Adaptive Transaction Scheduling for Transactional Memory Systems

Richard M. Yoo and Hsien-Hsin S. Lee

School of Electrical and Computer Engineering

Georgia Institute of Technology

Atlanta, GA 30332

{yoo, leehs}@ece.gatech.edu

Technical Report GIT-CERCS-07-04

February 2007

Abstract

Transactional memory systems are expected to enable parallel programming at lower programming complexity, while delivering improved performance over traditional lock-based systems. Nonetheless, we observed that there are situations where transactional memory systems could actually perform worse, and that these situations will actually become dominant in future workloads as more and larger-scale transactional memory systems are available. Transactional memory systems can excel locks only when the executing workloads contain sufficient parallelism. When the workload lacks the inherent parallelism, blindly launching excessive transactions can adversely result in performance degradation. To quantitatively demonstrate the issues, we introduce the concept of effective transactions in this paper. We show that the effectiveness of a transaction is closely related to a dynamic quantity we call contention intensity. By limiting the contention intensity below the desired level, we can significantly increase transaction effectiveness. Increased effectiveness directly increases the overall performance of a transactional memory system. Based on our study, we implemented a transaction scheduler which not only guarantees that hardware transactional memory systems perform better than locks, but also significantly improves performance for both the hardware and software transactional memory systems.

1 Introduction

Due to their radically simplified semantics, transactional memory systems [7] are anticipated to be the enabling technique for parallel programming for general-purpose computing at lower programming complexity. Especially for hardware transactional memory systems, the programmers need not worry about the correctness of a multithreaded program; transactional memory systems will safely parallelize those code portions that are parallelizable, while serializing those portions that are not. Moreover, by executing transactions speculatively and rolling back only when there is a collision, transactional memory systems can effectively extract parallelism to excel the performance of lock based systems which pessimistically serialize all the transactions that share common objects.

These promising characteristics, fueled by the emergence of chip multiprocessors or multi-core processors, led to the development of several transactional memory systems. On the hardware side, researchers proposed Transactional Memory Coherence and Consistency (TCC) [4], Unbounded Transactional Memory (UTM), Large Transactional Memory (LTM) [1], Virtualizing Transactional Memory (VTM) [14], and Log-based Transactional Memory (LogTM) [12]. On the software side, there are DSTM [6], WSTM [5], OSTM [3], ASTM [8], and RSTM [9]. Until now, transactional memory research thrusts mainly focus on implementation issues. Nonetheless, as we see more practical transactional memory implementations, the performance issues will become a new concern in deploying these systems.

From performance perspective, transactional memory systems seem to be perfect; it should always perform better than locks. This seems to be especially true for hardware transactional memory systems, which could minimize the overhead of launching and clearing up a transaction by hardware support. However, we stress that this belief is rather elusive. In this paper, we show that there are certain situations where transactional memory systems could actually perform worse than the lock-based systems.

This situation happens when the executing workload lacks sufficient parallelism. In this situation, lock based systems do not hurt the performance since they were pessimistically serializing every transaction. Nonetheless, for a transactional memory system, excessive transactions can actually result in performance degradation if they abort each other reciprocally, leading to a potential livelock situation. This behavior not only slows down the entire system, but could jeopardize the program completion. On the contrary, in a transactional memory system which only aborts younger transactions, the overhead of managing a transaction significantly increases the sunken cost of an aborted transaction as aborts become frequent. The cost of aborting a transaction is multidimensional. Firstly, all the computational work that an aborted transaction has finished become useless. If the transaction has aborted other transactions before, the total loss becomes even worse. Secondly, depending on the implementation, the cost to rollback a transaction could

be tremendous. Especially for transactional memory systems that have to roll back the memory state, the rollback operation would incur significant memory bandwidth. Lastly, because transactions are implemented on top of threads, frequently aborting a transaction can adversely affect the virtual memory page management policy of an operating system. For example, in Linux [2], starting a transaction and suddenly aborting it amounts to *polluting* the per-CPU page frame cache. This will lead to increased page faults.

To quantitatively explain the situation, we introduce the concept of *Transaction Effectiveness*. It is defined as the ratio of transactions that actually commit. In a workload where there is less parallelism, the effectiveness of a transaction will be low. If a workload is highly parallelizable, the effectiveness of a transaction will be close to 1. The reciprocal of transaction effectiveness gives us the number of ineffective transactions that were executed to successfully commit a single transaction. Lower transaction effectiveness means that more transactions are being aborted, and when multiplied by the cost of aborting a single transaction, the total cost simply lost in ineffective transactions can be calculated.

It is this loss in ineffective transactions that degrades the performance of a transactional memory system. Simply put, when there is not sufficient parallelism, blindly executing excessive transactions can actually hurt the performance. Even worse, as more cores or threads (i.e. parallelism) are available in the future workloads, the ineffective transactions will become a showstopper for transactional memory systems. Because the transactional memory systems guarantee the correctness of a parallel program, it is likely that future developers would devote less effort to write efficient codes.[1] Although the transactional memory system could guarantee correctness for those codes, it cannot guarantee parallelism when the code itself is written in an un-parallelizable manner. Not knowing that the code is inherently sequential, current transactional memory systems would lose performance in executing excessive transactions. A mechanism is needed to prevent transactional memory systems from launching excessive, conflicting transactions.

During our study we noticed that the effectiveness of a transaction is closely related to a dynamic quantity termed *Contention Intensity*. This feature denotes the intensity of the contention a transaction encounters. Note that contention intensity is not equal to transaction effectiveness since not all the colliding transactions will abort; contention intensity relates to transaction effectiveness via the actual contention management scheme of a transactional memory system. However, compared to the transaction effectiveness, which we do not have control over, contention intensity can be dynamically controlled by limiting the number of transactions to be executed at a given time. By limiting the contention intensity below the desired level we can significantly increase transaction effectiveness. Increased transaction effectiveness again directly improves the overall system performance.

[1] A similar argument is that when computers become faster, fewer programmers will invest time to write higher quality codes. The history repeats itself.

This paper is about how to detect and limit the contention intensity for improving performance on transaction memory systems. To this aim, we propose the concept of *adaptive transaction scheduling* on transactional memory systems. Based on this study, we successfully devise a transaction scheduling mechanism that guarantees that the transactional memory system perform at least as well as locks when the workload does not exhibit sufficient parallelism. Moreover, due to its adaptive nature, our scheduling system could significantly improve performance when there is sufficient parallelism. We implemented our transaction scheduler for both the hardware and software transactional memory systems to demonstrate its capabilities.

In summary, the main contribution of this paper is as follows.

- We show that transactional memory systems can actually perform worse than locks on workloads that lack parallelism.
- To quantitatively analyze this situation, we introduce the concept of transaction effectiveness and contention intensity.
- We propose a transaction scheduler which guarantees and improves the overall performance by adaptively controlling the contention intensity.

To the best of our knowledge, we believe that this paper is the first to incorporate transaction scheduling on transactional memory systems to adaptively exploit the maximum parallelism. The rest of the paper is organized as follows. In Section 2 we discuss in detail when transactional memory systems lose performance, then we formally define transaction effectiveness and contention intensity. Section 3 introduces our scheduling mechanism. Section 4 discuss the implementation of our scheduling mechanism on LogTM and RSTM. Section 5 presents the results. Related studies in transactional memory area can be found in Section 6. We finally conclude in Section 7.

2 Backgrounds

In Section 2.1 we discuss in detail where transactional memory systems can lose performance. To quantify the behavior we introduce the concept of transaction effectiveness in Section 2.2. In Section 2.3 we introduce the concept of contention intensity, which can be used to control transaction effectiveness.

2.1 Where Transactional Memories Fail

Transactional memory systems can be classified depending on when they detect conflicts among transactions. A *lazy* system detects conflicts upon the transaction commit time, while an *eager* system detects conflicts

when they acquire a memory block (object) during transaction. In the lazy detection systems [4], because transactions do not get aborted during the execution, in the worst case they only serialize all the transactions. However, computational resources tend to be wasted on those transactions that are doomed to abort. Due to this drawback, most transactional memory systems assume eager conflict detection.

In eager transactional memory systems, when there are no priorities among transactions, every transaction is free to abort the other transaction in the middle of execution. In the worst case scenario, transactions can abort each other reciprocally, leading to a livelock. Figure 1 illustrates such situation. This behavior not only significantly degrades the performance, but could also jeopardize the program completion. Note that this situation does not occur in lock based systems; in the worst case they simply serialize all the transactions.

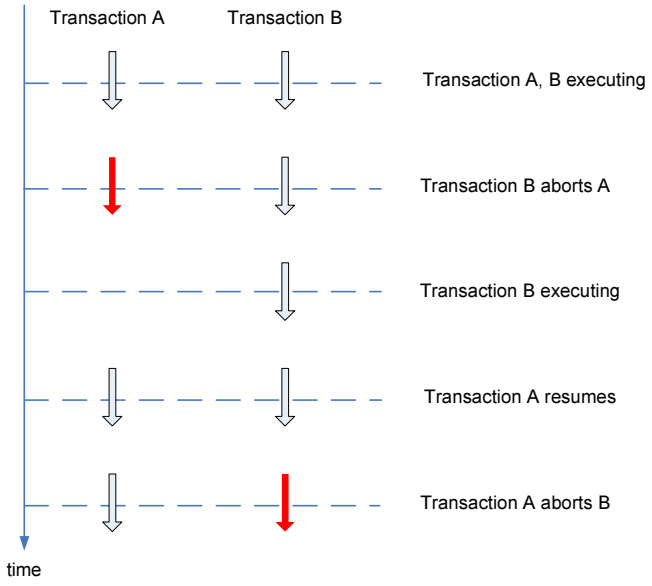

Figure 1: Transactions Aborting Each Other (A Livelock)

Although it has not been comprehensively studied on hardware transactional memory systems, some software transactional memory systems have incorporated *contention manager* [6, 9, 15] to improve this situation. Contention managers usually force some kind of priorities among transactions. Depending on the implementation, this priority could either be the age of a transaction determined by its time stamp, or the amount of work done determined by its memory footprint, etc [15]. When a transaction encounters a conflict, it calls its contention manager. With the priority and other information, contention manager heuristically evaluates and decides whether aborting the offending transaction will improve the overall throughput. When

the contention manager decides rather not to abort the offending transaction, the transaction that asked for decision will back off hoping that the offending transaction finishes before it asks for permission again. By varying the number of back-off attempts and their intervals, contention managers inject artificial parallelism to reduce contention.

However, transactions performing back-off cannot wait indefinitely. Since they were executing the transaction by the time they invoke contention manager, indefinite waiting for the other transaction to finish could introduce possible deadlocks [6]. So after a few back-off, the transaction *must* abort the offending transaction. Due to this characteristic, when the length of the transaction becomes significantly large, contention managers can fail to restrain transactions from aborting each other reciprocally.

This ping-pong scenario exemplifies that there are situations where excessive transactions can actually degrade the performance. Note that in the above example, the performance would have been significantly improved if only one transaction is allowed to be executed at a time. Compared to locks, transactional memories can only benefit when the workload has sufficient parallelism to exploit. If we regard the workload with less parallelism as a funnel with narrow exit, it would be irrational to launch excessive transactions. Nonetheless, since contention managers take effect only after a transaction encounters a conflict, it only cures the result; it does not reduce the cause of contention.

2.2 Transaction Effectiveness

To quantify this behavior, we introduce the concept of *Transaction Effectiveness*. Transactions that commit realize their effect over the entire system, while those that abort are non-effective. When a workload has significant parallelism, more transactions tend to be *effective*. Nonetheless, when the workload lacks parallelism, more transactions tend to be *non-effective*. Note that the resources dedicated to a non-effective transaction, e.g., computational resource, page frames, memory bandwidth, etc., become futile when the transaction fails to commit.

Transaction effectiveness is formally defined as follows:

$$Transaction\ Effectiveness = \frac{number\ of\ transactions\ that\ committed}{total\ number\ of\ transactions\ that\ began\ execution}$$

Simply put, transaction effectiveness amounts to the fraction of transactions that eventually commit among all the transactions that began execution. Workloads that are parallelizable tend to have transaction effectiveness close to 1. When the transaction effectiveness is low, it means that the workload is not parallelizable, and the transactional memory system is wasting its resource on non-effective transactions.

On the contrary, the reciprocal of this quantity gives us the total number of transactions that have been wasted in committing just a single transaction. When multiplied by the cost for aborting a single transaction, the total cost simply lost in ineffective transactions can be estimated. The correlation gradient of the transaction effectiveness and the actual performance is dependent on the actual implementation of the transactional memory system; especially by its check-pointing and rollback mechanisms. Nonetheless, in general, the overall system performance is strongly correlated to transaction effectiveness.

Note that, however, the transaction effectiveness itself is an outcome of execution; we do not have control over the quantity of the effectiveness. In the next section we introduce contention intensity, which not only is highly correlated to the transaction effectiveness but also can be manipulated.

2.3 Contention Intensity

The effectiveness of a transaction is closely related to the intensity of the contention a transaction encounters during its execution. Compared to the transaction effectiveness, which we do not have control over, the contention intensity can be dynamically controlled by limiting the number of concurrently executing transactions at a given time. By limiting the contention intensity below the desired level, we can significantly increase the transaction effectiveness. Increased transaction effectiveness will directly improve the overall system performance.

Contention Intensity can be detected either centralized or decentralized. In a centralized detection scheme a global module will collect the contention information over the entire system, whereas in a decentralized scheme each thread will keep their own contention information. In our study we chose decentralized scheme. Moreover, although there are many ways to quantify the contention intensity, we define the contention intensity as a dynamic average based on current available contention information. Each thread maintains the following quantity:

$$ContentionIntensity_{(n)} = \alpha * ContentionIntensity_{(n-1)} + (1 - \alpha) * CurrentContention$$

This equation is updated whenever a transaction commits or aborts. In this equation the term CurrentContetion is set to 0 when a transaction commits, and set to 1 when a transaction aborts. The weight variable α determines which portion the equation weighs more — either the past history or the current contention information. When the α value is large, the equation biases toward past history; the contention intensity varies slowly while canceling out the noise from the current contention information. On the contrary, when the α value is small, the current contention information is reflected more quickly. Maintaining contention

intensity information enables a parallelism feedback mechanism for a transactional memory system.

Note that the contention intensity relates to the transaction effectiveness via the contention management policy of a given transactional memory system; depending on the actual contention management policy, not all contentions might be realized as aborts.

3 Transaction Scheduling

In the previous section we introduced the concept of transaction effectiveness and contention intensity. In Section 3.1 we will place these concepts in the entire picture of transaction scheduling to show how these metrics can be used to enhance performance. Based on this notion, we devise a simple yet highly effective transaction scheduler in Section 3.2.

3.1 The Whole Picture

Figure 2 shows the typical organization of a transactional memory system. At the heart of performance management is the contention manager. However, it does not have control over the cause of the contention. In this organization, the application is in total control with respect to when to start a transaction.

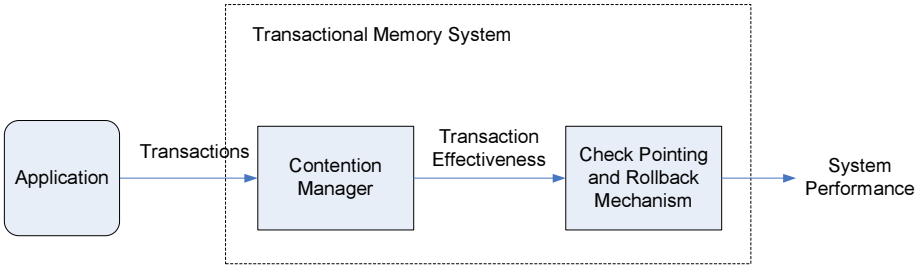

Figure 2: A Transactional Memory System without Transaction Scheduling

Introducing transaction scheduling gives control over the cause of the contention. Figure 3 shows the organization of a transactional memory system with transaction scheduling.

As can be shown from the figure, transaction scheduling introduces three new components: contention intensity, transaction scheduler, and the transaction dispatch unit. Maintaining the contention intensity information exposes the inherent parallelism exhibited in a given workload. High contention intensity means that the workload exhibit less parallelism, and an excessive number of transactions could be squandered

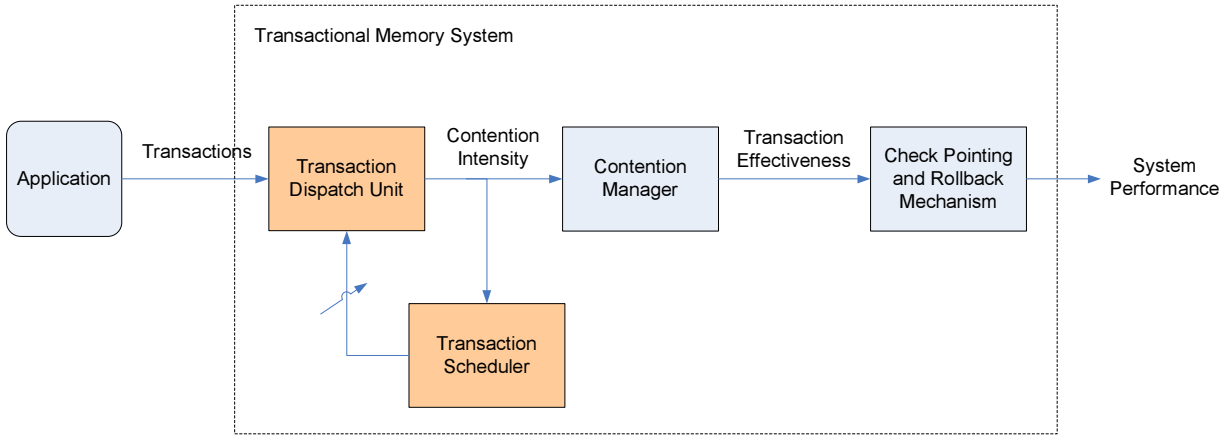

Figure 3: A Transactional Memory System with Transaction Scheduling

for the given workload. This information is then fed to the transaction scheduler. Based on the contention intensity, the transaction scheduler schedules transactions so that the transaction dispatch unit would execute only the appropriate amount of transactions to complete the given task.[2] To accomplish this, the transaction dispatch unit should be able to stall a transaction until the scheduled point. This scheduling will then reduce the contention intensity. Reduced contention intensity increases the transaction effectiveness, which will again increase the overall system performance. On the contrary, low contention intensity implies that there might be a headroom for performance improvement; when the contention intensity is low, transaction scheduler schedules transactions more aggressively so that they could fully exploit the inherent parallelism in the given workload. This organization introduces an adaptive feedback loop for the transactional memory system to adaptively control and manage the contention intensity.

Note that this adaptive transaction scheduling approach is totally different from the contention manager scheme; while the contention manager deals about how to efficiently handle the contention after it has been detected, our transaction scheduler fundamentally reduces the contention itself. Moreover, unlike the contention manager approach, there are no possible deadlocks in the scheduling scheme since the scheduling is performed before a transaction begins its execution. Transactions can wait indefinitely as long as it is guaranteed to execute at some point. This can be guaranteed by making sure that at least one transaction is being executed at a given time, and that every transaction eventually finishes. This also guarantees program completion. Nonetheless, the transaction scheduling scheme can be implemented complementarily with the

[2]Quantitatively speaking, the ideal appropriate amount should achieve the highest possible throughput for the given task.

contention manager approach as they address different properties of a transactional memory system.

3.2 A Simple Transaction Scheduler

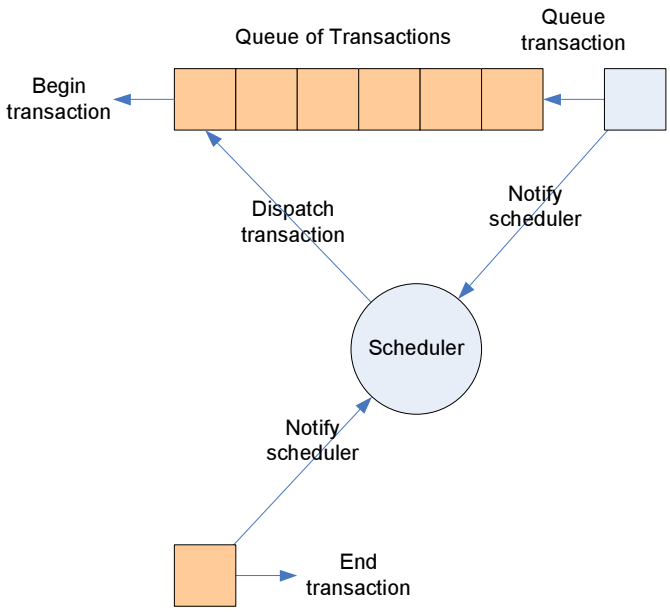

Figure 4: A Queue Based Transaction Scheduler

Based on Section 3.1, we proposed the use of a simple transaction scheduler. Figure 4 shows the organization. This scheduler maintains a single centralized queue of transactions, which resembles the run queue found in operating system thread scheduler. This queue dispatches one single transaction at a time. In this mechanism, each thread maintains their own contention intensity information as described in Section 2.3. When this contention intensity surpasses a designated threshold, each thread queues itself to the centralized queue and notifies the scheduler for a dispatch. If the transaction is at the head of the queue, and if no other transactions that have been dispatched from the queue are executing, it gets dispatched right away. Otherwise, the transaction waits in the queue until the above condition is met. Moreover, the transaction that have been dispatched from the queue must notify the scheduler when it has finished execution. This will trigger the next transaction to be dispatched.

Note that this queuing behavior effectively serializes transactions. At one extreme, when all the transactions are queued, this mechanism gracefully degenerates transactions into a single centralized lock. With properly chosen weight for the moving average and a threshold, this mechanism can guarantee that the performance of transactions would at least be comparable to a lock, in any situation.

The detailed behavior of this scheduler is illustrated in Figure 5 using an example. In this figure, the timeline flows from top to bottom; on the right side locates the hypothetical variation of contention intensity over time. When the contention intensity is below the threshold (timeline 1), transactions begin execution without resorting to the scheduler. However, as the contention intensity grows beyond the threshold, transactions start to queue themselves to the queue managed by the scheduler (timeline 2). When queued at the scheduler, only one transaction can be dispatched from the queue at a time. This effectively reduces the number of concurrent transactions that are executing. At timeline 3, as the transactions get serialized, the contention intensity starts to decrease. When the contention intensity drops below the threshold, some of the transactions that were queued to the scheduler will dequeue themselves from the scheduler to exploit more parallelism (timeline 4, 5).

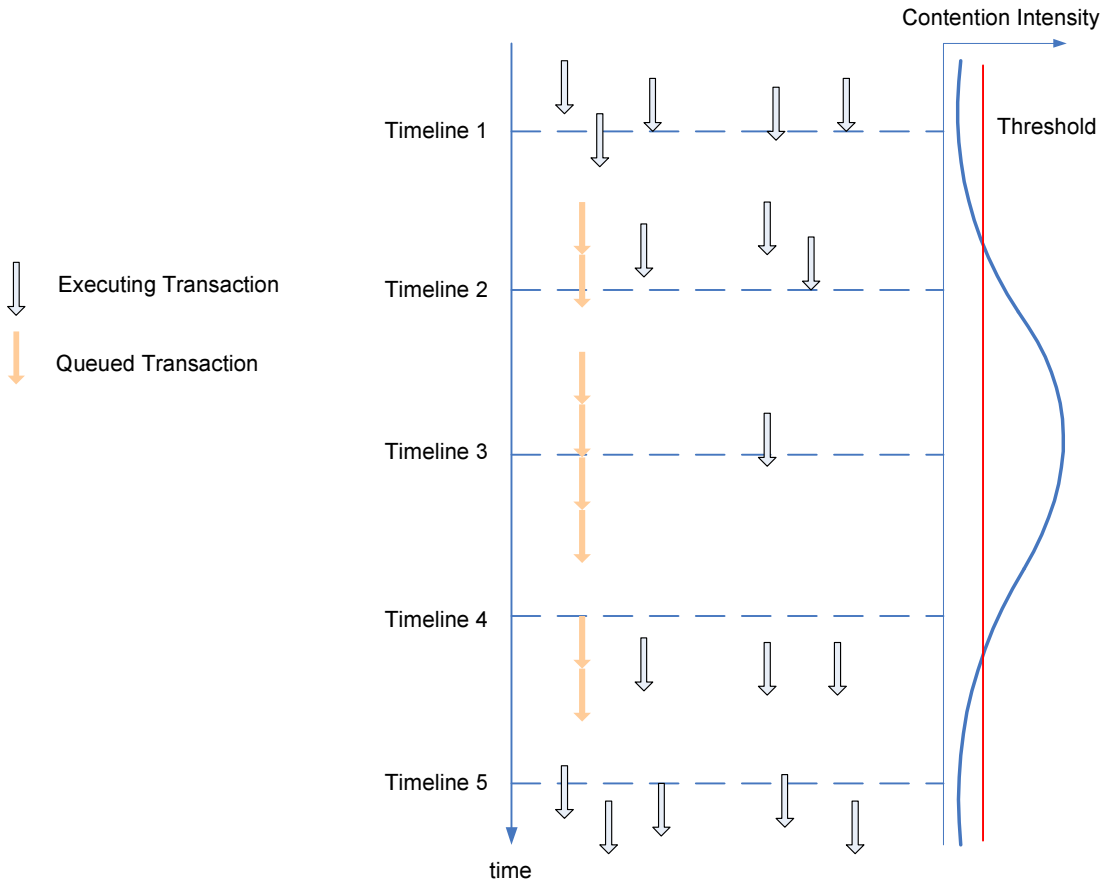

Figure 5: Behavior of a Queue Based Transaction Scheduler

The scheduler adaptively changes the number of concurrently active transactions, so that the contention intensity will not increase without bound. Since the contention intensity is updated dynamically, this simple

scheme can also adapt to phase changes during a program execution; the scheduler will exploit the maximum parallelism inherent at any given phase.

4 Experiment Settings

To study the behavior of transaction scheduling, we implement our proposed queue-based scheduling scheme (Section 3.2) in both the hardware and the software transactional memory systems. Section 4.1 details our scheduler implementation on LogTM [12], and Section 4.2 discuss our implementation on RSTM [9].

4.1 Experiments on LogTM

For the hardware transactional memory system we implemented our scheduling method on LogTM [12]. LogTM has been released as a memory timing module of the GEMS simulator [10]. Internally, LogTM has a dedicated module, the transaction manager, which is accessed when a transaction starts, aborts, and commits. We implemented our scheduling algorithm so that this transaction manager maintains the contention intensity information. Since our implementation supports one active transaction per CPU, the queue size amounts to the total number of CPUs on the system. When a transaction waits in a queue, a no-op is injected to the instruction stream of the pertaining CPU. Table 1 describes the specifications of the simulated machine in GEMS.

Simulated System Settings	
CPU	Sixteen 1GHz SPARCv9 single-issue, in-order, non-memory IPC=1
L1 Cache	4-way split, 64 KB, 5-cycle latency
L2 Cache	4-way unified, 16 MB, 10-cycle latency
Memory	4 GB
Directory	centralized, 6-cycle latency
Interconnection Network	hierarchical switch topology, 40-cycle link latency
LogTM Settings	
Fixed Penalty for Transaction Abort	40 cycles
Transaction Log Writeback Penalty per Block	60 cycles

Table 1: Simulated Machine Configurations for LogTM

The simulated system uses only the Ruby [10] memory timing model of the GEMS simulator. So the CPU is assumed a single-issue, in-order, functionally emulating the SPARCv9 processor. Cache coherence is managed by a central directory, and the interconnection network is based on a hierarchical switch. In LogTM, there was a fixed delay of 40 cycles when a transaction aborts from the system, and an additional

penalty of 60 cycles that is taxed for each block of log written back to the memory.

On this simulated machine we ran the modified version of deque microbenchmark included in the LogTM release. The pseudo-code description of this benchmark is illustrated in Table 2.

```
BEGIN_TRANSACTION(0);

r = get_random_value() % 4;
switch (r)
{
      case 0: enqueue_right(v);
              break;
      case 1: dequeue_right();
              break;
      case 2: enqueue_left(v);
              break;
      case 3: dequeue_left();
              break;
}

do_local_job( transaction_length);

increment_global_counter();

END_TRANSACTION(0);
```

Table 2: Pseudo-code description of deque microbenchmark

Each transaction first enques / dequeues a value on the left / right of a global deque; then it performs a local job; finally it increments the global counter. The major difference from the released version of deque benchmark is that the amount of local job done by a transaction is adjustable by the parameter transaction_length. This parameter controls the length of a transaction — shorter transactions typically increase the level of parallelism while longer transactions tend to reduce its likelihood. By continuously adjusting the parameter, we could examine our scheduler's behavior over a wide spectrum of level of parallelism. When comparing the performance to a lock-based implementation, BEGIN_TRANSACTION and END_TRANSACTION macros were substituted with pthread_mutex_lock(&global_lock) and pthread_mutex_unlock(&global_lock), respecitvely. Moreover, to maximize the concurrency thread affinity was fixed so that each thread executes on a single CPU.

Name	Description
RBTree	A transaction searches down a red-black tree, opening nodes in read-only mode. After locating the target node, the transaction opens it in read-write mode and goes back up the tree opening nodes that are relevant to the height balancing process also in read-write mode.
HashTable	Roughly equal number of insertion and deletion is performed on a hash table with 256 buckets. This hash table implementation has overflow chains.
LinkedList	Transactions traverse a sorted list to locate an insertion/deletion point, while opening nodes in read-only mode. Previously acquired nodes can be heuristically released by transactions if they do not interfere with program correctness. When the target node has been located, it is reopened as read-write mode.
RandomGraph	In an adjacency graph, transactions randomly insert and delete nodes while atomically updating all the neighboring nodes' neighbor lists.
LFUCache	Simulates a web cache which keeps track of the most frequently accessed pages with array-based index and a priority queue.

Table 3: RSTM Benchmark Suite

4.2 Experiments on RSTM

As for the software transactional memory system, we implemented our scheduler on RSTM from University of Rochester [9]. RSTM is a C++ transactional memory library that implements per-object transactions. When an object is passed as an argument to a template, the template returns a transaction enabled wrapper object; between BEGIN_TRANSACTION and END_TRANSACION macros, all the accesses through the read / write method of this wrapper object are treated as transactions. Managing these transactions are completely handled by the software library.

We keep the contention intensity information in each thread's local storage where the RSTM keeps their transaction descriptor. Moreover, the access to the central scheduling queue was serialized with conditional variables in pthread library. Compared to the baseline RSTM, our scheduler implementation is in fact penalized in performance since the conditional variables introduce synchronization overhead in scheduling.

To quantify the performance improvement, we measured the throughput of the entire system with 5 microbenchmark programs that were used in the original RSTM paper [9]. These benchmark programs are also included in the RSTM library release. Each microbenchmark is briefly described in Table 3. More details can be found in [15, 9].

Throughout all the benchmarks, Polka [15] contention manager was used as default. Polka implements an exponential back-off and memory footprint size based priority mechanism. RSTM also allows to configure 1) the visibility of the read-only transaction to other transactions (visible / invisible), and 2) the conflict detection mechanism (eager / lazy). Based on [9], best configurations were chosen for each benchmark. Namely, RBTree, HashTable, and LinkedList benchmark were executed with (invisible, eager) configuration, while RandomGraph and LFUCache were executed with (visible, lazy) configuration. Our scheduler enabled

library was experimented with the same contention manager and the same configurations. When comparing the throughput with the lock-based implementation, we used the cgl library included in RSTM release which transforms the transaction into coarse-grained locking.

We measured the throughput of these benchmark programs by running them on two different machines: a 2-way SMP system and an 8-way SMP system. The 2-way SMP represents the current top-of-the-line dual processor system, while the 8-way SMP system projects the future many-core processors but running at a stripped-down configuration with lower clock frequency and slower bus speed.[3] Table 4 describes the specifications of those two machines and their operating systems.

2-way SMP System	
CPU	2, Intel Xeon 3.0 GHz Front-Side Bus: 800 MHz L2: 2 MB HyperThreading off
Memory	2 GB
Operating System	Red Hat Enterprise Linux AS release 4 2.6.9-34.0.1.ELsmp
8-way SMP System	
CPU	8, Intel Pentium III 550 MHz Front-Side Bus: 100 MHz L2: 2 MB
Memory	4 GB
Operating System	Red Hat Enterprise Linux AS release 4 2.6.9-11.ELsmp

Table 4: RSTM Hardware Settings

5 Experimental Results

In this section we analyze the performance of our transaction scheduling technique on both the LogTM and RSTM systems in Section 5.1 and Section 5.2, respectively.

5.1 Transaction Scheduling on LogTM

In this experiment the deque benchmark was executed with 16 threads. The α value was set to 0.7 and the threshold was set to 0.3. For this value combination, the scheduler will kick in so long as there is one single abort since the coefficient (1-α) for CurrentContention is equal to the threshold value. Figure 6 shows the throughput variation as we increase the transaction length. As defined in Section 4.1, the transaction

[3]Due to the absence of hardware resources we could access, performance results on a 4-way SMP cannot be obtained.

length denotes the amount of local job a transaction performs. More specifically, this parameter amounts to the number of SPARCv9 instructions that a transaction executes locally. So shorter transaction lengths imply more parallelism will be available; while longer transaction lengths reduce the level of parallelism. Also note that the fluctuation of the performance in the graph is due to other non-deterministic elements, i.e., operating system thread scheduling, page frame management, etc.

When there is enough parallelism (transaction length $\leq$ 10240), transactional memory implementations tend to significantly outperform locks. Nonetheless, the baseline LogTM occasionally performs worse than the lock due to contention. On the contrary, as parallelism gets reduced (transaction length $\geq$ 10240), performance of both the transactional memory implementations and the lock tend to converge. However, the LogTM implementation with scheduling constantly outperforms both the baseline LogTM and the lock.

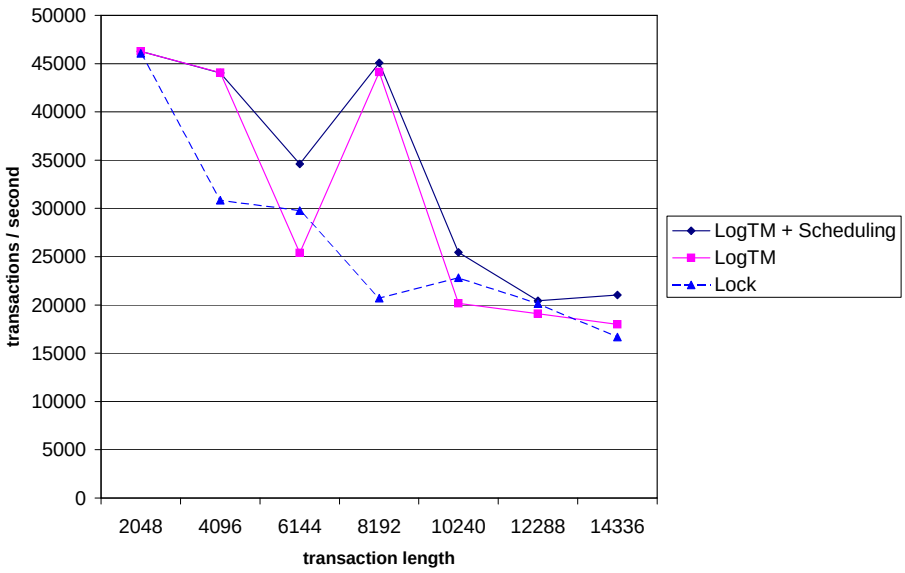

Figure 6: Throughput Variation on LogTM

Performance improvement of scheduler-enabled LogTM over the baseline LogTM can be explained with transaction effectiveness. Figure 7 shows the transaction effectiveness trend on both the scheduler enabled LogTM and the baseline LogTM.

When there exists enough parallelism, both the LogTM implementations show effectiveness close to 1. However, as the parallelism decreases, baseline LogTM's transaction effectiveness drops rapidly. While the baseline LogTM maintains the effectiveness value around 0.15, the scheduler enabled LogTM maintains effectiveness at 0.66. This shows that our simple transaction scheduling scheme can significantly increase transaction effectiveness. The difference in this transaction effectiveness results in performance improvement

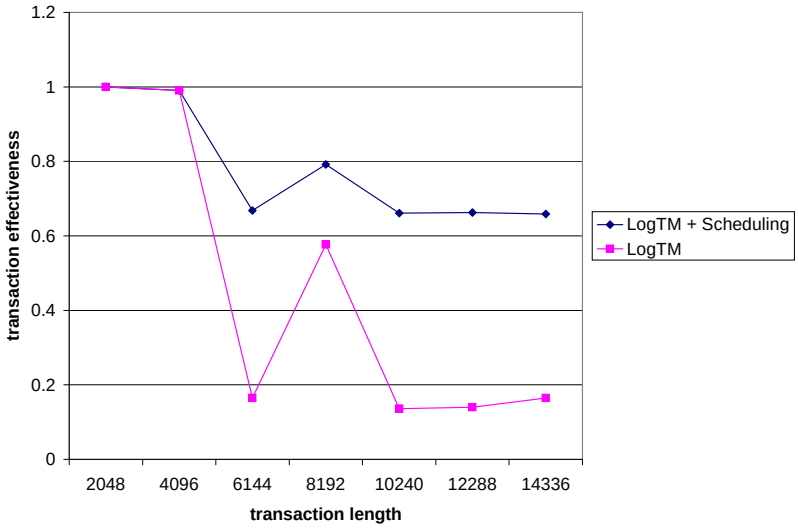

Figure 7: Transaction Effectiveness on LogTM

over the baseline LogTM.

Moreover, the fact that scheduling enabled LogTM significantly outperforms baseline LogTM at a certain transaction length means that the baseline version is launching excessive transactions for the given workload. For example, at transaction length of 6144, both from the Figure 6 and Figure 7 we can conclude that the baseline LogTM is squandering its resources on excessive transactions. To verify this we examined the behavior of the scheduling method as we decreased the thread count while fixing the transaction length at 6144. Figure 8 shows the result.

This figure clearly shows the effect of excessive transactions. As the thread count decreases, the number of excessive transactions decreases. Since the baseline LogTM does not spend its resources on those excessive transactions, the performance of the baseline quickly catches up the performance of scheduler enabled LogTM — at the thread count of 2, the performance becomes equal.

5.2 Transaction Scheduling on RSTM

5.2.1 Results on a 2-way SMP machine

To study the sensitivity of our contention intensity equation to the value of α, we executed the benchmarks with 3 different α values: $\alpha = 0.3$, 0.5, and 0.7. Throughout these experiments threshold was set to 0.5. Figure 9 summarizes the results.

In this figure y axis denotes the relative throughput over the baseline RSTM. Each data point denotes

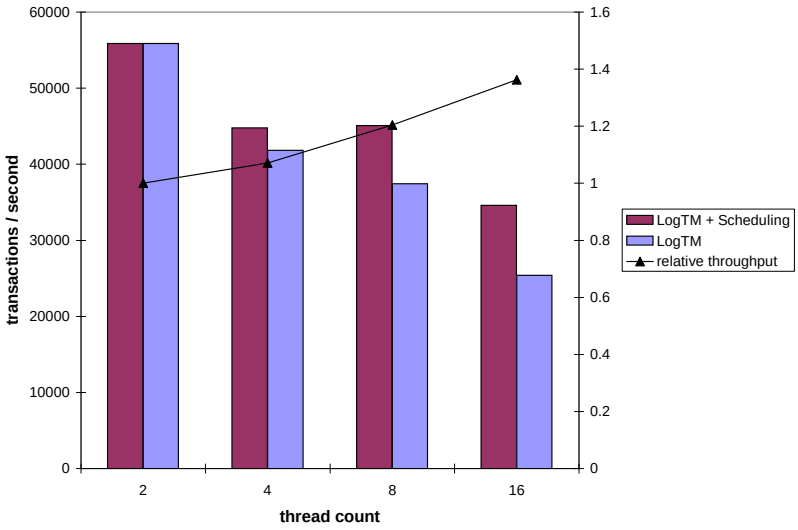

Figure 8: Effect of Thread Count on Transaction Scheduling

the harmonic mean of relative throughput calculated over 5 benchmarks. The x axis denotes the number of concurrent threads. Although the trend for the three α values gets entangled when significant number of threads execute (thread count ≥ 15), we can see that the $\alpha = 0.3$ case generally outperforms the other cases. This means that for these benchmarks weighing current contention information more than the past history brings about more performance improvement.

At this α value, the performance of scheduling enabled RSTM is then compared with the baseline RSTM and locks in Figure 10. The throughput for each benchmark was measured 3 times, and the average value was chosen as the representative value. In each of the subfigure, x-axis denotes the number of concurrent threads, while the y-axis denotes the throughput at log scale. Solid circles denote the throughput of RSTM with scheduler, squares denote the throughput of baseline RSTM, and triangles denote the throughput of locks. Note that unlike the hardware transactional memory systems, the performance of software transactional memory systems can be much worse than the locks since managing the per-object transaction information in software can cause significant overhead. This behavior has already been reported in the original RSTM paper [9] for a couple of benchmarks.

Our scheduling method shows great throughput improvement on RBTree, HashTable, and LFUCache, while showing comparable (RandomGraph) or slightly lower (LinkedList) performance in the other benchmarks. Figure 11 summarizes this information.

In the figure, the lower end of each vertical bar represent the minimum relative throughput of our scheduling method among the five benchmarks. In the same manner, the upper end of each vertical bar represent

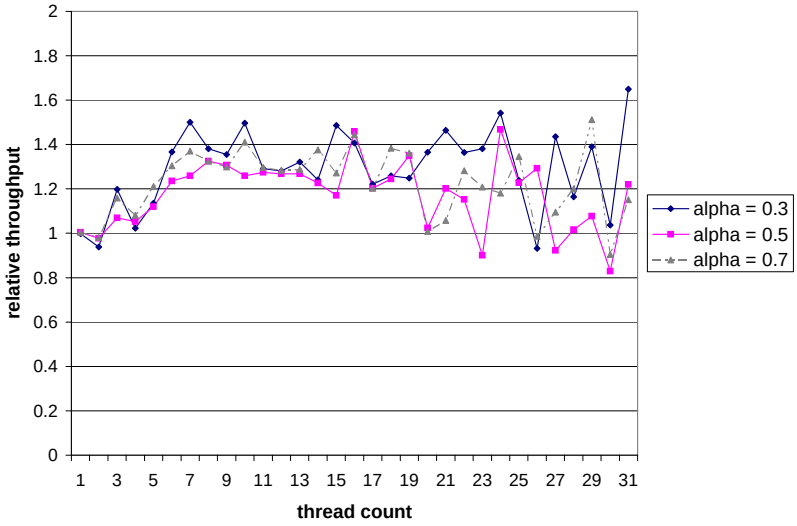

Figure 9: Sensitivity Study on 2-way SMP machine

the maximum relative throughput. The dotted line across these vertical bars represent the harmonic means of 5 relative throughputs for different thread counts, as the same in Figure 9. Notice that the aggregate performance improvement measured by harmonic mean is around $1.3 \sim 1.5$, while the maximum relative throughput can be as high as 5.9. The fact that those vertical bars usually locate in $y \geq 1$ region represents that our scheduling method brings net speedup over varying thread counts.

5.2.2 Results on an 8-way SMP machine

The same sensitivity study as in Section 5.2.1 was performed on 8-way SMP machine. Figure 12 shows the results. In this case, the $\alpha = 0.5$ case outperforms the others when sufficient number of threads (thread count ≥ 15) are executing.

The performance results on our 8-way SMP machine is shown in Figure 13. We attribute the widened gap between RSTM implementations and the lock to the slow FSB speed (100MHz) of our machine. Nonetheless, note the common trend in RBTree, HashTable, LinkedList, and RandomGraph; when the number of concurrent threads is small (thread count ≤ 15), the scheduler and the baseline version show similar throughputs. However, as the thread count increases, the scheduler-enabled RSTM starts to perform better than the baseline version. This trend is summarized in Figure 14.

As shown in the figure, when the number of concurrent threads is small, the relative throughput remains around 1. This is due to the fact that there is less contention when the thread count is smaller than the number of processors on our 8-way SMP machine. Nonetheless, as the contention rises with more concurrent threads, the scheduler-enabled RSTM shows performance improvement. The aggregate performance improvement

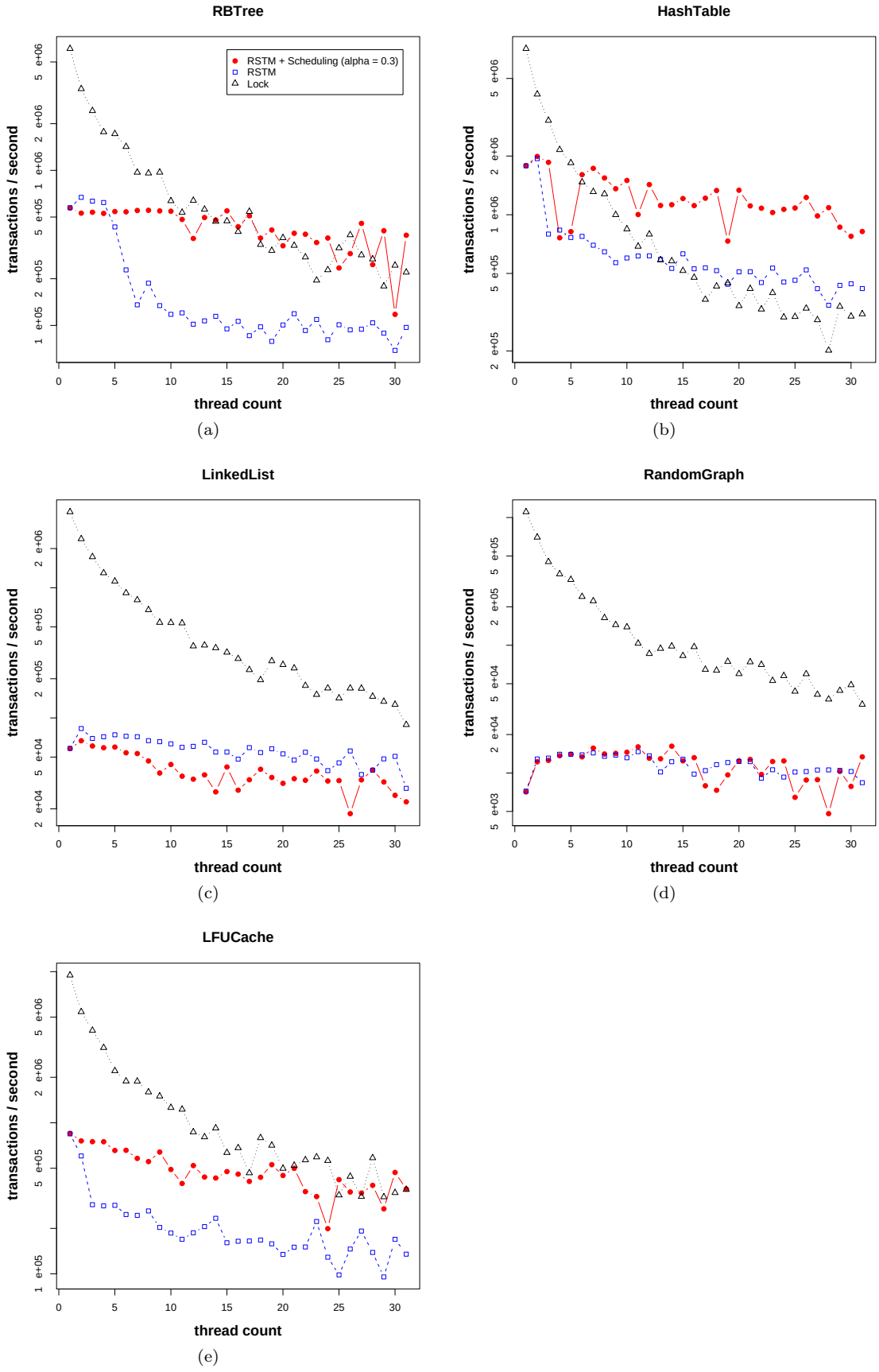

Figure 10: Individual Benchmark Result on a 2-way SMP machine

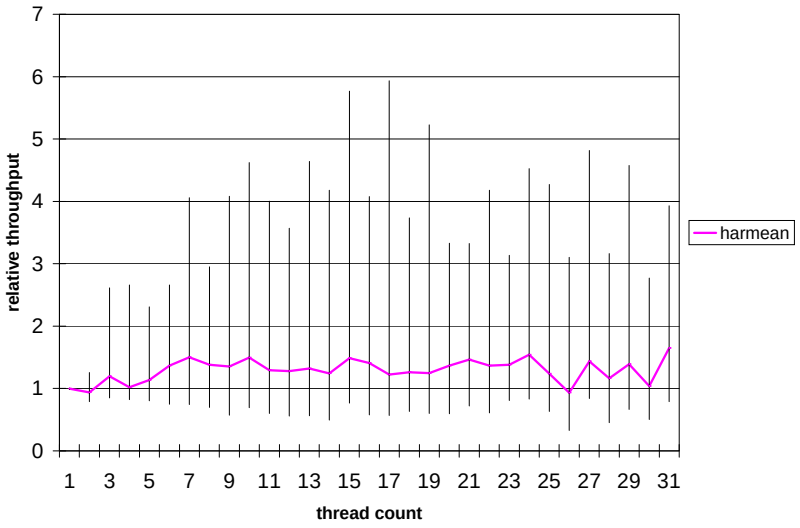

Figure 11: Relative Throughput Improvement over RSTM on 2-way SMP Machine

ranges from 1.1 to 1.4. Although there are some cases where minimum relative throughput goes below 1, the fact that those vertical bars locate higher than 1 tells that the scheduling mechanism results in an overall net performance gain. Considering that the hardware implementation of transaction scheduler would radically reduce the synchronization overhead, this performance improvement will be more significant.

6 Related Studies

Transactional memory [7] is one kind of approach to maximize parallel performance by speculative execution. Other approaches utilizing speculation also include Rajwar and Goodman's speculative lock elision [13] and speculative synchronization from Martnez and Torrellas [11]. Nonetheless, speculative methods potentially suffer from backfire when speculation fails frequently. Our paper minimizes this negative effect on transactional memory systems.

Other approaches to maximize the performance of transactional memory systems include contention managers [6, 15]. Contention managers try to maximize the performance by effectively handling the contention after it has been detected. Hardware support to utilize this information has been discussed in [16]. Rather than to take action after the contention has been detected, our method fundamentally reduces the contention itself.

There has been several hardware transactional memory implementations [7, 4, 1, 12]. A good summary of current status can be found in [12]. Software transactional memory systems have seen even more releases [6, 5, 3, 8, 9]. Among those, [6, 8, 9] incorporate contention manager approach. Our scheduling scheme can

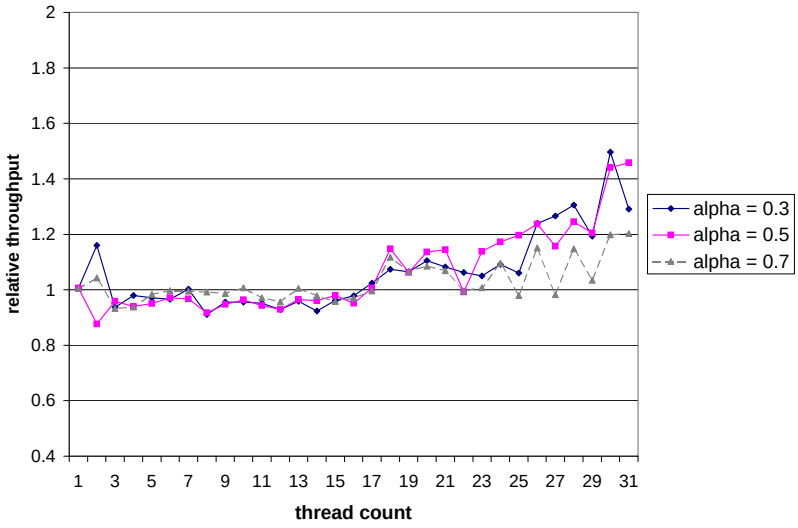

Figure 12: Sensitivity Study on 8-way SMP machine

be easily integrated into these systems.

7 Conclusion

In this paper we proposed the concept of adaptive transaction scheduling. First we have pointed out that there are certain situations where transactional memory systems perform worse than locks, and that these situations typically show less transaction effectiveness. With the parallelism feedback from the contention intensity detection mechanism we were able to significantly increase transaction effectiveness for those workloads that lack parallelism.

Based on this notion we devised a simple transaction scheduler. In this scheme the execution point of a transaction is dynamically controlled in an aim to maximally exploit the parallelism inherent in a given program phase. Through our case study we have shown that this scheduler not only guarantees that hardware transactional memory systems perform better than locks, but also significantly improves performance on both the hardware and software transactional memory systems. In our experiment the relative performance improvement on software transactional memory was 1.3x $\sim$ 1.5x, while the peak performance improvement reaching 5.9x.

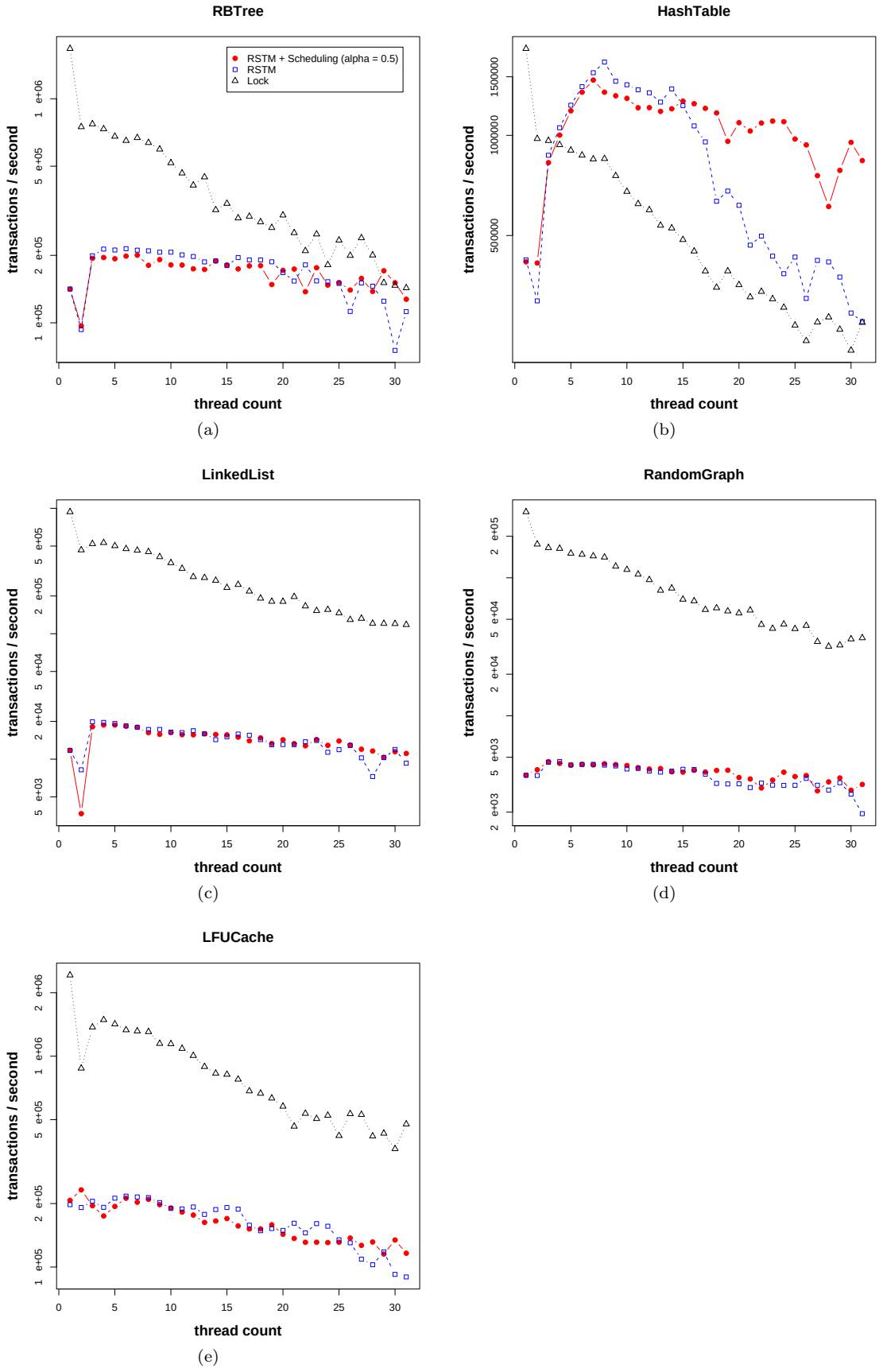

Figure 13: Individual Benchmark Result on an 8-way SMP machine

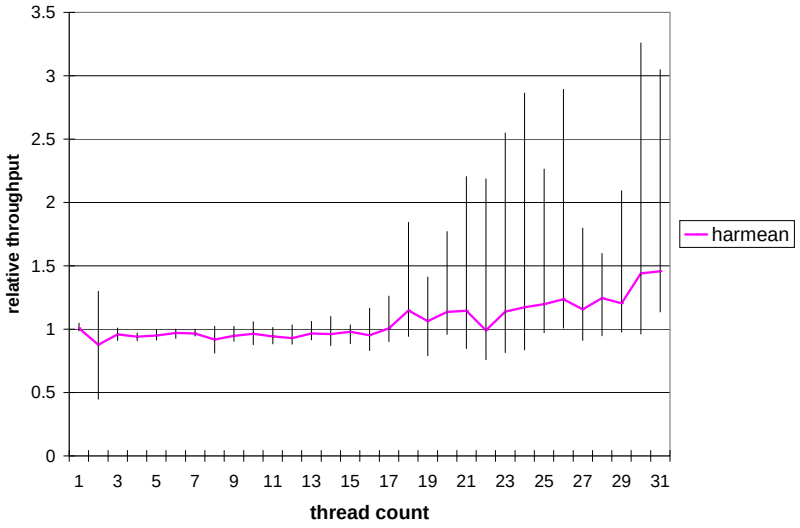

Figure 14: Relative Throughput Improvement on 8-way SMP Machine

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316 – 327, Feb 2005.

[2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, November 2005.

[3] K. Fraser. *Practical Lock-Freedom*. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, February 2004.

[4] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102 – 113. IEEE Computer Society, Jun 2004.

[5] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388 – 402, New York, NY, USA, 2003. ACM Press.

[6] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92 – 101, July 2003.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289 – 300. May 1993.

[8] V. Marathe, W. Scherer III, and M. Scott. Adaptive software transactional memory. In *Proceedings of the Nineteenth International Symposium on Distributed Computing*, 2005.

[9] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[10] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. In *Computer Architecture News (CAN)*, September 2005.

[11] J. Martnez and J. Torrellas. Speculative synchronization: Programmability and performance for parallel codes. In *IEEE Micro Top Picks from Microarchitecture Conferences*, 2003.

[12] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Conference on High Performance Computer Architecture*, pages 254 – 265, February 2006.

[13] 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*, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[14] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494 – 505. IEEE Computer Society, Jun 2005.

[15] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240 – 248, 2005.

[16] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.