# ADAPTIVE TRANSACTION SCHEDULING FOR TRANSACTIONAL MEMORY SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Richard M. Yoo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2008

# ADAPTIVE TRANSACTION SCHEDULING FOR TRANSACTIONAL MEMORY SYSTEMS

Approved by:

Professor Hsien-Hsin S. Lee, Advisor
School of Electrical and Computer Engineering
*Georgia Institute of Technology*

Professor Douglas M. Blough
School of Electrical and Computer Engineering
*Georgia Institute of Technology*

Professor Sudhakar Yalamanchili
School of Electrical and Computer Engineering
*Georgia Institute of Technology*

Date Approved: March 31, 2008

*To Truth.*

# ACKNOWLEDGEMENTS

This work would have been impossible without the professional and emotional support from a lot of people. First of all, I would like to give my heartfelt thanks to my family — mom, dad, and my little brother — for their understandings and encouragements. Through the darkest times, they have been my sole *raison d'être*.

I would then like to extend my great appreciation to my research advisor, Professor Hsien-Hsin Lee, for his dedicated and incessant support in research. To him I owe my every single professional success in the U.S. He has been a great mentor and an intellectual support throughout the journey which turned out not to be always sunny.

I cannot thank more of my colleagues at MARS (Microprocessor Architecture Research Society) lab: (in alphabetical order) Joonho Baek, Chinnakrishnan Ballapuram, Nishank Chandawala, Eric Fontaine, Joshua Fryman, Mrinmoy Ghosh, Nob Kladjarern, Dean Lewis, Pratik Marolia, Fayez Mohamood, Nak Hee Seong, Ahmad Sharif, Weidong Shi, Taeweon Suh, Vikas Vasisht, and Dong Hyuk Woo. Not only were they professionally inspiring and supportive, they were great people to spend life together.

Last but not least, I would like to thank all other friends and people who have been invaluable during my study at Tech.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# SUMMARY

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 will actually become dominant in future workloads when large-scale transactions are frequently executed.

In this thesis, 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.

# CHAPTER I

# INTRODUCTION

Due to its radically simple programming semantics, a transactional memory (TM) system [19, 21] represents a promising technology to enable highly efficient multi-threaded programming on the emerging multi-core platforms. Especially for hardware transactional memory (HTM), programmers do not need to concern themselves about the exact data conflict information. Rather, a programmer can speculatively mark a potentially conflicting code region as a transaction, since the sequential correctness among transactions will be guaranteed by the underlying TM implementation. In addition, by speculatively executing more than one transaction in an atomic block[1], a TM system can theoretically achieve higher performance by exploiting parallelism prohibited by a conventional lock-based system. Software transactional memory (STM), on the other hand, requires exact conflict information since transactional barriers (*i.e.*, transactional bookkeeping routines) are only executed by explicit calls from the user code. However, with the recent advent of compiler-driven barrier insertion [44, 2, 28], the STM programming model has been dramatically simplified.

In effect, TM separates the *implementation* from the *semantics* — programmers simply denote *what* transactions are, and the responsibility with regard to *how* to execute them falls onto the underlying TM system whose implementation details are hidden from the high-level application programmers. Ideally, TM systems can improve performance by leveraging the headroom between the implementation and the semantics. Unfortunately, due to its high implementation complexity, most HTM

[1] Although it is a common practice to use the term *critical section* and *atomic block* interchangeably, in this thesis we limit the use of critical section to describing traditional exclusive locking, and atomic block to describing TM approach.

research thrusts mainly focus on implementation issues with less attention to performance aspects [14, 1, 35, 30, 9, 26, 4].

Reducing the number of aborting transactions is one effective way to improve the efficiency of a TM system. The impact of aborting a transaction is multidimensional. First, all the completed work done by an aborted transaction becomes useless and will have to be retried. If an aborted transaction had aborted other transactions earlier, the total amount of work lost will be worse. Secondly, the cost to rollback a transaction can be tremendous, requiring large memory capacity and bandwidth. Thirdly, for TM implementations that keep speculative states in caches, invalidating those cache lines upon transaction abort will further increase the number of cache misses (Section 4.1.3.2). Likewise, because transactions are implemented on top of threads, aborting transactions too frequently could adversely affect the virtual memory page management (Section 4.2.2.3). In the worst case, those TM systems that detect conflicts upon each object's acquisition (eager conflict detection) can result in a livelock. Such ineffective transactions degrade the overall performance of a TM system.

STM researchers were aware of these performance issues. To alleviate these problems, researchers proposed the concept of the *contention manager* [18, 22, 40], a user-level code module that enforces priority among transactional accesses. The priority could be determined by several indices, such as the age of a transaction based on timestamps, the amount of work done based on memory footprint [18, 40], etc. 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 that consulted the contention manager will use a delay back-off, thus enabling the offending transaction to finish before the requestor

retries the NACKed permission. By varying the number of back-off attempts and their intervals, contention managers can significantly reduce the number of transaction aborts [40]. STM implementations employing contention managers have shown reasonable performance improvement over STMs without contention managers, and the capability of avoiding livelocks [18, 23, 22].

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 already started executing an atomic block, 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[2], 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 [42]. For this reason, contention managers tend to be distributed; they detect and resolve conflicts in 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

[2]As a good example, consult the contention manager interface of RSTM [22].

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 [18, 23, 22].

To tackle these problems, we propose in this thesis a new technique called *adaptive transaction scheduling* (ATS). In ATS, a *scheduler* adaptively controls the number of concurrent transactions executing in atomic blocks 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 light-weight 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 implemented on either the HTM or the 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. On 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 thesis is the first to incorporate transaction scheduling on TM systems to adaptively exploit the maximum parallelism. The rest of the thesis is organized as follows. Chapter 2 gives a brief overview of TM and contention manager. Chapter 3 then describes our scheduling framework. Experimental results can be found in Chapter 4. Specifically, Section 4.1 and Section 4.2 discuss our implementation on LogTM and RSTM and analyze their performance. In Chapter 5 we also propose a methodology to integrate ATS with operating system schedulers. Related studies in TM area are discussed in Chapter 6. Finally, Chapter 7 concludes.

# CHAPTER II

# TRANSACTIONAL MEMORY OVERVIEW

This chapter provides a high-level overview of TM. Rather than focusing on the implementation details, we focus on the programming semantics of TM; what a programmer should expect from a TM implementation. We also describe contention managers — the current state-of-the-art approach in optimizing TM system performance.

## *2.1 Transactional Memory*

Originally proposed in [19] and revitalized in [14], TM represents a new type of concurrency control mechanism for multithreaded applications. Especially, TM borrows the concept of transaction from the database community. Among the four ACID attributes of database transactions [36], transactions in TM generally implement 3 attributes: Atomicity, Consistency, and Isolation (ACI). In TM terminology, a *transaction* is a sequence of instructions that are endowed with ACI attributes.

- **Atomicity** requires the constituent instructions to take effect as a whole; either all or none of them take effect on the system state. When a transaction successfully finishes, it *commits*; when it fails it *aborts*. A TM system can re-execute an aborted transaction until it commits.

- **Consistency** enforces the system state to be consistent when seen from subsequent transactions. The end result of a transaction should transit the system to another consistent state.

- **Isolation** requires that each transaction produce a correct result, regardless of the other transactions executing concurrently.

In programming language, a transaction is typically denoted as an *atomic block*. Instructions within the scope of an atomic block would be executed with ACI attributes. The following pseudo code depicts an application of atomic block.

```
atomic {
    y = x++;
    foo(y);
}
```

Notice that the resulting atomic block is dynamically scoped; it encompasses all the instructions executed while the control is in the atomic block. So instructions that are not lexically included in the atomic block could still be executed transactionally if they happen to lie on the program execution path. For example, instructions from the function foo() would be executed transactionally, although they are not lexically included in the above atomic block.

A *TM system* is a set of components in hardware or software that are necessary to endow ACI attributes to transactionally executing instructions. Endowing ACI attributes typically requires a TM system to implement the following functionality. First, atomicity requires that the intermediate computation results be buffered. When a transaction commits, these buffered results are then applied to the system state. When a transaction aborts, the buffered results are discarded. Secondly, consistency mandates a version control system in TM. Depending on the execution outcome of previous transactions, the version control system decides whether the subsequent transactions be supplied with previous consistent system state or updated consistent system state. Lastly, isolation requires the buffered results of a transaction not be visible to other transactions.

In addition to these supports, a TM system must implement a conflict detection scheme [21]. Conflict detection schemes can be categorized by the granularity at

which they detect transactional conflicts. HTMs usually detect conflicts on cache line granularity [19, 14, 1, 30]. In these approaches, typically a transactional read / write bit is maintained for each cache line. Whenever a processor transactionally accesses a cache line, the pertaining bit is set. Once this bit is set, the system monitors incoming cache coherence traffic to detect conflicts. STMs can detect conflicts at object granularity [23, 22] or cache line (word) granularity [37, 11]. In fact, lock based STM systems [37, 11] can detect conflicts at a varying degree of granularity by changing the hash function utilized to map each transactional location to a lock [21].

When compared to traditional thread synchronization schemes such as locks, transactions improve abstraction and composition in programmability in that 1) it does not identify each atomic block, and that 2) it does not specify which memory locations are to be synchronized. This feature has been shown to radically simplify the multithreaded programming semantics, and to foster composability among multithreaded codes [21]. Performance-wise, TM approach can also improve performance by speculatively executing multiple transactions inside an atomic block, when compared to the exclusive locks where only one thread is allowed to make progress inside a critical section. TM has also been reported to eliminate some of the pitfalls in lock-based synchronization, such as priority inversion, convoying, and deadlock [19].

Although a popular concept in the database area, *nested transactions* [33] are currently supported to a limited degree on a TM system [26, 31]. More specifically, most TM systems limit the concurrency of child transactions so that each transaction can have at most one child transaction at a time. Such implementation is termed as *linear nesting.* There exists a design paper [33] which discusses design requirements to support multiple concurrent child transactions, but due to its complexity no TM implementation has been successful to implement it. In the rest of this thesis, our techniques for TM performance improvement would be focused on *outermost* transactions.

## *2.2 Contention Manager*

Traditional synchronization schemes such as locks and semaphores implement *blocking synchronization*; that is, the system may not make forward progress if threads are preempted while holding exclusive access to a shared resource. On the contrary, TM systems implement *nonblocking synchronization*; a stalled thread cannot cause all other threads to stall indefinitely. Depending on the degree of forward progress guarantee, nonblocking synchronization can be categorized as follows:

- **Wait Freedom**. A system implementing wait freedom guarantees that all the threads contending for shared objects make forward progress.
- **Lock Freedom**. A system implementing lock freedom guarantees that at least one thread among all the threads contending for shared objects makes forward progress.
- **Obstruction Freedom**. A system implementing obstruction freedom guarantees that a thread will make forward progress in the absence of contention over shared objects.

In the degree of forward progress guarantee, wait freedom is the strongest, lock freedom the next, and obstruction freedom the weakest. The unique feature of obstruction freedom is that it need not concern the system-wide progress. All it has to guarantee is that one thread will make progress if it is left to execute without any contention from other threads.

The performance of a TM system heavily depends on the number of transaction aborts [18, 22, 40]. At the least, aborting a transaction amounts to the loss of computation resources that were used up during the speculation. Even worse, those wasted resources sometimes represent larger sunken costs — the system might have performed better if those wasted resources were used to perform other computations,

*e.g.*, execute a different transaction, instead. In the worst case, for those TM systems that detect conflicts upon each object's acquisition (eager conflict detection) can result in a livelock.

To reduce the number of transaction aborts, a TM system should enforce a priority mechanism among the accesses from multiple transactions. However, exercising such priority on a wait free or a lock free TM is not a trivial task [18]. First, determining the priority while assuring that the system makes progress as a whole itself is not an easy task. Moreover, maintaining the system-wide progress information inside the priority-enforcing component usually amounts to the component becoming a central, serialized bottleneck.

Obstruction freedom, on the contrary, gives a great leeway to implementing the priority mechanism in a TM. All that the priority mechanism has to guarantee is that each transaction would be guaranteed to execute in isolation at some point in time. Then the underlying obstruction-free TM will make sure that such transaction would execute till completion. So as long as this guarantee is met, the priority mechanism can decide to abort any transaction at any time without worrying about the system-wide progress. Moreover, since the obstruction freedom does not impose any restriction on the number of threads that should be making progress at a given point in time, priority mechanisms in obstruction-free synchronization need not be aware of system-wide contention information; they can resolve the conflict on a peer-to-peer basis, and this radically simplifies the design of conflict resolution policies.

*Contention manager* refers to the class of such priority enforcing mechanisms that are implemented in an obstruction-free TM. When a transaction detects a conflict, it consults its contention manager to decide whether to abort the offending transaction. With the priority and other transactional information, the contention manager heuristically evaluates and decides whether aborting the offending transaction will improve the overall throughput. If it decides not to abort the offending transaction,

the requestor backs off hoping that the offending transaction would finish before it retries the denied object access.

To support flexible contention resolution policies, contention managers are usually implemented in software. In particular, it has been shown that STM implementations employing contention managers can obtain reasonable performance improvement over STMs without contention managers, and the capability of avoiding livelocks [18, 23, 22].

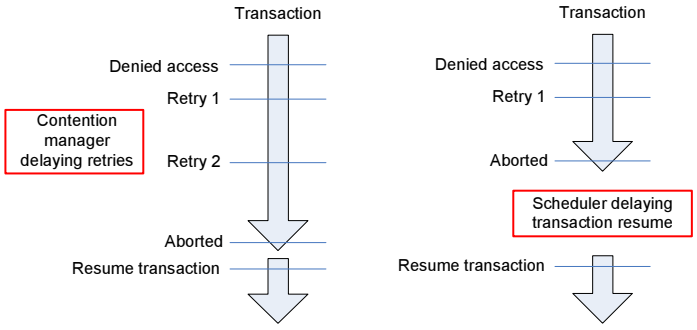
# CHAPTER III

# TRANSACTION SCHEDULING

In our execution model, a thread enters and leaves multiple *atomic blocks* throughout its lifetime. Upon *entering* an atomic block, the thread starts a *transaction*. The thread might resume the transaction multiple times if the previous transaction aborts. A thread *leaves* the atomic block 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). To say that a contention manager employs an exponential backoff scheme means that the retry interval expands 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 the 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 aborted transaction resumes with




(a) Contention Manager (b) ATS

**Figure 1:** Contention Manager versus ATS

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.

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

## *3.1 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, contention intensity can be dynamically controlled. 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 a 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 its 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, $\alpha$, determines which portion of the equation weighs more — either the past history or the current

contention information. When the $\alpha$ 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 $\alpha$ value is small, the current contention information is reflected more quickly. In fact, based on the past commit / abort history, contention intensity *predicts* the probability that a transaction would face another conflict.

By default, the contention intensity value should be reset to 0 when a thread changes the entered atomic block (which starts at a different PC). To put it differently, when a thread loops around the same single atomic block, 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 atomic block — *i.e.*, do not re-enter that particular atomic block before entering another atomic block — each thread's contention intensity is already close to 0 due to the phased behavior of multi-threaded applications.

## *3.2 A Low Cost Transaction Scheduler*

In ATS, 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 3.1. 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 surpasses the threshold, the thread will stall and report to the scheduler, waiting a dispatch. On the contrary, when the contention intensity is below the threshold, the thread begins a transaction normally. So when the contention intensity is low, ATS has little effect except for the penalty of intensity check. A thread that resorts to the scheduler will be freed from the scheduler and begin a transaction normally when the contention intensity subsides below the threshold.

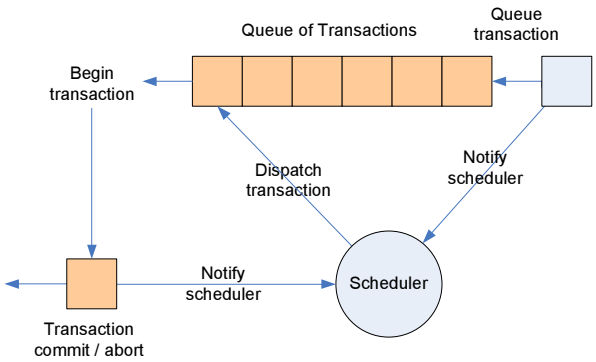



**Figure 2:** A Queue-Based Transaction Scheduler

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 the 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 transactions that have been dispatched from the queue are still executing, it is dispatched immediately. Otherwise, the transaction waits in the queue until exclusivity is met. Moreover, the transaction that has been 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 to be 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.

## *3.3 Transaction Scheduler in Action*

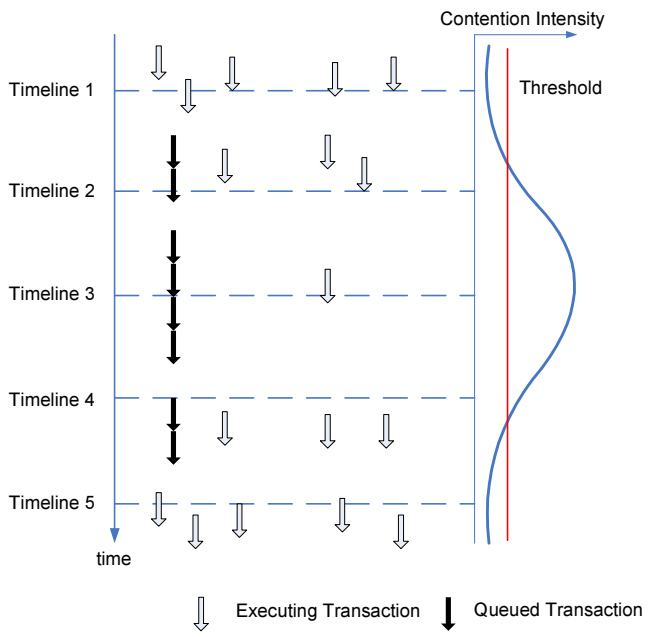



**Figure 3:** Behavior of a Queue-Based Scheduler

In Figure 3 we use an example to detail the behavior of a transaction scheduler. For now, let us assume that there is only one atomic block 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 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. When the contention intensity of a transaction drops below the threshold, it will be dequeued from the scheduler to exploit 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.

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

## *3.4 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.

Second, 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; for example, controlling the number of concurrent transactions is only possible when we have a global view of the contention. Nonetheless, due to its adaptive nature, a centralized ATS does not undermine the scalability of an HTM system.

Last, infrequent access and a simple design of ATS enable its low-cost integration with HTM, as we describe in Section 4.1. More importantly, contention managers can only be implemented on obstruction-free TM systems [17, 18]; largely due to this assumption, contention managers can resolve conflicts on a peer-to-peer basis without the system-wide contention information. On the contrary, 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.

# CHAPTER IV

# EXPERIMENTAL RESULTS

To evaluate the advantages of ATS, we implement our proposed queue-based scheduling scheme in both HTM and STM systems. Section 4.1 details our ATS implementation on LogTM [30] (an HTM system), and Section 4.2 discusses our implementation on RSTM [22] (an STM system).

## *4.1 Hardware Transactional Memory Integration*

### 4.1.1 LogTM Settings

For an HTM system, we implemented ATS on LogTM [30]. LogTM is an eager version management, eager conflict detection TM system that detects conflicts on cache line granularity. LogTM has been released as a memory timing module of the GEMS simulator [24][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. 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

[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.

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 waited for hundreds to thousands of cycles. Table 1 specifies the simulated machine in GEMS.

**Table 1:** Simulation Settings for LogTM

| **Simulated System Settings** | |
|---|---|
| CPU | Sixteen 1GHz SPARCv9<br>single-issue, in-order<br>non-memory IPC=1 |
| L1 Cache | 4-way split, 64 KB<br>5-cycle latency |
| L2 Cache | 4-way unified, 16 MB<br>10-cycle latency |
| Memory | 4 GB |
| Directory | centralized, 6-cycle latency |
| Interconnection Network | hierarchical switch topology<br>40-cycle link latency |
| **LogTM Settings** | |
| m_abortStartupDelay<br>m_abortPerBlockDelay | 40 cycles<br>20 cycles |

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* [30]; 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.

### 4.1.2 Benchmark Suite

Our benchmark suite includes selected SPLASH-2 applications [45] and a modified Deque microbenchmark included in the LogTM release. These workloads are transactionized by replacing the lock with transactions. Table 2 lists the benchmark suite. We use the same subset of SPLASH-2 applications used in the original LogTM paper [30]. Those omitted ones are not considered as representative TM applications since their atomic blocks mostly perform trivial memory operations, *e.g.*, single induction variable increments.

**Table 2:** LogTM Benchmark Suite

| **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 |

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.

**Table 3:** Execution Time Statistics on LogTM

| Benchmark | Execution Time | | Xact Begin | | Commit | | Abort (%) | |
|---|---|---|---|---|---|---|---|---|
| | base | ATS | base | ATS | base | ATS | base | ATS |
| Water-nsquared | 52383081 | 52383081 | 17664 | 17664 | 17664 | 17664 | 0 (0%) | 0 (0%) |
| Water-spatial | 65462563 | 65462563 | 285 | 285 | 285 | 285 | 0 (0%) | 0 (0%) |
| Ocean | 291813916 | 291813916 | 1666 | 1666 | 1664 | 1664 | 2 (0.1%) | 2 (0.1%) |
| Raytrace | 50333882 | 50852127 | 48654 | 48128 | 47782 | 47781 | 872 (1.8%) | 347 (0.7%) |
| Cholesky | 22596229 | 22258328 | 6754 | 6550 | 5938 | 5935 | 816 (12.1%) | 615 (9.4%) |
| Barnes | 25015887 | 23878230 | 3055 | 2575 | 2319 | 2319 | 736 (24.0%) | 256 (10.0%) |
| Deque-14436 | 24037196 | 20970633 | 3713 | 1783 | 1200 | 1200 | 2513 (67.7%) | 583 (32.7%) |
| Deque-2048 | 20081574 | 13783990 | 3492 | 1866 | 1200 | 1200 | 2292 (65.2%) | 666 (35.7%) |
| Radiosity | 472253209 | 239312955 | 490658 | 336154 | 276917 | 278711 | 213741 (43.6%) | 57443 (17.1%) |

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 [29].

In all of these benchmarks, one out of 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, to maximize concurrency, thread affinity was fixed so that each thread executes on a single CPU. Throughout the experiments, $\alpha$ was fixed to 0.7, while the threshold was fixed to 0.5.

### 4.1.3 LogTM Result Analysis

#### *4.1.3.1 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.

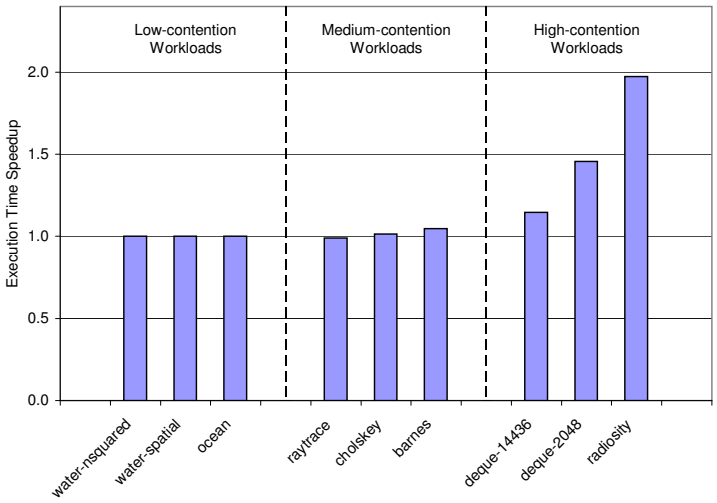



**Figure 4:** Execution Time Speedup

Not surprisingly, low-contention workloads exhibit zero or negligible abort rates. As explained in Section 4.1.2, atomic blocks 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

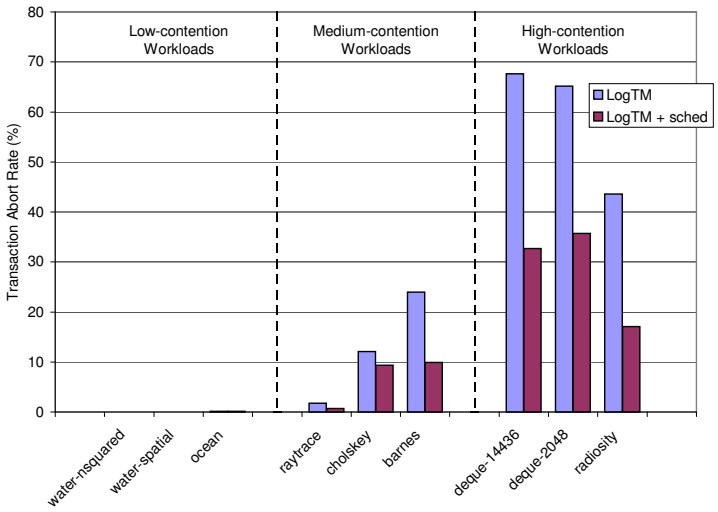



**Figure 5:** Transaction Abort Rate

5% speedup respectively. For Raytrace, the chosen $\alpha$ 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 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.

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.

#### *4.1.3.2 Improving the Quality of Transactions*

Not only does ATS reduce execution time and transaction abort rate, it also improves the *quality* of each transaction. Table 4 denotes the characteristics related to the quality of transactions. Same as in Table 3, we show the numbers for the baseline LogTM (base) and the LogTM enhanced with ATS for each category.

**Table 4:** Transaction Quality Statistics on LogTM

| **Benchmark** | **Xact Latency (stdv)** | | **L1D MPKI** | |
|---|---|---|---|---|
| | base | ATS | base | ATS |
| Water-nsquared | 1548 (315) | 1548 (315) | 2.70 | 2.70 |
| Water-spatial | 501 (4764) | 501 (4764) | 1.25 | 1.25 |
| Ocean | 849 (279) | 849 (279) | 2.33 | 2.33 |
| Raytrace | 6857 (37280) | 6332 (11499) | 6.73 | 6.67 |
| Cholesky | 1553 (4824) | 1174 (2720) | 1.10 | 1.11 |
| Barnes | 9326 (74878) | 2245 (4736) | 0.96 | 0.95 |
| Deque-14436 | 129541 (143462) | 39045 (35107) | 5.06 | 2.32 |
| Deque-2048 | 72857 (102182) | 10641 (8948) | 2.50 | 1.54 |
| Radiosity | 16488 (60769) | 1738 (4975) | 12.96 | 3.99 |

The first of such characteristics 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 requesting transaction until the memory request can be satisfied [30]. Hence higher contention typically leads to 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, 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

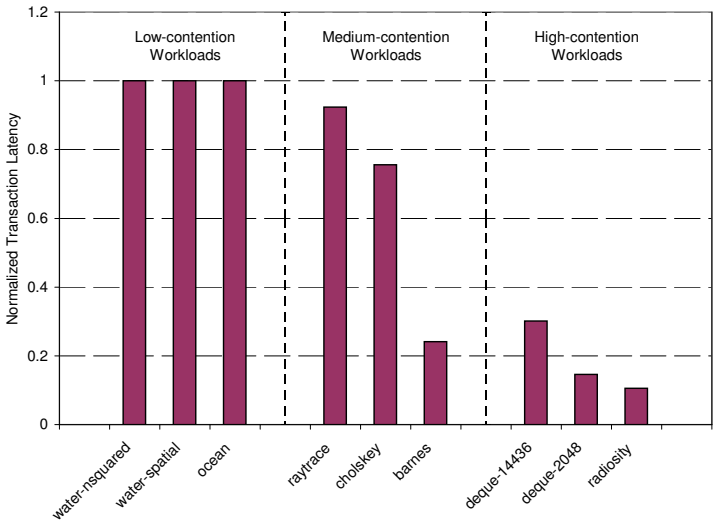



**Figure 6:** Normalized Transaction Latency

stdv values in Table 4 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 4.1.3.1, 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

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

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 [30]) 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 4 shows the L1D cache Misses Per Kilo Instructions. As expected, we can see that high-contention workloads benefit from our technique.

#### *4.1.3.3 Guaranteeing Performance Lower Bound*

One way to obtain a TM workload is to convert critical sections guarded by coarse-grained locks into atomic blocks (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 amounts to degenerating 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 conflict 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 [14]. TMs that detect conflict 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.

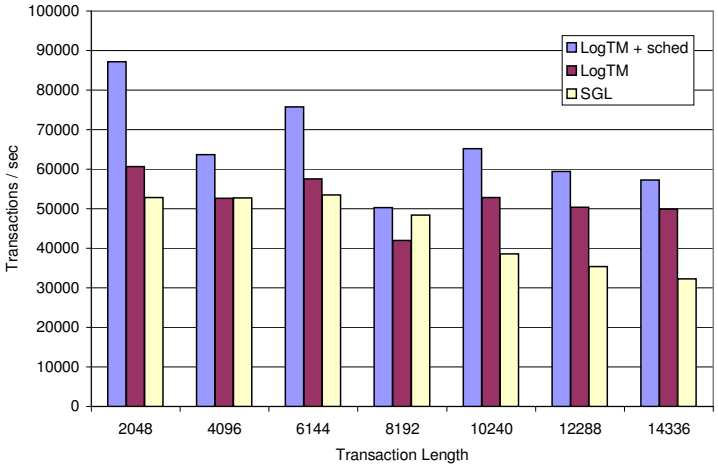



**Figure 7:** Throughput on Deque Microbenchmark

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 [4, 5]. 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.

## 4.2 Software Transactional Memory Integration

### 4.2.1 RSTM Settings

As for the STM, we implemented our ATS on top of the RSTM framework from the University of Rochester [22]. 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 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 [40, 22, 42].

The contention manager *Polka* [40] 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 [22].

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 5 describes the specifications of those two machines and their operating systems.

**Table 5:** RSTM Hardware Settings

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

[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.

### 4.2.2 RSTM Result Analysis

#### *4.2.2.1 Results on a 2-way SMP Machine*

To study the sensitivity of our contention intensity equation to the value of $\alpha$, we executed the benchmarks with 3 different $\alpha$ values: $\alpha = 0.3$, 0.5, and 0.7. Throughout these experiments the threshold was set to 0.5. Figure 8 summarizes the results.

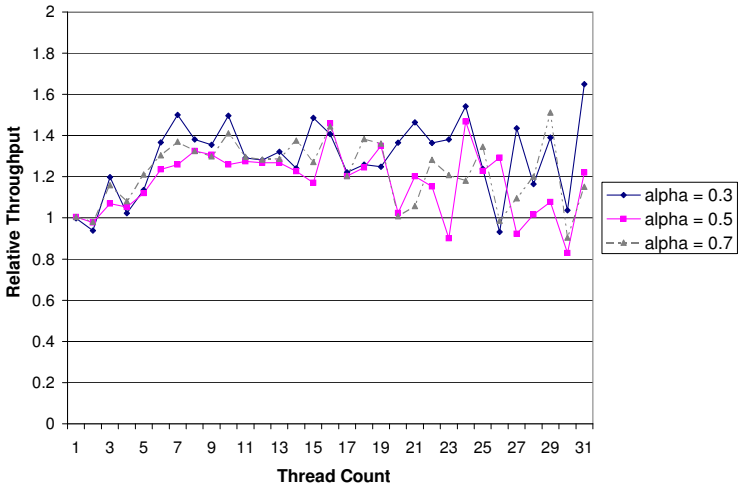
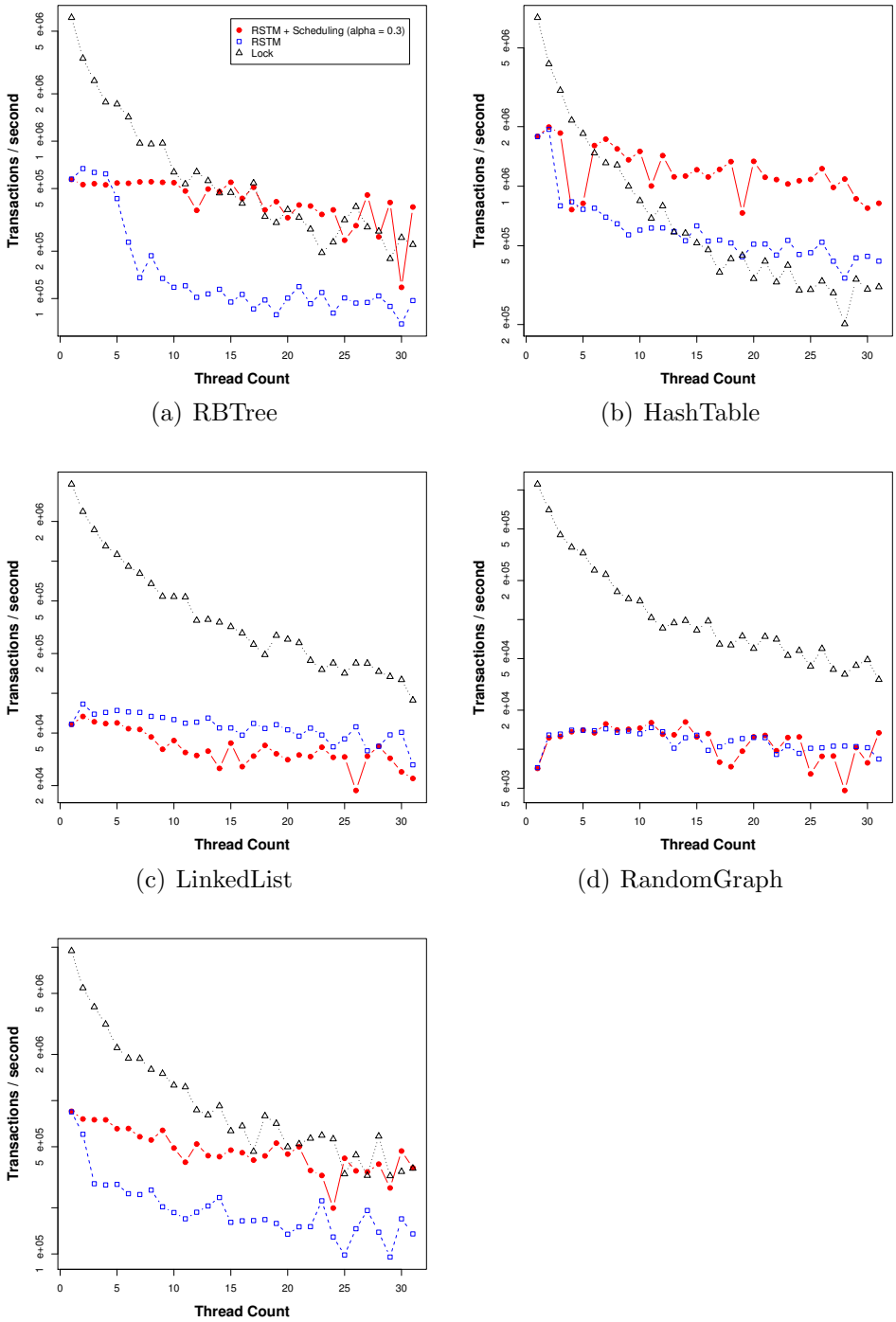



**Figure 8:** Sensitivity Study on a 2-way SMP Machine

In this figure, the y axis denotes the relative throughput of ATS-enabled RSTM over the baseline RSTM; the x axis denotes the number of concurrent threads. Each data point denotes the harmonic mean of relative throughput calculated over the 5 benchmarks. Although the trend for the three $\alpha$ values gets entangled when significant number of threads execute (thread count $\geq$ 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 $\alpha$ value, the performance of ATS-enabled RSTM is then compared with the baseline RSTM and locks in Figure 9. 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




(a) RBTree

(b) HashTable

(c) LinkedList

(d) RandomGraph

(e) LFUCache

**Figure 9:** Individual Benchmark Results on a 2-way SMP Machine

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 10 summarizes all the RSTM experimental results on 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.

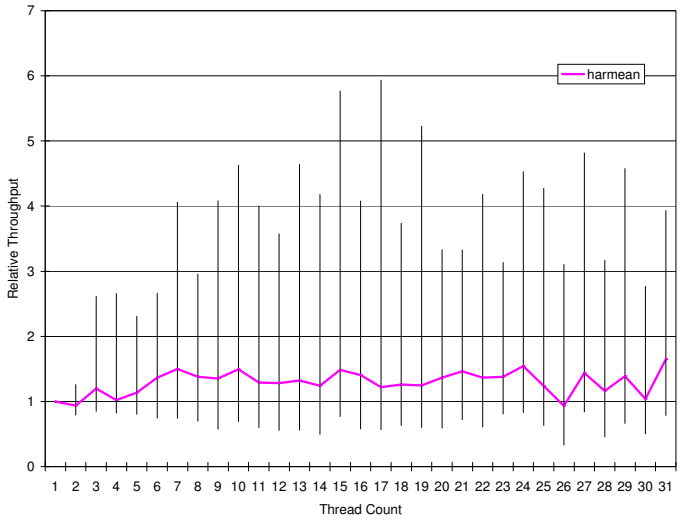



**Figure 10:** Scheduler Performance on a 2-way SMP Machine

#### *4.2.2.2 Results on an 8-way SMP Machine*

The same sensitivity study as in Section 4.2.2.1 was performed on an 8-way SMP machine. Figure 11 shows the results.

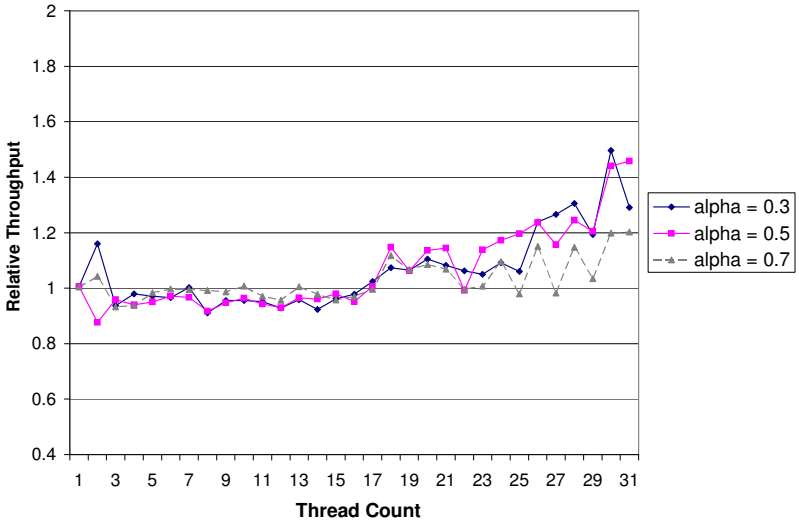
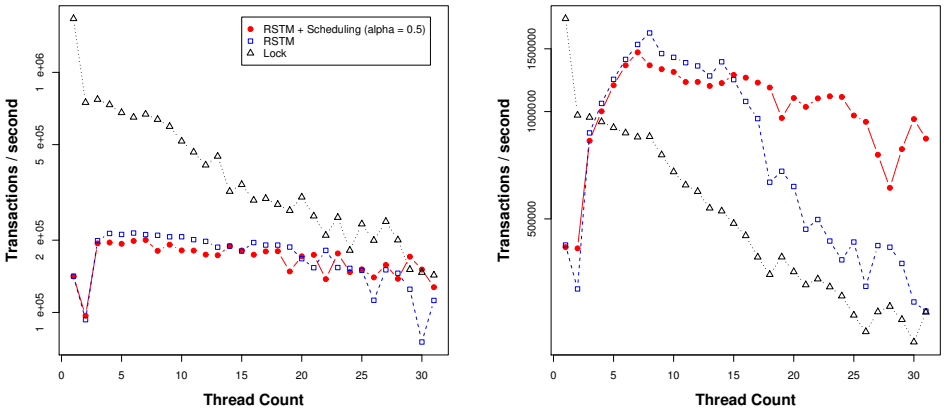



**Figure 11:** Sensitivity Study on an 8-way SMP machine

In this experiment, the $\alpha = 0.5$ case outperforms the others when sufficient number of threads (thread count $\geq$ 15) are executing. With $\alpha = 0.5$, a new thread will encounter two consecutive aborts before resorting to the scheduler.

The performance results on our 8-way SMP machine is shown in Figure 12. 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 $\leq$ 15), the ATS and the baseline show similar throughputs. However, as the thread count increases, the ATS-enabled RSTM starts to perform better than the baseline. This trend is summarized in Figure 13.

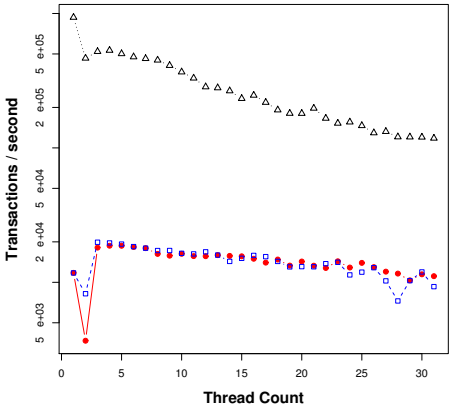
As shown in the figure, when the number of concurrent threads is small, the relative throughput remains around 1. In these scenarios, the overheads of the queue synchronization actually bring slight performance degradation. As the contention increases with more concurrent threads ($\geq$ 17), the ATS-enabled 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



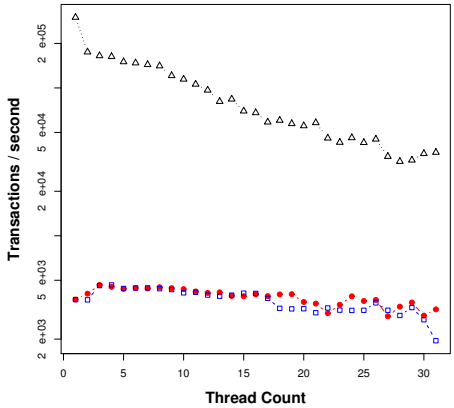

(a) RBTree



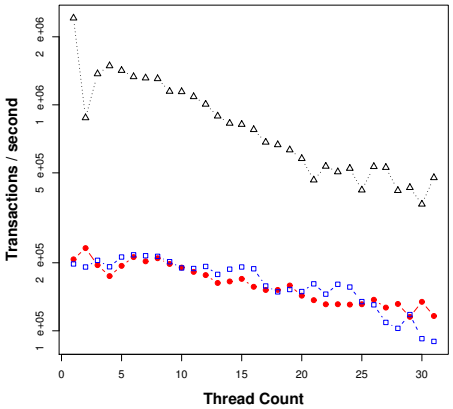

(b) HashTable




(c) LinkedList




(d) RandomGraph




(e) LFUCache

**Figure 12:** Individual Benchmark Results on an 8-way SMP Machine

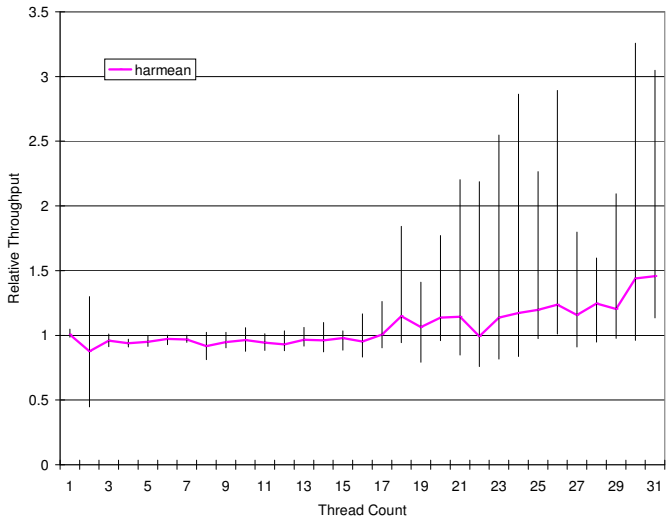



**Figure 13:** Scheduler Performance on an 8-way SMP Machine

below 1, that those vertical bars position higher than 1 indicates that the scheduling mechanism results in an overall net performance gain.

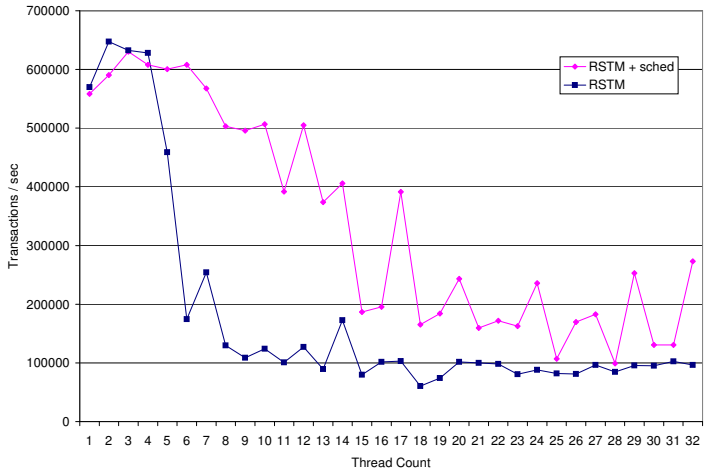
#### *4.2.2.3 Effect of Page Faults on Performance*

To better explain the significant performance improvement of ATS, 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 [8].

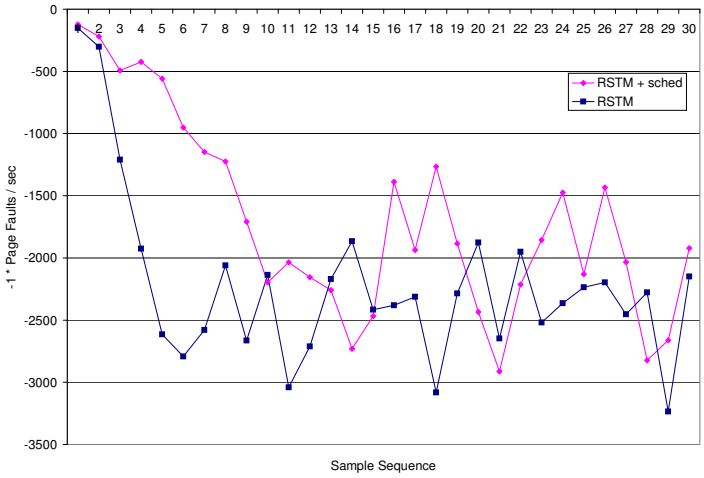
By manually performing correlation analysis over the collected counters, we found that the number of page faults (including major and minor page faults[4]) shows the best correlation to the performance improvement.

Figure 14(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 14(b) shows the numbers of page faults

[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 14:** Effect of Page Faults on Performance (2-way SMP)

as the sampling sequence increases. The figure has been flipped against x-axis to show the similarity of the trend to Figure 14(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 [6], 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 [6]. By reducing the number of transactions dispatched, ATS increases the hit rate of page frame cache, which leads to better performance.

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 [22, 40, 39, 41].

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 pre-determine the correct number of threads to execute. Second, under the strict nested transaction model [33] 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.

#### *4.2.2.4 Tuning the $\alpha$ Value Dynamically*

Throughout our experimental results we have shown that the value of $\alpha$ plays a significant role in determining the overall ATS performance. In this section we discuss how to adapt the $\alpha$-value automatically for performance given an application's dynamic

behavior.

Recall from Section 3.1 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 $\alpha$-value determining which predictor to weigh more. By applying competitive learning between these two predictors, the $\alpha$-value can be adjusted automatically. In this scheme, penalizing one predictor will reward the other. The following pseudo-code 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 $\alpha$-value only when an abort has materialized due to a misprediction. Upon each abort, $\alpha$ 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 $\alpha$-value does not change when both predictors mispredict.

Figure 15 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 $\alpha$-value.

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

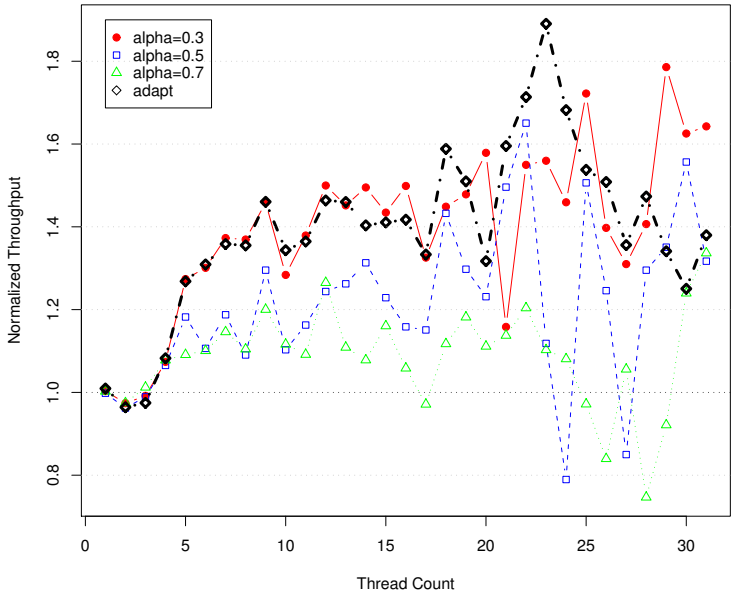



**Figure 15:** Automatic Tuning of the $\alpha$ Value (2-way SMP)

Although $\alpha$ was initially set to 0.5, we can see that the adaptive scheme more closely follows the performance of $\alpha = 0.3$, which is the best setting among the three constant $\alpha$-values. We also observed that no matter the initialization value, $\alpha$ 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.

# CHAPTER V

# OPERATING SYSTEM INTEGRATION

Implementing ATS in an STM actually amounts to a user-level thread scheduling. Since synchronization overhead could sometimes degrade the performance, it would be better to delegate transaction scheduling to a central module that is inherently synchronized: the OS scheduler. Implementing transaction scheduling in the OS also has the benefit that the OS can assume an active role to optimize TM performance regardless of the optimization approach taken in the actual TM implementation. This is also true for the HTM. Although we have shown in Section 4.1 that a simple transaction scheduling could bring about significant performance improvement, the scheduling algorithm was confined to be simple so that it could be implemented in hardware. Implementing ATS in the OS alleviates this limitation, thus enabling a more sophisticated scheduling algorithm.

To integrate ATS, the OS should first introduce an exclusive scheduling class for threads currently executing transactions: *i.e.*, the TX class. Threads of the TX class should be favored in scheduling over other threads. Moreover, there should be a dedicated queue to keep track of those threads that belong to the TX class, such as the real time queue found in Solaris [27]. Ideally, the OS should maintain one such queue for each process.

On the hardware side, the processor should dedicate one of its interrupt number to trap into the OS scheduler when necessary. If we assume to implement the same contention intensity detection algorithm as in Section 3.1, the processor should also provide two registers for the $\alpha$ value and the threshold. Moreover, there should be a bit denoting whether to trap to scheduler or not. This could be implemented as

a bitmask that could be set or cleared by the OS. Note that these registers and the bitmask are part of the thread context and must be backed up and restored across context switches. Figure 16 and 17 shows the interaction of the proposed hardware / OS interface.

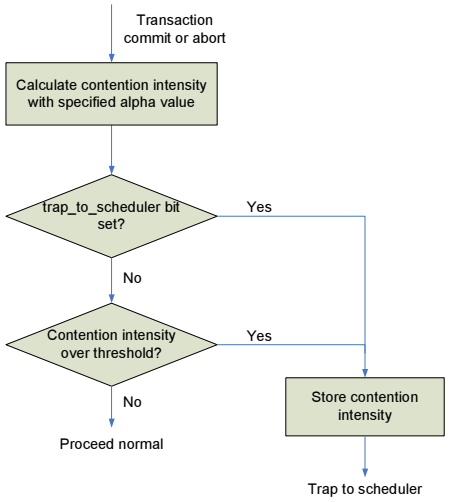



**Figure 16:** Processor Transition

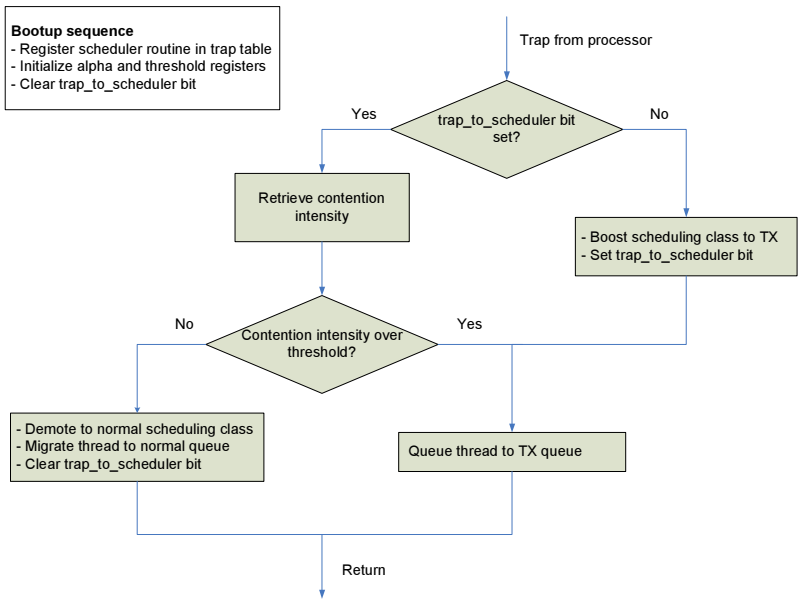



**Figure 17:** Operating System Transition

Upon bootup, the OS registers the transaction scheduler routine in the trap table. In addition, it initializes the $\alpha$ and the threshold register, and clears the

trap_to_scheduler bit. When the trap_to_scheduler bit is not set, the processor begins, commits, and aborts a transaction as usual. Meanwhile, the processor updates contention intensity upon every transaction's commit or abort.

When the contention intensity is below the threshold, it proceeds as normal, otherwise, the processor stores the contention intensity in a predetermined location (either in memory or a software trap argument register), then traps to the scheduler. Note that trapping to the OS in this situation does not degrade performance significantly since the processor would have been losing performance due to high contention anyways.

The scheduler first checks the trap_to_scheduler bit upon trap. 1) If the bit is not set, this is the first time the thread consulting the scheduler. In this case, the scheduling class of the thread is raised to TX, the trap_to_scheduler bit is set, and the thread gets queued in the TX queue. 2) If the bit is set, the thread has already consulted the scheduler earlier. The scheduler then decides whether the thread should remain in the TX class by examining the contention intensity. If the value is still above threshold, the thread gets queued at the back of the TX queue. If the value is below threshold, the scheduler demotes the thread to normal scheduling class, migrates it to a normal scheduling queue, and finally clears the trap_to_scheduler bit.

Once the trap_to_scheduler bit is set, the start of a transaction is controlled by the OS thread scheduling policy. Moreover, when the bit is set, the processor should trap to the scheduler when a transaction commits or aborts.

# CHAPTER VI

# RELATED WORK

TM [19, 21] 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 [34] and speculative synchronization from Martnez and Torrellas [25]. Nonetheless, speculative methods potentially suffer from backfire if speculation fails frequently. Our thesis minimizes this negative effect on TM systems.

Other approaches to maximize the performance of TM systems include contention managers [18, 40]. 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 [47]. Rather than to take action after the contention has been detected, our method fundamentally reduces the contention itself. Bai *et al.* [3] 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 [43]. Under admission control, an OS can delay the admission of the work until the system utilization subsides below some threshold.

Previously proposed retry construct [15] 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.

There have been several HTM implementations [19, 14, 1, 30, 26, 4, 7, 46]. STM systems have seen similar releases [18, 16, 12, 23, 22, 37, 11, 13]. The current trend in HTM research is to accelerate STM with hardware components [38, 42, 29], or to implement a hybrid [10, 20]. As long as there are transaction aborts, ATS can be implemented to improve performance. The queue-based transaction scheduler is particularly suitable for those TM systems that stall all other transactions in favor of one overflown transaction or a transaction performing I/O or a system call [4, 5].

The energy aspect of transactional memory has already been studied [32]. Nonetheless, the authors in [32] sacrifice performance in favor of energy reduction by blindly serializing all the transactions. Due to its adaptive nature, our scheme is expected to save energy while increasing performance.

# CHAPTER VII

# CONCLUSION

In this thesis, 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 improve performance for both generic HTM and STM systems. In our experiments, the maximum performance speedup on an HTM system reaches 1.97x. Relative performance speedup's on STM are from 1.3x to 1.5x while the speedup of the peak performance is 5.9x.

# REFERENCES


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

[2] Baek, W., Minh, C. C., Trautmann, M., Kozyrakis, C., and Olukotun, K., "The OpenTM transactional application programming interface," in *PACT '07*, pp. 376–387, 2007.

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

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

[5] Blundell, C., Lewis, E. C., and Martin, M., *Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions*. Technical Report TR-CIS-06-09, University of Pennsylvania, May 2006.

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

[7] Ceze, L., Tuck, J., Torrellas, J., and Cascaval, C., "Bulk disambiguation of speculative threads in multiprocessors," in *ISCA-33*, pp. 227–238, 2006.

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

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

[10] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D., "Hybrid transactional memory," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[11] Dice, D., Shalev, O., and Shavit, N., "Transactional locking II," in *Proceedings of the 20th International Symposium on Distributed Computing*, 2006.

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

[13] Gottschlich, J. and Connors, D. A., "DracoSTM: A practical C++ approach to software transactional memory," in *Proceedings of the 2007 ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD)*, October 2007.

[14] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K., "Transactional memory coherence and consistency," in *ISCA-31*, pp. 102 – 113, June 2004.

[15] Harris, T., Marlow, S., Jones, S. P., and Herlihy, M., "Composable memory transactions," in *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[16] Harris, T. and Fraser, K., "Language support for lightweight transactions," in *OOPSLA '03*, pp. 388 – 402, 2003.

[17] Herlihy, M., Luchangco, V., and Moir, M., "Obstruction-free synchronization: Double-ended queues as an example," in *ICDCS-23*, p. 522, 2003.

[18] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N., "Software transactional memory for dynamic-sized data structures," in *Proceedings of the Symposium on Principles of Distributed Computing*, pp. 92 – 101, 2003.

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

[20] Kumar, S., Chu, M., J. Hughes, C., Kundu, P., and Nguyen, A., "Hybrid transactional memory," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

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

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

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

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

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

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

[27] McDougall, R. and Mauro, J., *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Prentice Hall, 2006.

[28] Milovanovic, M., Ferrer, R., Unsal, O., Cristal, A., Martorell, X., Ayguade, E., Labarta, J., and Valero, M., "Transactional memory and OpenMP," in *International Workshop on OpenMP*, June 2007.

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

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

[31] Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A., "Supporting nested transactional memory in LogTM," in *ASPLOS-12*, pp. 359–370, 2006.

[32] Moreshet, T., Bahar, R. I., and Herlihy, M., "Energy reduction in multiprocessor systems using transactional memory," in *ISLPED '05*, pp. 331–334, 2005.

[33] Moss, J. E. B. and Hosking, A. L., "Nested transactional memory: Model and preliminary architecture sketches," in *the 2005 Workshop on Synchronization and Concurrency in Object Oriented Languages (SCOOL '05), held at OOPSLA*, October 2005.

[34] Rajwar, R. and Goodman, J. R., "Speculative lock elision: enabling highly concurrent multithreaded execution," in *MICRO-34*, pp. 294–305, 2001.

[35] Rajwar, R., Herlihy, M., and Lai, K., "Virtualizing transactional memory," in *ISCA-32*, pp. 494 – 505, June 2005.

[36] Ramakrishnan, R. and Gehrke, J., *Database Management Systems, 3rd Ed.* McGraw-Hill, 2002.

[37] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B., "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[38] Saha, B., Adl-Tabatabai, A.-R., and Jacobson, Q., "Architectural support for software transactional memory," in *MICRO-39*, pp. 185–196, 2006.

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

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

[41] Scott, M. L., Spear, M. F., Dalessandro, L., and Marathe, V. J., "Delaunay triangulation with transactions and barriers," in *Proceedings of the 2007 IEEE International Symposium on Workload Characterization*, pp. 107–113, 2007.

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

[43] Stankovic, J. A., "Admission control, reservation, and reflection in operating systems," *IEEE Bulletin of Technical Committee on Operating Systems and Application Environments (TCOS)*, vol. 10, no. 2, 1998.

[44] Wang, C., Chen, W.-Y., Wu, Y., Saha, B., and Adl-Tabatabai, A.-R., "Code generation and optimization for transactional memory constructs in an unmanaged language," in *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, pp. 34–48, March 2007.

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

[46] Yen, L., Bobba, J., Marty, M. R., Moore, K. E., Volos, H., Hill, M. D., Swift, M. M., and Wood, D. A., "LogTM-SE: Decoupling hardware transactional memory from caches," in *HPCA-13*, pp. 261–272, 2007.

[47] Zilles, C. and Baugh, L., "Extending hardware transactional memory to support nonbusy waiting and nontransactional actions," in *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.