Adaptive Transaction Scheduling for Transactional Memory Systems

Richard M. Yoo
yoo@ece.gatech.edu

Hsien-Hsin S. Lee
leehs@ece.gatech.edu

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332

ABSTRACT

Transactional memory systems are expected to enable parallel programming at lower programming complexity, while delivering improved performance over traditional lock-based systems. Nonetheless, there are certain situations where transactional memory systems could actually perform worse. Transactional memory systems can outperform locks only when the executing workloads contain sufficient parallelism. When the workload lacks inherent parallelism, launching excessive transactions can adversely degrade performance. These situations are likely to become dominant in future workloads when large-scale transactions are frequently executed. In this paper, we propose a new paradigm called adaptive transaction scheduling to address this issue. Based on the parallelism feedback from applications, our adaptive transaction scheduler dynamically dispatches and controls the number of concurrently executing transactions. In our case study, we show that our low-cost mechanism not only guarantees that hardware transactional memory systems perform no worse than a single global lock, but also significantly improves performance for both hardware and software transactional memory systems.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Design, Performance

Keywords

Contention Intensity, Parallelism, Performance, Transaction Effectiveness, Transactional Memory Systems

1. INTRODUCTION

Due to its radically simple programming semantics, a transactional memory (TM) system [11, 12] has loomed as a promising parallel programming model for the emerging multi-core platforms. Especially for hardware transactional memory (HTM), programmers do not need to worry about the exact data conflict information. Rather, a programmer can speculatively mark a potentially conflicting code region as a transaction and let the underlying TM implementation to guarantee the sequential correctness among transactions. In addition, by speculatively executing more than one transaction in a critical section, a TM system can theoretically achieve higher performance by exploiting parallelism inhibited by a conventional lock-based system. Unfortunately, due to its implementation complexity, most HTM research thrusts mainly focus on implementation issues with less attention to the performance [7, 1, 22, 19, 6, 17, 3].

Software transactional memory (STM) researchers, on the contrary, were well aware of the performance issues on TM. Recently, STM researchers proposed the concept of the *contention manager* [10, 14, 23], a user-level code module that enforces priority among transactional accesses. The priority could be determined by several indicators, such as a timestamp-based age of a transaction, the amount of work done based on memory footprint, etc [10, 23]. When a transaction encounters a conflict, it consults its contention manager. With the priority and other transactional information, the contention manager heuristically evaluates and decides whether aborting the offending transaction will improve the overall throughput. When the contention manager decides not to abort the offending transaction, the transaction consulting the contention manager will use a delay back-off, thus allowing the offending transaction to finish before the requestor retries the NACKed permission. By varying the number of back-off attempts and their intervals, the contention managers can significantly reduce the number of transaction aborts [23]. An STM implementation employing a contention manager has shown reasonable performance improvement over one without. It also demonstrates the capability of avoiding livelocks [10, 13, 14].

However, contention managers have their limitations. First, contention managers are fundamentally reactive, for their policies are enforced only after a conflict is detected. When a transaction invokes its contention manager due to a conflict, it cannot wait indefinitely. Since the transaction has already started executing a critical section, the situation would be equal to a deadlock if it indefinitely waits for the contention to disappear. Hence, contention managers are only good at resolving imminent contention; the offending transaction should commit very soon, or it would be forced to abort. Once the contention disappears by aborting the opponent, contention managers have no control over when the aborted transaction should resume. This is rather myopic, since the aborted transaction can restart immediately, conflicting with other transactions again. Contention managers only cure the outcome of the contention; they do not reduce the cause of the contention itself.

Secondly, contention managers are accessed frequently. Depending on the implementation [14], they are called whenever 1) a transaction starts, aborts, or commits, 2) a transaction acquires an object, 3) a transaction reads / writes an object (to collect information), or 4) a conflict is present (to enforce priority). Prior research has shown that contention management code itself accounts for a large portion of the execution time when contention traffic is high [26].

SPAA'08, June 14–16, 2008, Munich, Germany.

For this reason, contention managers tend to be distributed; they detect and resolve conflicts in a per-transaction manner. This nature significantly limits the capability of a contention manager to maintain a system-wide coherent view of contention, and also limits the conflict resolution scheme to heuristics.

Lastly, since contention managers are user-level code modules, it is difficult to incorporate a contention manager into an HTM implementation. To do so, hardware would have to trap into software for each transactional event to collect transactional information. Otherwise, an HTM could maintain the transactional information in some architectural registers and expose them to contention managers, but it would be hard to define a generic contention manager interface given that different applications benefit from different types of contention management schemes [10, 13, 14].

To tackle these problems, in this paper, we propose a new technique called *adaptive transaction scheduling* (ATS). In ATS, a *scheduler* adaptively controls the number of concurrent transactions executing in critical sections based on the contention feedback from the application. This is done by selectively scheduling those transactions that tend to abort frequently.

The scheduler is designed such that it is consulted only when a transaction starts under high contention. This infrequent access allows the scheduler to be implemented as a centralized module, thereby enabling an advanced and coherent system-wide scheduling scheme. Although it is a centralized scheme, under HTM it does not suffer from scalability issues due to its adaptive and lightweight nature. The scheduling scheme has no adverse effect on performance when contention is low, and will mitigate performance degradation as contention grows.

Based on this framework, we designed a low-cost scheduler that can be easily integrated into either an HTM or STM. For the HTM, we observed that ATS not only improves performance by reducing the number of transaction aborts, but also improves the *quality* of transactions. We also show that our scheme guarantees a performance lower bound when coarse-grained lock (CGL) critical sections are transformed into transactions. For STM, we demonstrate that ATS delivers significant performance improvement while acting as a QoS safety net under an oversubscription configuration.

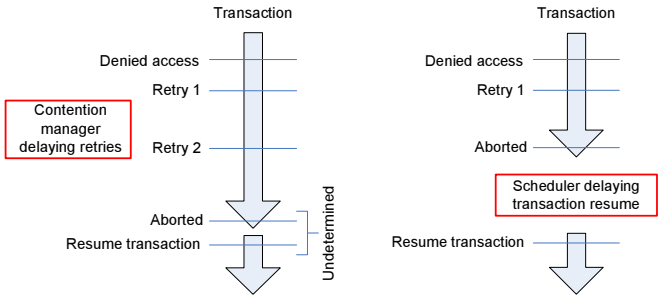
To the best of our knowledge, this paper is the first to incorporate transaction scheduling on TM systems to adaptively exploit the maximum parallelism. The rest of the paper is organized as follows. Section 2 describes our scheduling framework. Section 3 and Section 4 discuss our implementation on LogTM and RSTM and analyze their performance. Related studies in TM area are discussed in Section 5. Finally, Section 6 concludes.

2. TRANSACTION SCHEDULING

In our execution model, a thread enters and leaves multiple *critical sections* throughout its lifetime. Upon *entering* a critical section, the thread starts a *transaction*. The thread might resume the transaction multiple times if the previous transaction aborts. A thread *leaves* the critical section when it commits the transaction.

Figure 1 highlights the difference between our transaction scheduling approach and the contention manager approach. As shown in Figure 1(a), a contention manager tries to reduce the contention by adjusting when to retry the denied object (*e.g.*, a cache line). Typically, a contention manager employs an exponential backoff scheme with retry interval expanding exponentially to a maximum limit until success. A contention manager can decide to abort a certain transaction, but it does not deal with *when* to resume an aborted transaction.

In contrast, our transaction scheduling scheme in Figure 1(b) specifically deals with *when* to resume the aborted transaction. It dynamically schedules the point where an aborted transaction resumes its execution. To say that a transaction scheduler uses an exponential backoff scheme means that an aborted transaction resumes with an exponentially increased interval to a maximum limit. As can be seen, these two approaches are orthogonal, dealing with different aspects of TM's characteristics.



(a) Contention Manager (b) ATS

Figure 1: Contention Manager versus ATS

We do not schedule all the transactions. Transactions resort to the scheduler only when they encounter high contention. To detect when to schedule a transaction, we introduce a dynamic metric called *contention intensity*.

2.1 Contention Intensity

The *effectiveness* of a transaction is closely related to the intensity of the contention a transaction encounters during its execution. By limiting the number of concurrently executing transactions at a given time, we can dynamically control the contention intensity. Once the contention intensity is kept below a desired level, we can significantly increase the transaction effectiveness, thereby improving the overall system performance.

Contention Intensity can be detected in either a centralized or decentralized manner. A centralized detection scheme relies on a global module to 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 used the decentralized scheme. To quantify the contention intensity, we define the contention intensity as a dynamic average based on current available contention information. Each thread maintains its contention intensity (CI) in the following manner:

$$CI_n = \alpha \times CI_{n-1} + (1-\alpha) \times CC$$

Maintaining contention intensity information enables a parallelism feedback mechanism for a TM system. Initially, CI is set to 0. This equation is then evaluated whenever a transaction commits or aborts, based on the previous CI and the current contention (CC). In this equation the CC term is set to 0 when a transaction commits, and set to 1 when a transaction aborts. The weight variable, α, determines which portion of 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. When the α value is small, the current contention information is reflected more quickly. In fact, based on the past commit / abort history, the contention intensity *predicts* the likelihood that a transaction would face another conflict.

By default, the contention intensity value should be reset to 0 when a thread changes the entered critical section (which starts at a different PC). To put it differently, when a thread loops around the same single critical section, the contention intensity is not reset. Nonetheless, we observed that resetting the contention intensity does not have a significant impact on performance, since by the time all the threads leave a particular critical section — *i.e.*, do not re-enter that particular critical section before entering another critical section — each thread's contention intensity is already close to 0 due to the phased behavior of multi-threaded applications.

2.2 A Low Cost Transaction Scheduler

In our ATS scheme, aside from the OS thread scheduler, we implement a transaction scheduler directly inside a TM system. To utilize the scheduler, each thread maintains its own contention intensity as described in Section 2.1. When a thread either begins

a transaction or resumes a transaction after abort, it compares its contention intensity with a designated threshold. When the contention intensity is below the threshold, the thread begins a transaction normally. Otherwise, the thread will stall and report to the scheduler, awaiting a dispatch. The scheduler will then signal back the thread to proceed once the thread is ready to restart. Therefore, when the contention intensity is low, ATS has little effect except for the penalty of intensity check. A thread will no longer consult the scheduler and begin a transaction normally when the contention intensity subsides below the threshold.

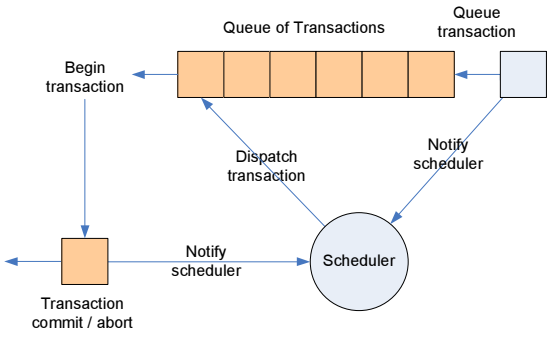


Figure 2: A Queue-Based Transaction Scheduler

Figure 2 shows the organization of a queue-based scheduler. This scheduler maintains a single centralized queue of transactions, which resembles the run queue found in a regular OS thread scheduler. This queue dispatches one single transaction at a time.

If a transaction is at the head of the queue, and if no other transaction dispatched from the queue is still executing, it is dispatched immediately. Otherwise, the transaction will wait in the queue until the exclusivity is met. Moreover, a transaction that was dispatched from the queue must notify the scheduler when it commits or aborts. This will trigger the dispatch of the next transaction.

Note that this queuing behavior effectively serializes high contention transactions. At one extreme, when all the transactions are queued, this mechanism gracefully degenerates transactions into a lock. With a 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 single coarse-grained lock.

It is noteworthy to point out that we strived to keep the design of the scheduler as simple as possible. This way, the scheduler could be easily implemented in an HTM system. Simplicity is not necessarily bad when it does show significant performance improvement, as we will demonstrate later.

2.3 Transaction Scheduler in Action

In Figure 3 we use an example to detail the behavior of a transaction scheduler. Let us first assume that there is only one critical section throughout the entire program with only one global transaction queue. In this figure, the timeline flows from top to bottom; on the right side locates the hypothetical variation of the contention intensity over time (an average of all CIs from running threads). When the contention intensity is below the threshold (Timeline 1), transactions begin execution without resorting to the scheduler. As the contention intensity grows beyond the threshold, some transactions start to report themselves to the queue managed by the scheduler (Timeline 2). The scheduler dispatches only one single transaction at a time from the queue, which effectively reduces the number of concurrently executing transactions. At Timeline 3, as more transactions are queued and serialized, the contention intensity starts to decrease. Once the contention intensity of a transaction drops below the threshold, it will begin transactions without consulting the scheduler, exploiting more parallelism (Timeline 4, 5). Although demonstrated with one average CI for all running threads in this figure, each thread actually maintains its own CI. This design also prevents threads from exhibiting abrupt group behavior.

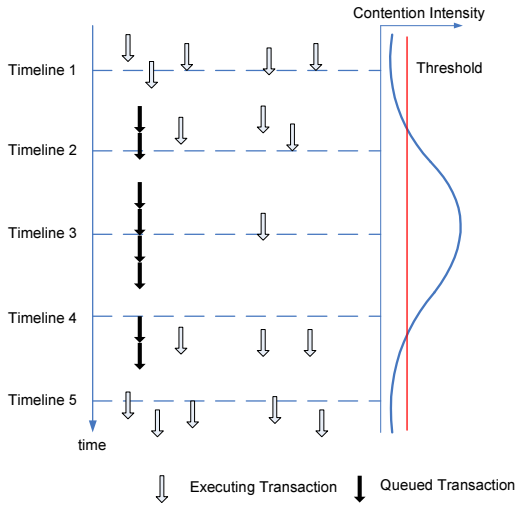


Figure 3: Behavior of a Queue-Based Scheduler

The scheduler adaptively changes the number of concurrently active transactions, so that the contention will not increase without bound (livelock). In essence, the scheduler tries to keep the number of concurrent transactions close to the maximum number of data parallel transactions, while the contention intensity acts as an error signal. Since the contention intensity is updated dynamically, this scheme can also adapt to phase changes during the execution. In other words, the scheduler will exploit the maximum inherent parallelism at any given phase.

Now let us consider the case where there are multiple critical sections starting at different PCs. When we maintain a dedicated queue for each critical section, the scheduler can control the number of concurrent transactions in each of the critical sections. On the contrary, when we maintain a single queue for all critical sections, the scheduler would control the number of concurrent transactions executing in any of the critical sections. Due to the phased behavior of multi-threaded programs, however, we noticed that threads usually enter and leave a critical section at roughly the same time. Therefore, at any given point of execution, the case of different threads executing different critical sections was rather rare, and a single global queue for all critical sections will be sufficient. This mimics current TM systems which do not differentiate transactional accesses from different critical sections. Throughout the rest of this paper, we only focus on the case where we have a single global queue for all the critical sections.

2.4 Comparison with Contention Manager

This ATS approach is completely different from the contention manager approach. First, the two approaches take effect at different points of transaction execution. A contention manager takes effect *after* a transaction has started; it is invoked when there is a conflict. ATS takes effect *before* a transaction starts executing, to reduce the potential contention. For example, upon discovering transactions A and B conflict, a contention manager could stall transaction B to resolve the conflict. However, it cannot prevent another transaction C from starting execution although it is highly likely that it will induce yet another conflict.

Secondly, our ATS scheme differs from the contention manager approach in the frequency of module accesses. To collect transactional information, contention managers are called whenever there is a transactional update; this forces contention managers to be distributed. In contrast, since our transaction scheduler is accessed only when a transaction starts under high contention, it can be centralized. This centralization enables advanced, coherent scheduling policies. Controlling the number of concurrent transactions is only possible when we have a global view of the contention. Due to its adaptive nature, a centralized ATS does not undermine the scalability of an HTM system.

Lastly, infrequent access and a simple design of ATS enables its low-cost integration with HTM, as we will describe in Section 3. More importantly, contention managers can only be implemented on obstruction-free TM systems [9, 10]; largely due to this assumption, contention managers can resolve conflicts on a peer-to-peer basis without the system-wide contention information. In contrast, ATS can be implemented on other types of TM systems (*e.g.*, lock-free) as well.

In fact, ATS is a complementary technique to the contention manager approach as they address different properties of a TM system. We can say that ATS performs *macro scheduling* to orchestrate when to start a transaction based on mutual contention information collected, and after a transaction has started contention manager will perform *micro scheduling* to reduce contention on the fly.

3. HTM INTEGRATION

To evaluate the advantages of ATS, we implement our proposed queue-based scheduling scheme in both HTM and STM systems. This section details our ATS implementation on LogTM [19] (an HTM system), and Section 4 discusses our implementation on RSTM [14] (an STM system).

3.1 LogTM Settings

For an HTM system, we implemented ATS on LogTM [19]. LogTM has been released as a memory timing module of the GEMS simulator [15][1]. LogTM contains a dedicated module, the transaction manager, which is accessed whenever a transaction starts, aborts, or commits. We implemented our scheduling algorithm so that this transaction manager maintains the contention intensity information.

We assume that the hardware queue resides in a central location of the system. Since our implementation supports one active transaction per CPU, the queue depth amounts to the total number of CPUs on the system. For those HTM systems that support more than one transaction per CPU, the system could fall back to contention manager-only approach when the queue is full. When a CPU decodes a transaction_begin instruction, it compares the current contention intensity value with the threshold. When the contention intensity is above threshold, the CPU generates a signal directed to the scheduler asking for intervention; at the same time, it stalls the thread. When the queued transaction becomes ready for execution, the scheduler signals back the CPU to start the transaction. We assigned a 16-cycle delay for the signal to propagate from a CPU to the global queue, and another 16-cycle delay for the queue to notify the CPU to proceed. In our experiments, however, the actual value of the latency did not affect the performance to a significant degree since a queued transaction typically waits for hundreds to thousands of cycles. Table 1 specifies the simulated machine in GEMS.

The simulated system uses only the Ruby memory timing model of the GEMS simulator. Thus, the CPU simulates a single-issue, in-order 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 20 cycles that is taxed for each block of log written back to the memory. By default, LogTM's contention management scheme is *stalling* [19]; the NACKed transaction keeps retrying the access with a fixed time interval unless it detects a possible deadlock situation. Our ATS scheme was built on top of this contention manager.

[1]The delayRestart feature of LogTM had to be replaced with exponential backoff since it caused deadlock in rare situations. This means that in the baseline LogTM, aborted transactions will automatically resume execution with exponentially increased time interval to a maximum limit.

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	
m_abortStartupDelay	40 cycles
m_abortPerBlockDelay	20 cycles

Table 1: Simulation Settings for LogTM

3.2 Benchmark Suite

Our benchmark suite includes selected SPLASH-2 applications [28] and a modified Deque microbenchmark included in the LogTM release. These workloads are transactionized by replacing the locks with transactions. Table 2 lists the benchmark suite. We use the same subset of SPLASH-2 applications used in the original LogTM paper [19]. Those omitted ones are not considered as representative TM applications since their critical sections mostly perform trivial memory operations, *e.g.*, single induction variable increments.

Workload	Input Set	# Threads
Water-nsquared	512	15
Water-spatial	512	15
Ocean (contiguous_partitions)	258	8
Raytrace	teapot	15
Cholesky	14	15
Barnes	512 bodies	15
Radiosity	test	15
Deque	N/A	15

Table 2: LogTM Benchmark Suite

For the Deque microbenchmark, each transaction first enqueues (dequeues) a value on the left (right) of a global deque. It then performs a local job, and increments the global counter at the end. 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 potential parallelism. When comparing the performance to a lock-based implementation, BEGIN_TRANSACTION and END_TRANSACTION macros were substituted with pthread_mutex_lock() and pthread_mutex_unlock() to a single global lock, respectively. Despite its small size, this microbenchmark heavily stresses the transaction scheduling and the contention management scheme of the underlying TM system. As more TM systems become available, we expect this type of coarse-grained, frequent transaction will become more prevalent [18].

In all of these benchmarks, one of the 16 CPUs was dedicated for the OS to prevent the kernel thread from preempting application threads. Hence, most of the workloads were executed with 15 threads. Benchmark Ocean was executed with 8 threads, for it requires the number of threads to be power of two. Moreover, since LogTM did not support thread migration, thread affinity was fixed so that each thread executes on a single CPU. Throughout the experiments, α was fixed to 0.7, while the threshold was fixed to 0.5.

Benchmark	Execution Time		Xact Begin		Commit		Abort (%)		Xact Latency (stdv)		L1D MPKI	
	base	ATS	base	ATS	base	ATS	base	ATS	base	ATS	base	ATS
Water-nsquared	52383081	52383081	17664	17664	17664	17664	0 (0%)	0 (0%)	1548 (315)	1548 (315)	2.70	2.70
Water-spatial	65462563	65462563	285	285	285	285	0 (0%)	0 (0%)	501 (4764)	501 (4764)	1.25	1.25
Ocean	291813916	291813916	1666	1666	1664	1664	2 (0.1%)	2 (0.1%)	849 (279)	849 (279)	2.33	2.33
Raytrace	50333882	50852127	48654	48128	47782	47781	872 (1.8%)	347 (0.7%)	6857 (37280)	6332 (11499)	6.73	6.67
Cholesky	22596229	22258328	6754	6550	5938	5935	816 (12.1%)	615 (9.4%)	1553 (4824)	1174 (2720)	1.10	1.11
Barnes	25015887	23878230	3055	2575	2319	2319	736 (24.0%)	256 (10.0%)	9326 (74878)	2245 (4736)	0.96	0.95
Deque-14436	24037196	20970633	3713	1783	1200	1200	2513 (67.7%)	583 (32.7%)	129541 (143462)	39045 (35107)	5.06	2.32
Deque-2048	20081574	13783990	3492	1866	1200	1200	2292 (65.2%)	666 (35.7%)	72857 (102182)	10641 (8948)	2.50	1.54
Radiosity	472253209	239312955	490658	336154	276917	278711	213741 (43.6%)	57443 (17.1%)	16488 (60769)	1738 (4975)	12.96	3.99

Table 3: Execution Time Statistics on LogTM

3.3 LogTM Result Analysis

3.3.1 Execution Time Characteristics

Table 3 shows a variety of execution time statistics gathered for the parallel sections of each benchmark. For each category, we show the numbers for the baseline LogTM (base) and the LogTM enhanced with ATS. Also, we experimented two Deque scenarios denoted by Deque-14436 and Deque-2048 that set their transaction_length parameters to 14436 and 2048 memory operations, respectively.

The most prominent effect by ATS is the reduction of execution time. Figure 4 shows the speedup over the baseline LogTM by measuring the parallel sections of the program. Based on the application characteristics, we divided the applications into three groups: low-contention, medium-contention, and high-contention workloads. Figure 5 shows the transaction abort rate for each application.

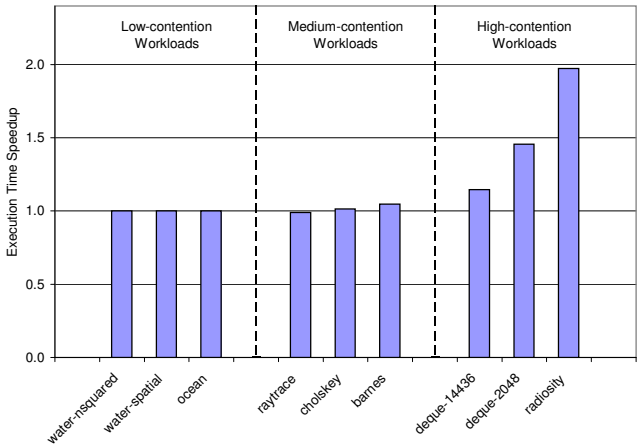


Figure 4: Execution Time Speedup

Not surprisingly, low-contention workloads exhibit zero or negligible abort rates. As explained in Section 3.2, critical sections performing simple induction variable increments are the most common in this category. For this type of workload, the scheduler has neither positive nor negative effect. Hence, the execution time remains the same.

With the medium-contention workloads, the abort rates become more noticeable. Raytrace, Cholskey, and Barnes belong to this category. ATS shows marginal performance effect in these cases. As shown in Figure 4, Cholskey and Barnes show 2% and 5% speedup respectively. For Raytrace, the chosen α value (0.7) turned out to be too conservative such that the scheduler overly serialized transactions, resulting in 1% slowdown. Nevertheless, the scheduler significantly reduces the transaction abort rate for all three workloads. Note that the Xact Begin column of Table 3 shows the baseline starting transactions in excess but committing (Commit column) nearly the same amount of transactions as the ATS-enabled LogTM. In those workloads that rely on convergence as their termination condition, the number of committed transactions could be slightly different from the baseline case since ATS changes the application behavior.

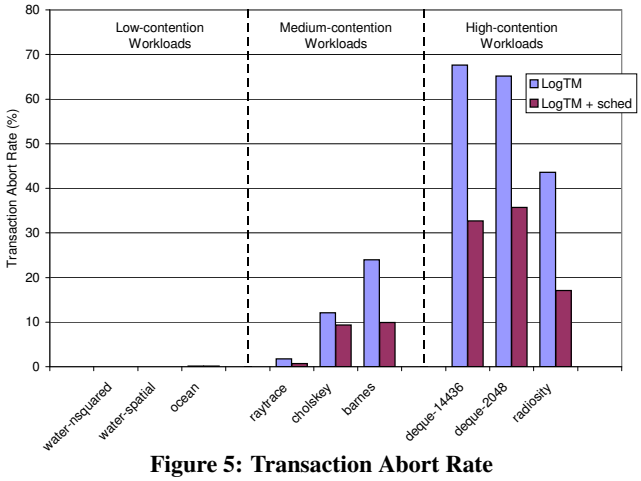


Figure 5: Transaction Abort Rate

ATS shows a huge benefit when running high-contention workloads. Both Deque microbenchmark programs show 15% and 46% speedup respectively, while Radiosity is improved by 97%. As also shown, the scheduler nearly halves the transaction abort rates. In addition, Table 3 indicates that for high contention workloads the baseline issues 50% to 100% more transactions than the ATS-enabled LogTM. Aside from performance advantages, reducing the number of aborted transactions would also improve power consumption and cache pollution when thread affinity is not applied.

3.3.2 Improving the Quality of Transactions

Not only does ATS reduce execution time and transaction abort rate, it also improves the *quality* of each transaction. The first of such characteristic is the *transaction latency*, *i.e.*, the number of cycles of a committed transaction's lifetime. In LogTM, when there is a contention, it does not abort the offending transaction right away. It stalls the offending transaction until the memory request can be satisfied [19]. Hence higher contention typically leads to a longer transaction latency. While stalling for the opponent, the stalled transaction graduates no useful instructions. It simply squanders CPU cycles and energy. Figure 6 shows the normalized average transaction latency for the committed transactions. For example,

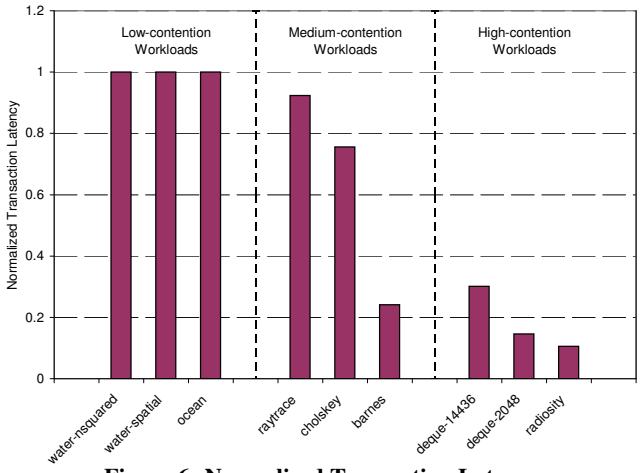


Figure 6: Normalized Transaction Latency

the transaction latency of Radiosity was reduced down to around 10% of the baseline. Moreover, not only does our scheduler reduce the *average* of transaction latency, it also reduces the *standard deviation* of transaction latency. The stdv values in Table 3 show this trend. The implication is that the scheduler not only shortens the transactions, but also makes them more deterministic and predictable, something expected from a "high-quality scheduler."

Combined with the results from Section 3.3.1, the ATS mechanism demonstrates fewer CPU cycles are wasted while performing the same amount of work, leading to performance improvement and energy savings. Moreover, under a multitasking OS, the overall throughput could be improved by context-switching to a thread of a different process while ATS delays resuming a transaction[2].

Nevertheless, as explained by Amdahl's law, the transaction latency reduction cannot always be translated to a proportional speedup. For example, in Barnes the amount of time when there is at least one transaction executing was less than 30% of the total execution time. Further, we found that the execution time of Barnes is usually dominated by only a few long transactions. As such, even though the latencies of the majority of the transactions were reduced, the overall execution time did not decrease much as long as the execution times of those long transactions were not reduced. We expect our scheduler to perform better when the lengths of the transactions are of uniform duration.

The second aspect of the quality of a transaction is observed from the cache miss rate. Upon each transaction abort, TM implementations that keep speculative results in caches (eager version management [19]) must invalidate buffered results following the cache coherence protocol. Frequent aborts amount to more cache line invalidations which lead to a higher cache miss rate when a transaction resumes. The L1D MPKI column in Table 3 shows the L1D cache Misses Per Kilo Instructions. As expected, we can see that high-contention workloads benefit from our technique.

3.3.3 Guaranteeing Performance Lower Bound

One way to obtain a TM workload is to convert critical sections guarded by coarse-grained locks into transactions. This amounts to expanding the contention scope of threads, since threads that were contending on different locks will now contend with each other. This contention scope is similar to that where all the critical sections are synchronized by a single global lock.

Due to its queue-based nature, ATS would serialize most of the transactions under extreme contention. This essentially degenerates its behavior to that of a single global lock. In other words, 1) ATS can guarantee a performance lower bound for workloads that were obtained by transforming coarse-grained lock critical sections into transactions, and 2) the performance in this situation will be comparable to the case where all the critical sections are synchronized by a single global lock.

TM implementations that detect conflicts at commit time (lazy conflict detection) can guarantee a similar performance lower bound since at least one transaction would commit at a single commit phase [7]. TMs that detect conflicts at object acquisition (eager conflict detection), unfortunately, cannot guarantee such a bound as transactions can repeatedly abort each other under high contention. ATS gives a performance lower bound to such eager conflict detection TMs. Considering that most of the current TM workloads are generated by the aforementioned approach, this performance guarantee would be necessary.

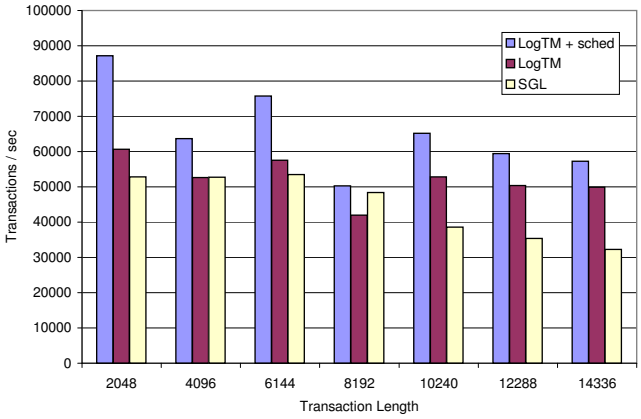


Figure 7: Throughput on Deque Microbenchmark

Figure 7 shows the throughput variation of Deque microbenchmark with varied transaction lengths. The longer the transaction length is, the harder to parallelize the workload. Overall, we see a gradual decrease in throughput for all three schemes as transaction length increases. The baseline LogTM occasionally exhibits worse performance compared to the single global lock (SGL). On the contrary, ATS-enabled LogTM always shows better or on-par performance with respect to the single global lock.

This serialization is particularly well suited for those HTM implementations that stall all other transactions in favor of one overflown transaction or a transaction performing I/O or a system call [3]. In our scheduling framework, stalling all other transactions amounts to forcing all transactions to report to the scheduler, with the overflown (syscall) transaction positioning at the head of the queue.

4. STM INTEGRATION

4.1 RSTM Settings

As for the STM, we implemented our ATS on top of the RSTM framework from the University of Rochester [14]. RSTM is a C++ TM 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_TRANSACTION macros, all the accesses through the read / write method of this wrapper object are treated as transactional reads / writes. Managing these transactions are completely handled by a software library.

We keep the contention intensity information in each thread's local storage where RSTM keeps each transaction's transaction descriptor. Moreover, the access to the central scheduling queue was serialized with a single global lock, and the conditional variables found in the pthread library were used to synchronize the communication between the scheduler and transactions. For proper synchronization, each conditional variable was again guarded with a local lock. Compared to the baseline RSTM, the performance of our scheduler implementation is actually penalized since the global

[2]We could not obtain performance results for such cases since a transaction in the baseline LogTM cannot survive a context switch.

lock and the locks guarding conditional variables introduce synchronization overheads in scheduling.

To quantify the performance, we measured the throughput of the entire system with 5 microbenchmark programs from the RSTM library: RBTree, HashTable, LinkedList, RandomGraph, and LFUCache, which are the common benchmarks repeatedly used in STM literature [23, 14, 26]. The contention manager *Polka* [23] was used as the default in our experiments. Polka implements a mixture of exponential back-off and memory footprint-size based priority mechanism. RSTM also allows programmers to configure 1) the visibility of the read-only transactions to other transactions, and 2) the conflict detection mechanism (eager or lazy). We selected the best configuration for each workload [14]. Namely, RBTree, HashTable, and LinkedList were executed with (invisible, eager) configuration while RandomGraph and LFUCache were executed with (visible, lazy) configuration. Our ATS-enabled library was implemented on top of the same contention manager using the same configurations. When comparing the throughput with the lock-based implementation, we used the cgl library included in RSTM release which transforms transactions back into critical sections guarded by a single global lock.

We measured the throughput on two real 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

4.2 RSTM Result Analysis

4.2.1 Results on a 2-way SMP Machine

We first performed a sensitivity study on α, resulting in the values α = 0.3 and threshold = 0.5 which showed the best average performance to demonstrate the results of our ATS scheme. In Section 4.2.4, we will discuss an algorithm that can self-tune the α value within the ATS.

Figure 8 shows the performance (in transactions per second) of our ATS-enabled RSTM versus the baseline RSTM and the locks by varying the number of threads. In each subfigure, the x-axis plots the number of concurrent threads, while the y-axis plots the throughput at log scale. Unlike the HTM systems, the performance of STM systems can be much worse than locks since managing the per-object transaction information in software causes significant overheads. As shown, ATS improves the throughput substantially over the baseline RSTM for RBTree, HashTable, and LFUCache, while showing almost on-par (RandomGraph) or slightly lower (LinkedList) performance in the others.

Figure 9 summarizes all the RSTM experimental results on the 2-way SMP machine. In the figure, the lower end of each vertical bar represents the minimum relative throughput of our scheduling method for the five benchmarks. In the same manner, the upper end of each vertical bar represents the maximum relative throughput. The line across these vertical bars represents the harmonic mean of 5 relative throughputs for each thread count. The aggregate performance speedup in harmonic mean is around 1.3x $\sim$ 1.5x, while the maximum relative throughput can be as high as 5.9x. The vertical bars skewing toward the y $\geq$ 1 region indicates the effectiveness of our ATS scheme in performance across the different number of threads used.

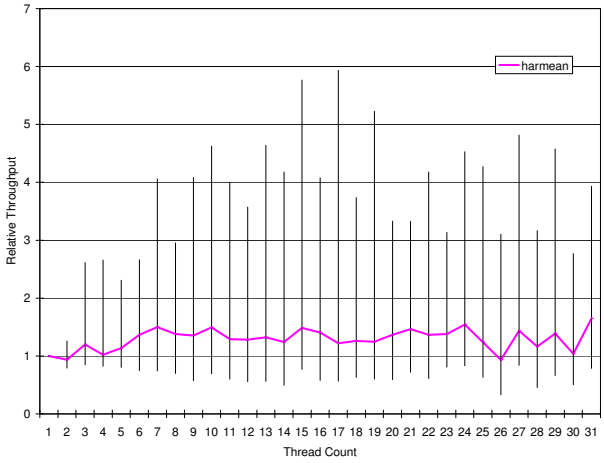


Figure 9: Scheduler Performance on a 2-way SMP Machine

4.2.2 Results on an 8-way SMP Machine

We performed the same sensitivity study and measured their results for an 8-way SMP machine. In this case, we selected $\alpha = 0.5$ as it outperformed the others when a sufficient number of threads (thread count $\geq$ 17) were executing. With $\alpha = 0.5$, a new thread will encounter two consecutive aborts before resorting to the scheduler. Due to the space limitation, we only summarize the performance results in Figure 10.

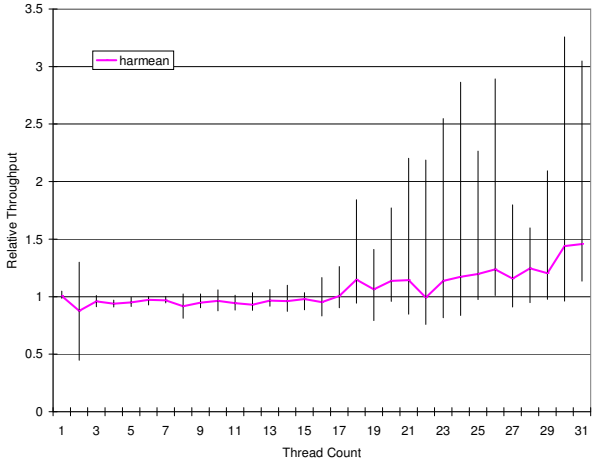
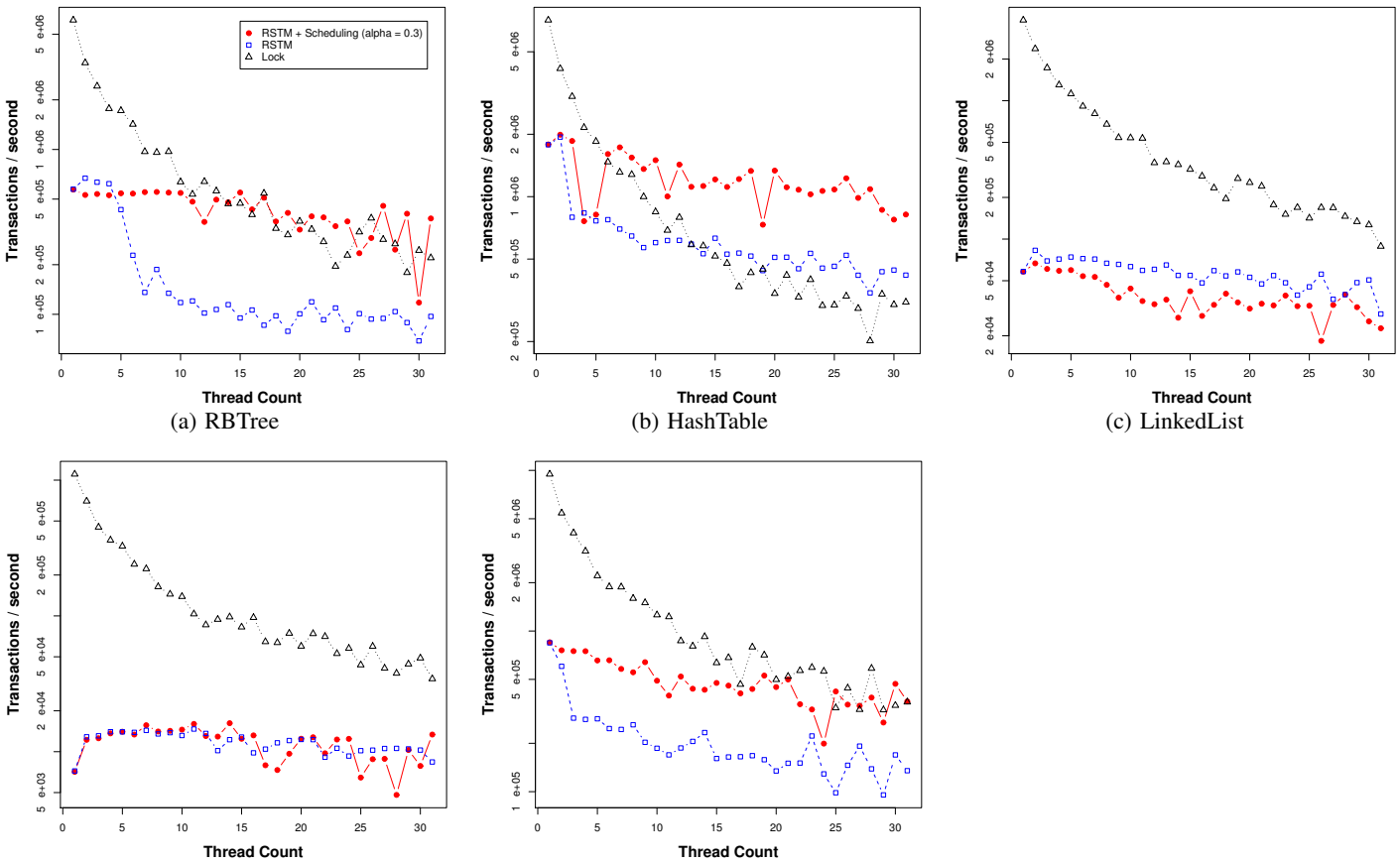


Figure 10: Scheduler Performance on an 8-way SMP Machine

As shown in the figure, when the number of concurrent threads is small, the relative throughput remains around 1. In these scenarios, the overheads of the queue synchronization actually bring slight performance degradation. As the contention increases with more concurrent threads ($\geq$ 17), the ATS-enabled RSTM starts to show performance improvement. The aggregate performance improvement ranges from 1.1x to 1.4x. Although there are some cases where minimum relative throughput goes below 1, that those verti-

[3]Due to the absence of hardware resources we could access, performance results on a 4-way SMP cannot be obtained. Moreover, 8-way SMP was the largest machine we had access to.



(a) RBTree (b) HashTable (c) LinkedList

(d) RandomGraph (e) LFUCache

Figure 8: Individual Benchmark Results on a 2-way SMP Machine

cal bars position higher than 1 indicates that the scheduling mechanism results in an overall net performance gain.

4.2.3 *Effect of Page Faults on Performance*

To better explain the significant performance improvement of ATS, in a separate experiment, we collected SAR counters while the workloads are running. SAR is a Linux performance monitoring tool that collects OS level statistics such as CPU utilization, number of context switches, interrupts, page faults, etc., for a specified time interval. We resorted to OS level counters since it has been reported that OS level counters play a key role in identifying the behavior of an application using a runtime library [5]. By manually performing correlation analysis over the collected counters, we found that the number of page faults (including major and minor page faults[4]) shows the best correlation to the performance improvement.

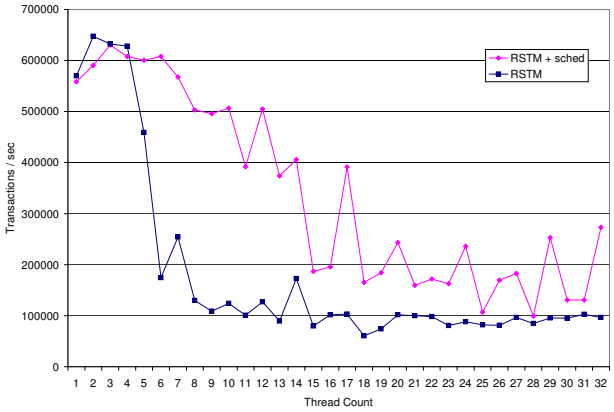
Figure 11(a) shows the throughput trend of RBTree on the 2-way SMP system as the number of threads increases. We chose it for our further analysis as ATS shows the most noticeable performance gain. Figure 11(b) shows the numbers of page faults as the sampling sequence increases. The figure has been flipped against x-axis to show the similarity of the trend to Figure 11(a). The SAR counter sampling frequency was not perfectly synchronized with the thread count increase. Nonetheless, these two graphs show that the performance improvement of ATS has close correlation to the reduction of page faults.

We attribute this to our scheduling scheme reducing the number of transactions that start execution. When there are more transactions, they tend to allocate more pages. Especially in Linux where page frame is managed on a per-CPU basis [4], when some of those transactions are aborted, pages that were brought in without contributing to the overall progress would adversely *pollute* the per-CPU page frame cache [4]. By reducing the number of transactions dispatched, ATS increases the hit rate of page frame cache, which in turn leads to better performance. This also explains the lower performance improvements of ATS on the 8-way SMP system when compared to the 2-way SMP system, since the increased number of processors — hence the number of per-CPU page frame caches — tend to increase page frame cache locality.

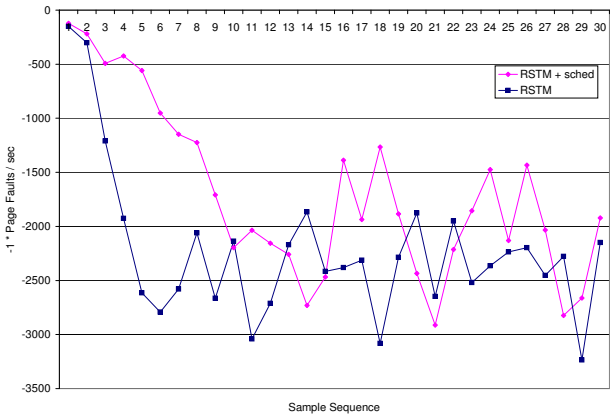
One could argue why someone would want to run a TM system in such an oversubscribed configuration. Unlike HTM systems, one of the key virtues of STM systems was to provide virtualized transactions that can survive context switches. Therefore, for an STM system, performance results for the oversubscribed configuration are equally important to those of the undersubscribed configuration. Many studies report the performance results of STM systems under an oversubscribed configuration [14, 23, 24, 25].

More specifically, there are two likely scenarios where a user might launch more threads than there are in the system. First, we can never assume that the end-user will dedicate the entire system to execute a single TM workload. For a dynamic scenario where multiple workloads are running in the system, a user cannot predetermine the correct number of threads to execute. Second, under the strict nested transaction model [20] a transaction can spawn multiple concurrent child transactions. Therefore, a TM implementation should not pose any upper bounds on the number of threads. These two scenarios will be prevalent once TM systems get deployed. Under such situation, ATS would act as a safety net to provide a reliable QoS on the overall system throughput.

[4]Major faults are those faults that actually end up loading a memory page from disk. Minor faults are those faults that only miss in the OS frame cache.



(a) Performance over Increasing Threads



(b) Page Fault Trend over Sample Sequence

Figure 11: Effect of Page Faults on Performance (2-way SMP)

4.2.4 *Tuning the α Value Dynamically*

Due to the space limitation, until now we have only shown the experimental results with the best performing α-values. However, throughout our experiments the value of α played a significant role in determining the overall ATS performance. In this section we discuss how to adapt the α-value automatically for performance given an application's dynamic behavior.

Recall from Section 2.1 that the contention intensity is comprised of two parts: past history and current contention. The past history and the current contention act as two different conflict predictors, while the α-value determining which predictor to weigh more. By applying competitive learning between these two predictors, the α-value can be adjusted automatically. In this scheme, penalizing one predictor will reward the other. The following pseudocode describes the algorithm:

```
if ( abort == true) {
    if ( current_contention == 0) {
        // penalize current contention
        alpha += Scheduler::DELTA;
    }
    if ( past_history <= Scheduler::THRESHOLD) {
        // penalize past history
        alpha -= Scheduler::DELTA;
    }
    // clip alpha value
    alpha = (alpha < 0) ? 0 : alpha;
    alpha = (alpha > 1) ? 1 : alpha;
}
current_contention = abort ? 1 : 0;
past_history = alpha * past_history +
                    (1 - alpha) * current_contention;
```

This algorithm adjusts the α-value only when an abort has materialized due to a misprediction. Upon each abort, α is adjusted by a step function to penalize the predictor that mispredicted. Penalizing the current contention rewards the past history, and vice versa, but the α-value does not change when both predictors mispredict.

Figure 12 shows the result of applying the above algorithm to the ATS-enabled RSTM. All the performance results were measured with the same 5 microbenchmarks on the 2-way SMP machine. We specifically chose this 2-way configuration since ATS showed the most sensitivity over the α-value.

Each line in the figure represents the harmonic mean of the relative throughput at a particular α configuration. Three of the lines represent the α-values previously used for the sensitivity analysis: 0.3, 0.5, and 0.7. The other implemented the above algorithm with α initially set to 0.5, and the constant adjustment set to 0.1.

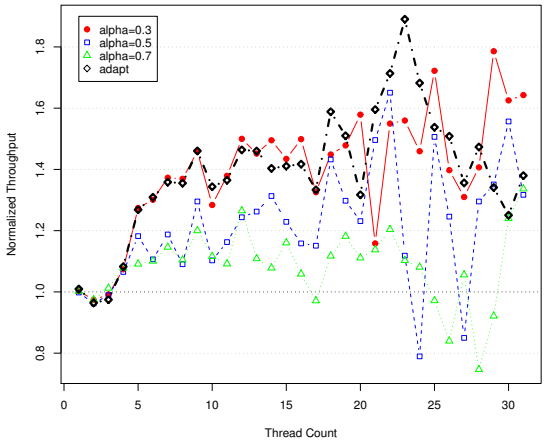


Figure 12: Automatic Tuning of the α Value (2-way SMP)

Although α was initially set to 0.5, we can see that the adaptive scheme more closely follows the performance of α = 0.3, which is the best setting among the three constant α-values. We also observed that no matter the initialization value, α converged to 0.4 for most of the workloads. With this training technique, the ATS scheduler will be able to adapt dynamically to maximize transaction throughput based on the online workload behavior observed.

5. RELATED WORK

TM [11, 12] 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 [21] and speculative synchronization from Martínez and Torrellas [16]. Nonetheless, speculative methods potentially suffer from backfire if speculation fails frequently. Our paper minimizes this negative effect on TM systems.

Other approaches to maximize the performance of TM systems include contention managers [10, 23]. Contention managers try to maximize the performance by effectively handling the contention after it has been detected. Hardware support to utilize this information has also been discussed [29]. Rather than to take action after the contention has been detected, our method fundamentally reduces the contention itself. Bai *et al.* [2] also propose a different method to reduce the contention itself. Nonetheless, the approach is limited in that it requires the Java executor framework, and it is only applicable to dictionary-based structures. ATS is more closely related to *admission control* found in an OS [27]. Under admission control, an OS can delay the admission of the work until the system utilization subsides below some threshold.

Previously proposed `retry` construct [8] is also similar to ATS in a sense that it delays the resume of a read transaction until a value in its read set changes. However, `retry` is more of a language construct for transactional synchronization, not meant to be

used as a performance optimization feature. Moreover, the construct does not specify in which order retried transactions should resume. We suspect that the resume order would have significant impact on workload performance, and in that case our scheduler could be utilized to impose ordering on the retrying transactions.

6. CONCLUSION

In this paper, we propose the concept of adaptive transaction scheduling, called ATS, that addresses performance issues caused by excessive transactions in both hardware transactional memory and software transactional memory systems. With the runtime parallelism feedback obtained from the contention intensity detection mechanism, we can significantly increase transaction effectiveness for workloads that lack parallelism due to high contention. Differing from a contention manager that manages contention after it has been detected, our ATS proactively reduces contention itself upon transaction scheduling. In our experiments, we show that our ATS scheme is a complementary technique that delivers additional performance on top of a contention manager.

Based on this notion, we demonstrated a very low-cost adaptive transaction scheduler. In this scheme, the number of concurrent transactions are adaptively adjusted by dynamically controlling the execution point of a transaction to maximally exploit the parallelism inherent within a given program phase. Through our case study, we have shown that our scheduler not only guarantees that an ATS-enabled HTM system can perform better than approaches using single global lock, but also significantly improves performance for both generic HTM and STM systems. In our experiments, the maximum performance speedup on an HTM system reaches 1.97x. The relative performance speedup's on STM are from 1.3x to 1.5x while the speedup of the peak performance is 5.9x.

7. ACKNOWLEDGMENTS

We would like to thank Kevin Moore, Josh Fryman, and several anonymous reviewers for their constructive input to this research. This work was supported in part by an NSF CAREER Award (CNS-0644096).

8. REFERENCES

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA-11*, February 2005.

[2] T. Bai, X. Shen, C. Zhang, W. N. Scherer, III, C. Ding, and M. L. Scott. *A Key-based Adaptive Transactional Memory Executor*. Technical Report URCS TR 909, University of Rochester, December 2006.

[3] C. Blundell, J. Devietti, E. C. Lewis, and M. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA-34*, 2007.

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

[5] K. Chow, R. Morin, and K. Shiv. Enterprise Java performance: Best practices. In *Intel Technology Journal, volume 7, issue 1*, pages 32–46, February 2003.

[6] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *ASPLOS-12*, pages 347–358, 2006.

[7] 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 *ISCA-31*, pages 102 – 113, June 2004.

[8] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05*.

[9] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS-23*, page 522, 2003.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC 2003*, pages 92 – 101.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA-20*, pages 289 – 300. May 1993.

[12] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[13] V. Marathe, W. Scherer III, and M. Scott. Adaptive software transactional memory. In *DISC-2005*.

[14] 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 *TRANSACT*, June 2006.

[15] 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*, September 2005.

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

[17] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA-33*, pages 53–65, 2006.

[18] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA-34*, pages 69–80, 2007.

[19] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA-12*, pages 254 – 265, February 2006.

[20] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *SCOOL'05*.

[21] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO-34*, pages 294–305, 2001.

[22] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA-32*, pages 494 – 505, June 2005.

[23] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC 2005*, pages 240 – 248.

[24] W. N. Scherer, III and M. L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the 2004 ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[25] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC-2007*, pages 107–113.

[26] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA-34*, pages 104–115, 2007.

[27] J. A. Stankovic. Admission control, reservation, and reflection in operating systems. *IEEE Bulletin of Technical Committee on Operating Systems and Application Environments (TCOS)*, 10(2), 1998.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA-22*, pages 24–36, 1995.

[29] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *TRANSACT*, 2006.