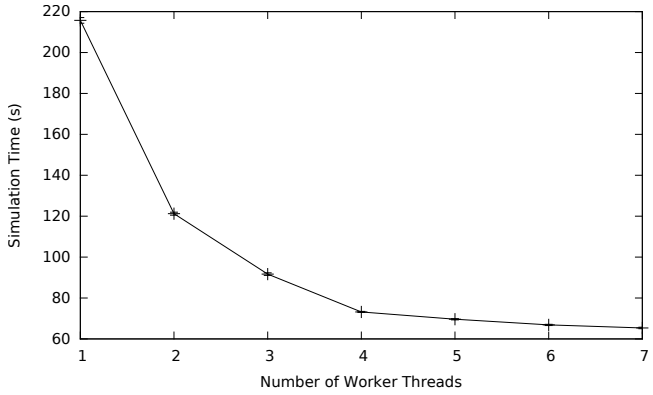


**Figure 8: warped Simulation Time versus Worker Thread Count for Epidemic Model**

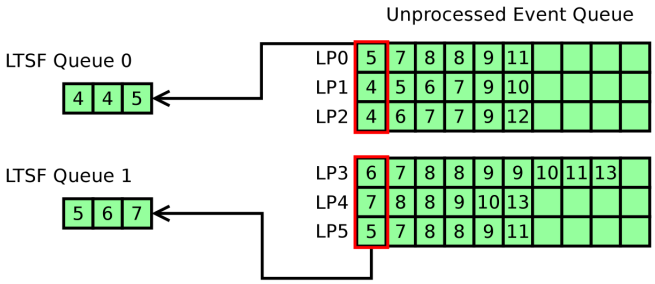


**Figure 9: Pending Event Set Scheduling with Multiple LTSF Queues**

The initial WARPED implementation execution time was measured and analyzed using 1 to 7 worker threads on an Intel i7-4770 with 2-way hyperthreading on 4 processing cores. These results can be seen in Figure 8. It is evident that simulation time becomes less and less affected by increasing the worker thread count, especially when the worker thread count surpasses 4. 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 multicore processor trend will continue to increase the number of simultaneous execution threads available, consequently increasing the contention problem.

## 8.4 Previous Solutions to Contention

Dickman *et al* explored the use of various data structures in the WARPED pending event set implementation, specifically, the STL multi-set, splay tree, and ladder queue data structures [5]. 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 work, the ladder queue implementation was being heavily modified and could not be included in this study.

Another focus of the Dickman *et al* study was the utilization of multiple LTSF queues [5]. Multiple LTSF queues are created at the beginning of the simulation. Each LP is assigned to a specific LTSF queue as shown in Figure 9. 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 [5]. This study expands upon the previous multiple LTSF queue study to evaluate if contention can be reduced even further with TSX.

## 8.5 Thread Migration

Another potential solution to contention is to distribute worker threads that migrate events from the LPs to subsequent LTSF queues. That is, 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 7. In this implementation, the worker thread inserts the next event into a different LTSF queue, based on a circularly

```
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
    unlock LPi queue

    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
  end loop
```

**Figure 10: Generalized event execution loop for migrating worker threads. Many details have been omitted for clarity.**

incremented counter. This approach dynamically reassigns worker threads LTSF queues by migrating the threads to new LTSF queues. It also implicitly balances the load between the all the LTSF queues. The number of LTSF queues is specified in a configuration file, and has no restrictions as in the static assignment.

In a separate (unpublished) study, UC researchers discovered that this implementation resulted in poor performance on Non-uniform Memory Access (NUMA) architectures. Jingjing Wang *et al* also noticed similar performance degradation, which they attributed to poor memory locality due to the movement of LPs to different threads [23]. To offset these performance hits, a migration count was implemented in this scheme. Instead of continuous migration, threads are reassigned to their original LTSF queue after executing a certain number of events. The threads will continue to schedule events from their original LTSF queue for the remainder of the simulation.

# 9. WARPED WITH TSX

This section discusses the various critical sections of WARPED that use the TSX mechanism for this study. As previously mentioned, the primary focus of this study is the shared LTSF queue. The LP event queues also modified to use the TSX mechanism. In this study, experiments with both the RTM and HLE mechanisms are explored.

The following functions require synchronization to access the LTSF queue:

- `insert()`: copy the least time-stamped event from a specific LP's unprocessed queue into the LTSF queue.
- `updatedScheduleQueueAfterExecute()`: find the source LP of the previously executed event, and copy the least time-stamped event from that LP's unprocessed queue into the LTSF queue using the `insert()` function above.
- `nextEventToBeScheduledTime()`: return the time of the event at the beginning of the LTSF queue.
- `clearScheduleQueue()`: clear the LTSF queue.
- `setLowestObjectPosition()`: update the lowest object position array.
- `peek()`: dequeues the next event for execution from the head of the LTSF queue.
- `peekEvent()`: if a simulation object is not specified, call `peek()`.

Most of these critical sections involve write operations, typically through queuing and dequeuing events. Queuing and dequeuing requires the multi-set and splay tree data structures to readjust themselves thus adding more memory locations to the transaction's read-set and write-set. `nextEventToBeScheduleTime()` is the only critical section that performs strictly read operations. Furthermore, many of these critical sections overlap with critical sections from the unprocessed and processed queues, which are described below.

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. However 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 while the first worker thread is in the middle of executing `nextEventToBeScheduleTime`. This scenario necessitates synchronization. However, in a different scenario, 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 runtime events that can cause transactions to abort, as well as physical limitations.

## 9.1 TSX Implementation

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

```
static inline int _xacquire(int *lockOwner,
    const unsigned int *threadNumber)
{
  unsigned char ret;
  asm volatile("mov $0xFFFF, %%eax\n"
         _XACQUIRE_PREFIX "lock cmpxchg %2, %1\n"
         "sete %0"
         : "=q"(ret), "=m"(*lockOwner)
         : "r"(*threadNumber)
         : "memory", "%eax");
  return (int) ret;
}
```

**Figure 11: HLE `_xacquire` Inline Assembly Function**

```
static inline int _xrelease(int *lockOwner,
    const unsigned int *threadNumber)
{
  unsigned char ret;
  asm volatile("mov %2, %%eax\n"
         _XRELEASE_PREFIX "lock cmpxchg %3, %1\n"
         "sete %0"
         : "=q"(ret), "=m"(*lockOwner)
         : "r"(*threadNumber), "r"(0xFFFF)
         : "memory", "%eax");
  return (int) ret;
}
```

**Figure 12: HLE `_xrelease` Inline Assembly Function**

#### 9.1.1 Hardware Lock Elision (HLE)

The generic algorithm presented in Figure 1 only works for locks with a binary value, *i.e.,* the lock is free or it is 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 the CMPXCHG lock instruction.

These functions are shown in Figures 11 and 12. The `_xacquire()` function loads the value 0xFFFF (the value indicating the lock is free) into a specific register, then compares the lockOwner variable with the the previously loaded value to determine if the lock is free. If the values are the same, the CMPXCHG instruction will write the value of the threadNumber variable into the lockOwner variable and return the result. The `_xrelease()` function loads the value of the lockOwner variable into a specific register, then compares the threadNumber variable with the previously loaded value. If the lockOwner value is the same as the thread number, the cmpxchg writes the value 0xFFFF into the lockOwner variable to indicate the lock is free. Of course, if the processor successfully transitions into transactional execution with the HLE prefixes, the write operations technically never occur. They only *appear* to occur to the local thread. Any other thread still sees the lock as free.

#### 9.1.2 Restricted Transactional Memory (RTM)

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 13. 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 before retrying by using the _mm_pause busy-wait loop to try and offset the execution of the conflicting threads. This is done 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 back-off 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. This ideally reduces the number of transaction attempts for an extended period of time. If the transaction commit rate increases, the retry limit increases up to the initial limit specified at compile time. The retry limit increases if the commit rate passes the abort to commit rate ratio threshold.

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

## 10. EXPERIMENTAL ANALYSIS

This study compares the performance of the WARPED simulation kernel using conventional synchronization mechanisms, Hardware Lock Elision (HLE), and Restricted Transactional Memory (RTM). All simulations were performed on a system with an Intel i7-4770 running at 3.4 GHz with 32GB of RAM. The average execution time and standard deviation were calculated from a set of 10 trials for each simulation configuration. When comparing synchronization mechanisms, the simulation execution times are compared for the same LTSF queue and worker thread configurations. When comparing the LTSF queue configurations, the multiple LTSF queue configuration execution time is compared with the execution time of the same configuration with 1 LTSF queue.

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.

### 10.1 The Default Multi-set Schedule Queue

The default implementation of the LTSF queue is the STL multi-set data structure. It is a self-adjusting binary search tree which keeps the least time-stamped event in the left most leaf node of the tree.

#### 10.1.1 Static Thread Assignment

In the original WARPED thread scheduling scheme, threads are statically assigned to an LTSF queue. Contention will clearly be a problem if the simulation only schedules from

```
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 before retry using _mm_pause busy-wait loop

  increment retry count
end loop

acquire lock

execute critical section
```

**Figure 13: RTM Retry Algorithm**

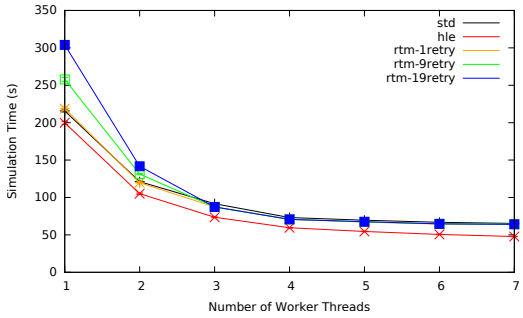


**Figure 14: Performance of a Single Multi-set LTSF Queue**

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,
3. RTM with 1 retry,
4. RTM with 9 retries, and
5. RTM with 19 retries.

These results are shown in Figure 14. It is clear that using HLE improves simulation performance, but still suffers from the same rise in contention as the number of worker threads is increased. The performance using RTM for any retry count used is worse than the standard locking mechanism initially. As the number of worker threads is increased, the performance using RTM is slightly better than the standard locking mechanism, but only by about 2 or 3%.

It is evident from Figure 14 that contention is increasing as the number of worker threads increases, regardless of the synchronization mechanism used. This is somewhat expected as contention is still high for the single LTSF queue. Transactional memory exposes concurrency where it can, but some critical sections simply cannot be executed concurrently. It should be noted that the performance of HLE does not flatten quite as much as the other synchronization mechanisms.

The initial solution to alleviate contention for the LTSF queue is the utilization of multiple LTSF queues. The data for different numbers of schedule queues is limited by the necessity to have a number of LTSF queues evenly divisible by the number of worker threads. This is because of the way threads are assigned to LTSF queues; if the numbers are not evenly divisible, the simulation becomes unbalanced. LPs assigned to a certain LTSF queue can get far ahead or behind of other LPs on different LTSF queues resulting in significant rollbacks and thus performance degradation.

Figure 15 shows the simulation results for varying worker thread configurations using 2 LTSF queues. The load balancing restrictions discussed above restrict the available data for these results. Each synchronization configuration yields roughly the same increasing performance trend. RTM performance seems to be worse with more retries with a lower worker thread count, but eventually converges with the single retry scheme. On the other hand, HLE synchronized

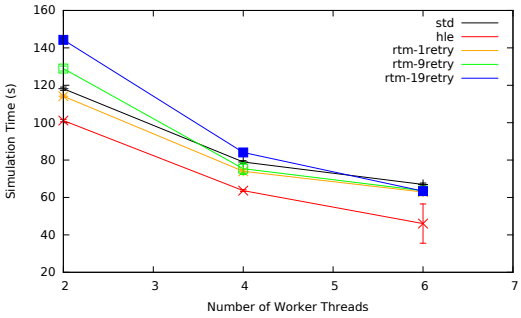


**Figure 15: Performance of Multiple Worker Threads, 2 LTSF Queues**

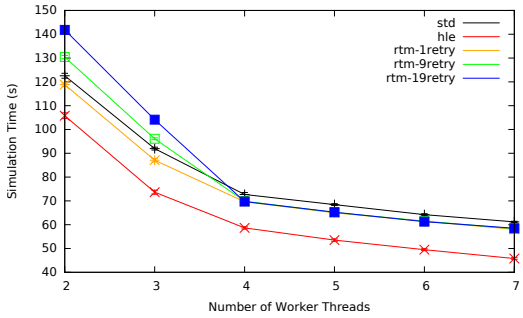


**Figure 17: Performance of Multiple Dynamically Assigned Worker Threads, 2 LTSF Queues**

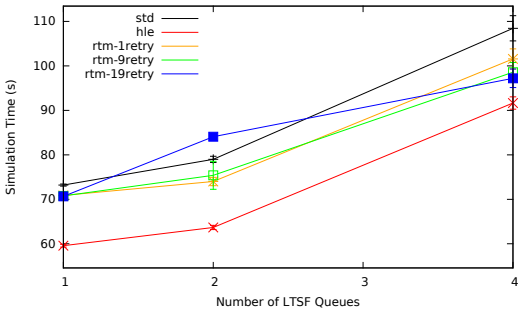


**Figure 16: Performance of Multiple Multi-set LTSF Queues, 4 Statically Assigned Worker Threads**

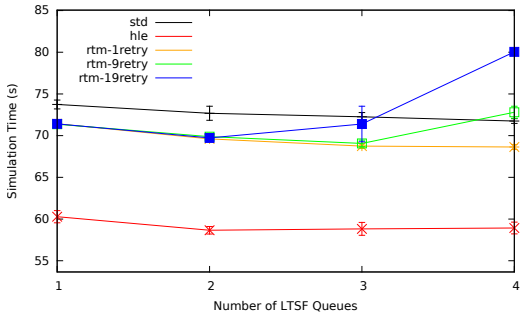


**Figure 18: Performance of Multiple Multi-set LTSF Queues, 4 Dynamically Assigned Worker Threads**

simulations consistently outperform simulations using the standard synchronization.

The LTSF queue count configuration per worker thread configuration results are shown Figure 16. Using 2 LTSF queues with 2 statically assigned worker threads appears to alleviate contention. Using HLE, simulation execution time was reduced by 13-14% regardless of the number of LTSF queues used. RTM improved performance using only 1 retry, but only by about 1-3%. Using any more retries resulted in worse performance. Using the standard locking mechanisms, simulation execution time reduced by about 2.5% increasing the LTSF queue count from 1 to 2. With TSX, simulation execution time reduced by about 4% when increasing the LTSF queue count from 1 to 2. While only a small difference, TSX managed to reduce contention a bit more in conjunction with multiple LTSF queues.

While TSX, specifically HLE, substantially improved simulation performance, as much 22%, simulation execution time increased as the number of LTSF queues used was increased in other configurations. It was noted that these simulations resulted in significantly higher rollbacks, the most likely cause of the increased execution time. These poor performance results could be attributed to the lack of a proper load balancing procedure, which is addressed with dynamic thread assignment.

### *10.1.2 Dynamic Thread Assignment*

Another solution to contention is to distribute worker threads that try to simultaneously access the same LTSF queue to different LTSF queues. Worker threads are dynamically assigned to LTSF queues rather than statically.

The first solution continuously migrates the worker threads to the next LTSF. That is, the worker thread processes an event from $LTSF_i$ and then $LTSF_{(i+1)modn}$ where n is the number of LTSF queues. As the worker thread moves among the LTSF queues, the worker thread also moves the next event from the just processed LP to the next LTSF queue. This also helps distribute the critical path of events in the LPs around the LTSF queues. This solution implicitly balances the work load between LTSF queues. Therefore, any number of LTSF queues can be used with any number of worker threads.

Figure 17 shows the simulation results for 2 LTSF queues using the continuous migration scheme as the number of worker threads is varied. Similarly to the static scheduling scheme, the simulations for each synchronization mechanism seem to follow almost the same trends. The more retries the RTM algorithm attempted, the worse performance was for 2 and 3 worker threads. However, the number of retries did not affect the RTM performance for 4 or more worker threads.

Simulation execution time decreased slightly by increasing the number of LTSF queue with 4 worker threads (Figure 18). Each multiple LTSF configuration reduced simulation execution time by 2-3% when compared to the single LTSF queue configuration. The only exception to this trend is the 4 LTSF queue configuration with HLE; it reduced simulation execution time slightly less than standard locking mechanisms, but the difference seems trivial. While RTM performed well for lower LTSF queue counts, the increased retry counts resulted in worse performance for greater LTSF queue counts. In any configuration, HLE still reduces exe-

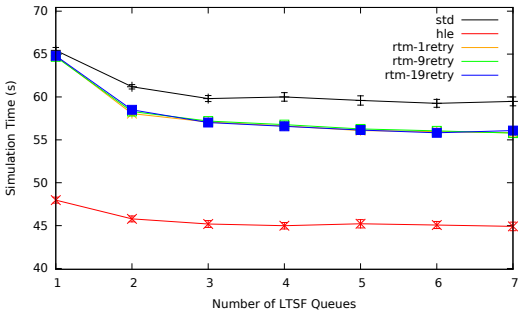


**Figure 19: Performance of Multiple Multi-set LTSF Queues, 7 Dynamically Assigned Worker Threads**

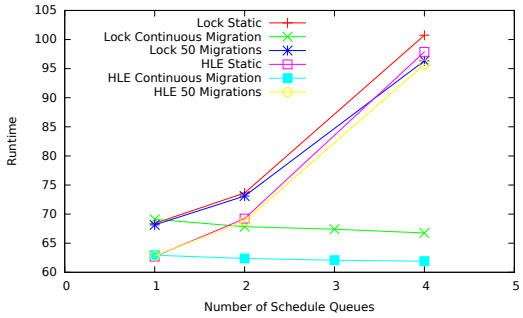


**Figure 21: Comparison of Migration Schemes for 4 Worker Threads with X LTSF Queues**

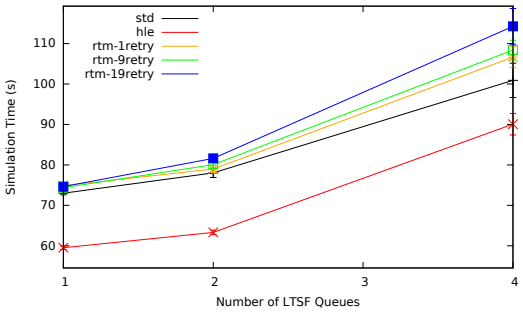


**Figure 20: Simulation Time versus Number of STL Multi-set LTSF Queues for 4 Worker Threads**

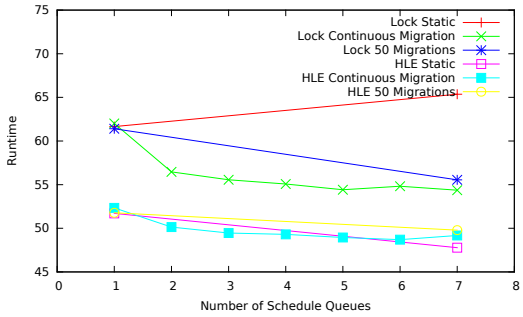


**Figure 22: Comparison of Migration Schemes for 7 Worker Threads with X LTSF Queues**

cution time by about 18%, while RTM generally generally reduces execution time by about 3-4% when comparing the two to standard locking mechanisms.

The final simulation configuration uses 7 worker threads with 1 to 7 LTSF queues (Figure 19). Using standard locking mechanisms with multiple LTSF queues reduces execution time by 6% to 9% as the number of LTSF queues is increased. Surprisingly, HLE only reduces execution time by 3% to 5%. But again, HLE still well outperforms the standard locking mechanisms by as much as 27%. RTM only outperforms standard locking mechanisms by about 5%. However, it becomes much more effective with more LTSF queues. Execution time improvements increased from 9% to almost 14% when using RTM with increasing LTSF queues counts.

As previously discussed, the continuous thread migration approach does not work well for NUMA architectures due to memory locality issues. The thread migration scheme was modified to migrate threads between LTSF queues for the first 50 events a thread executes. In the first implementation of this scheme, after a thread executes 50 events, it is no longer reassigned to a different LTSF queue. It continues to schedule from the same LTSF queue as it did for the 50th event for the remainder of the simulation.

While the continuous migration scheme is not problematic for the system under test, the comparison was made to thoroughly evaluate TSX using this scheme as a viable solution to contention. TSX may also one day become available on NUMA architectures. Further testing would need to be performed, but at least it will be known if this solution has any significant impact on contention.

These results are shown in Figure 20. It is evident that any static thread to LTSF queue assignment suffers from the same problems. Except for the 2 worker thread, 2 LTSF queue and 3 worker thread, 3 LTSF queue configurations, performance suffers as the number of LTSF queues is increased. Load balancing becomes an issue with this migration scheme because worker threads can become unevenly divided among the LTSF queues leading.

The second implementation attempts to address the load balancing issue by reassigning worker threads to their original LTSF queues after successfully executing the specified number of events. After a thread is reassigned to its original LTSF queue, it continues to schedule events from that queue for the remainder of the simulation.

Unfortunately, the simulation results were incredibly inconsistent using this scheduling scheme. A significant portion of the simulations did not complete execution in the allotted time. The longer running simulations experienced significantly higher rollbacks. When the simulation does appear to run normally, it executes slightly faster than the strictly static thread assignment scheme. However, the instability of this migration scheme made it infeasible to obtain data.

The migration scheme makes a significant difference in contention and load balancing. Figures 21 and 22 show the comparison of the migration schemes used. The first implementation of the event limited migration scheme is shown below since the second implementation performance could not be adequately measured.

## 10.2 Conclusions

This paper explored the use of Intel's transactional mem-

ory implementation, Transactional Synchronization Extensions (TSX) in the multi-threaded WARPED PDES kernel to alleviate contention for the pending event set. The WARPED pending event set consists of a global Least Time-Stamped First (LTSF) queue and local event set queues for each LP.

Based on the results, it clear that TSX improved the performance of WARPED. HLE consistently shows speedup over conventional synchronization mechanisms. It even slightly reduces execution time when the simulation only uses one LTSF queue. In other configurations, HLE reduces execution time by as much as 27% and consistently reduces execution time by 20%.

While HLE is the superior synchronization mechanism, RTM still showed increases in performance, generally by about 5%. It also works with multiple LTSF queues better than HLE. This is most likely attributed to the retry algorithm. HLE transactions only have one chance to execute a transaction. If contention is high at certain times, the transaction will most likely abort. The RTM retry algorithm uses abort information to decide when to retry transactional execution, rather than immediately aborting the transaction or using conventional synchronization mechanisms. RTM might not perform as well as HLE due to the overhead associated with RTM. The retry algorithm requires abort statistics to be calculated and maintained which adds a bit more overhead to RTM.

TSX is not likely to allow simultaneous access to the same LTSF queue when the structure is being written. TSX synchronization mechanisms also appear to be more expensive. The performance increases seen with TSX are most likely result from the concurrent execution of critical sections involving only read operations. Furthermore, some critical sections bypassed their write operations under certain conditions. For example, a check was performed within a critical section to ensure the LTSF queue was not empty. If the queue was empty, the critical section ended without performing any operations. With standard synchronization, this critical section would still suffer from the locking overhead, even though it wasn't necessary. With TSX synchronization, the check could potentially execute concurrently with another thread. The same scenarios apply to each LP's processed and unprocessed queue. Overall, TSX reduced unnecessary contention.

In conclusion, TSX significantly improves simulation performance for the WARPED PDES kernel. While other solutions to contention showed improvements in performance, they were not nearly as significant as TSX, especially HLE. TSX only showed slight improvements in its own performance when combined with these other solutions. Regardless, TSX is powerful solution to contention.

## Acknowledgments

Support for this work was provided in part by the National Science Foundation under grant CNS–0915337.

## 11. REFERENCES

[1] 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? *Communications of the ACM*, 51(11):40–46, 2008.

[2] V. Chitters, A. Midvidy, and J. Tsui. Reducing synchronization overhead using hardware transactional memory. Technical report, University of California at Berkeley, Berkeley CA, 2013.

[3] 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 *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 39–50, New York, NY, 2010. ACM.

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

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

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

[7] 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 *Proc of the 23rd International Conference on Supercomputing (ICS'09)*, pages 126–135, 2009.

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

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

[10] J. Hay. Experiments with hardware-based transactional memory in parallel simulation. Master's thesis, School of Electronic and Computing Systems, University of Cincinnati, Cincinnati, OH, June 2014.

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

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

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

[14] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, Mar. 1986.

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

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

[17] K. S. Perumalla and S. K. Seal. Discrete event modeling and massively parallel execution of epidemic outbreak phenomena. *Simulation*, 88(7):768–783, July 2012.

[18] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, New York, NY, 2001. ACM.

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

[20] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley and Sons, Inc., 8th edition, 2008.

[21] R. M. Stallman et al. *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 gene/q hardware support for transactional memories. In *Proc of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT-12)*, pages 127–136, 2012.

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

[24] Z. Wang, H. Qian, H. Chen, and J. Li. Opportunities and pitfalls of multi-core scaling using hardware transactional memory. In *Proc of the 4th Asia-Pacific Workshop on Systems*, pages 3:1–3:7, 2013.

[25] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Proc of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 19:1–19:11, 2013.