Computer Science Department

Dynamic Prediction based Scheduling for TM

submitted in partial fulfillment
of the requirements for the degree of

Master of Science

Anmol Tomar

supervised by

Prof. Rachid Guerraoui

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Distributed Programming Laboratory
EPFL Switzerland

16 January 2009

Acknowledgments

I would like to thank my research advisor, Prof. Rachid Guerraoui, for his constant guidance throughout the course of my thesis. He provided me with ample time and encouragement to explore new ideas. I enjoyed his high expectations and the freedom to explore, which gave me great confidence in my work. I would also thank Aleksandar Dragojevic for his help in working with STM implementations. His ideas, especially in the email brainstorming sessions, helped me in maintaining a correct approach in my thesis.

I would then like to thank my family, Ma, Papa, and my little sister, Pinky, whose love and encouragement is an endless source of my strength, in all my endeavors. I also thank Amma, and especially Appa for his constant motivation for research during the course of my studies at EPFL. I would also thank my grandparents, other family members and family-in-law, along with my beautiful nieces Muskaan and Khushi, and the two little ones, Milee, and Dev, for their great affection and patience for the 1.5 years, when I have not been able to visit them.

Last but not the least, I would thank Vasu for his never-ending support and motivation, in all times.

Abstract

Transactional memory (TM) provides an intuitive and simple way of writing parallel programs. TMs execute parallel programs speculatively and deliver better performance than conventional lock based parallel programs. However, in certain scenarios when an application lacks scope for parallelism, TMs are outperformed by conventional fine-grained locking. TM schedulers, which serialize transactions that face contention, have shown promise in improving performance of TMs in such scenarios.

In this thesis, we develop a Dynamic Prediction based Scheduler (DPS) that exploits novel prediction techniques, like temporal locality and locality of access across repeated transactions. DPS predicts the access sets of future transactions based on the access patterns of the past transactions of the individual threads. We also propose a novel heuristic, called serialization affinity, which tends to serialize transactions with a probability proportional to the current amount of contention. Using the information of the currently executing transactions, the current amount of contention, and the predicted access sets, DPS dynamically serializes transactions to minimize conflicts. We implement DPS in two state-of-the-art STMs, SwissTM and TinySTM. Our results show that in scenarios where the number of threads is higher than the number of cores, DPS improves the performance of these STMs by up to 55% and 3000% respectively. On the other hand, the overhead of prediction techniques in DPS causes a performance degradation of just 5-8% in some cases, when the number of threads is less than the number of cores.

Contents

List of Figures

Chapter 1

Introduction

The saturation of Moore's law for uniprocessors has led to the widespread adoption of multiprocessors. Multiprocessors offer an extensive scope for boosting computing performance, due to the possibility to issue multiple instructions per cycle, on multiple cores. Thus, multicores have made parallel programming a dominant paradigm in computing today. In a parallel program, multiple threads execute concurrently on different cores of a multiprocessor machine, and access the underlying shared data. To prevent inconsistencies due to concurrent multiple thread accesses, shared data is conventionally protected with locking primitives like, semaphores and mutexes. Lock-based synchronization guarantees mutual exclusion of shared data, and locking algorithms (both in hardware and software) have been extensively researched to minimize latency and maximize throughput [MCS91, KBG97]. However, lock-based synchronization has several drawbacks, like problems of guaranteeing progress: deadlocks, livelocks; and problems arising due to negative influences of CPU scheduling: lock convoying, priority inversion, starvation. Moreover, while coarse-grained locking is easier to implement, it sacrifices performance. Fine-grained locking gives better performance, but complicates implementation. Thus, writing efficient and correct parallel programs using locks is a hard task. As concurrency bugs can manifest themselves in rare scenarios, it is difficult to debug parallel programs.

Researchers have also explored non-blocking synchronization on shared memory multiprocessors [MS96]. Non-blocking algorithms use hardware provided atomic read-modify-write operations, and avoid the classical problems of lock-based synchronization. But, non-blocking algorithms are extremely hard to design and to prove correct, making them a poor choice for non-expert programmers.

Transactional Memory (TM) is an emerging paradigm for simplifying the task of writing parallel programs. A TM enables a programmer to think in terms of coarse-grained code blocks that appear to execute atomically. Thus, TMs remove the burden of correct fine-grained locking from the programmer.

1.1 Transactional Memories

Inspired by how databases manage concurrency [EGLT76], TM was proposed by Herlihy and Moss [HM93] as a mechanism for writing efficient concurrent programs. TMs use lock-based[1] or non-blocking synchronization to avoid inconsistencies, and rely on some level of speculative execution to boost performance of parallel programs. The basic notion of execution in a TM is a transaction, derived from database transactions. A *transaction* is a sequence of memory accesses followed by a commit or an abort. A transaction guarantees the three ACI (atomicity, consistency, and isolation) properties. If a transaction *commits*, all its operations become visible to other threads. If a transaction *aborts*, none of the operations take effect, and the transaction restarts its execution. Transactional memories have been implemented in hardware [HM93, RG01, RG02,

[1] Note that a TM is designed by an expert, and thus, locks inside TMs are unlikely to cause classical problems that occur when programmers directly use locks for synchronization

HWC$^+$04, MBM$^+$06a, MBM$^+$06b, YBM$^+$07, BGH$^+$08], software [ST95, HLMI03, HF03, HMJH05, MIS05, MSH$^+$06, SATH$^+$06, DSS06, RFF07, DGK08], and as hardware-software hybrids [AAK$^+$05, RHL05, DFL$^+$06, KCH$^+$06, SSH$^+$07, MTC$^+$07]. A comprehensive treatment of transactional memories was done by Larus et al. [LR07].

1.1.1 Conflict detection

A conflict occurs when two transactions access the same data, and at least one of the transactions writes. Typically TMs run transactions optimistically, making the transactions prone to conflicts with each other. A conflict detection scheme is hence required for guaranteeing correctness in TMs. These schemes vary among different TMs, in their degree and nature of speculation. For example, different TMs detect read-write and write-write conflicts in different ways. Conflict detection schemes have been broadly identified under two categories - *lazy* and *eager* conflict detection. Eager schemes [MBM$^+$06a, RFF07] detect conflicts early and thus sacrifice sharing among transactions. For example, early detection of read-write conflicts shall avoid read-write sharing of data between transactions. Lazy schemes [DSS06, SSH$^+$07], on the contrary, detect conflicts later during the transaction, allowing sharing but wasting computational effort as some transactions get aborted after performing a lot of work. For example, a transaction which has read a stale version of an object needs to restart the whole transaction when it detects that the version is no longer fresh. There also exist hybrid conflict detection schemes. Specifically, a *mixed invalidation* [SMIS06] scheme employs eager write-write conflict detection and lazy read-write conflict detection. This scheme has shown to perform better [DGK08] than both eager and lazy conflict detection schemes.

1.1.2 Version management

The fact that a transaction running in a TM can possibly abort, necessitates managing different versions of the data at the same time. TMs use either lazy or eager version management. In lazy version management [HM93, SSH$^+$07], a thread creates a shadow (local) copy of all addresses written during a transaction. Upon commit of the transaction, the thread updates the global copy (in memory). If the transaction aborts, the thread throws away the local changes made during the transaction. In eager version management [MBM$^+$06a], a thread updates the global copy during a write. Upon commit, a thread has to do nothing, while upon an abort, a thread has to undo the writes made during the transaction's execution. Thus, eager version management makes commits faster, while lazy version management makes aborts faster.

1.1.3 Access granularity

The access granularity of a transactional memory defines the unit of memory over which read and write conflicts are detected. Generally, TMs implement either *object granularity*, which detects conflicts on objects, or *word granularity*, which detects conflicts on memory words. In object granularity [HLMI03], two transactions may conflict even if they access different members of an object. Word granularity [DSS06] is finer grained than object granularity, and provides better concurrency and hence, better performance in programs using data structures like arrays. At the same time, the finer granularity results in larger amount of required book-keeping which could degrade the performance in some cases. In hardware TMs, in which conflict detection is done using the cache coherence protocol, conflicts are often detected on the level of cache blocks. This is referred to as *block granularity.*

1.1.4 Correctness in TM

Different implementations of TMs satisfy different safety and liveness properties. The safety properties generally expected in TMs are *strict serializability* [Pap79] and *opacity* [GK08]. The property, strict serializability, carries from the database community. It states that any set of committed transactions which are executed by different threads, appear as if executed in some serial order, such that the order of non-overlapping transactions is preserved. The stronger property, opacity, is more relevant to TMs. It states that any set of transactions (committed or aborted) which are executed by different threads, appear as if executed in some serial order. The motivation behind opacity is to prevent non-committed transactions from observing inconsistent values, which could in turn result in unexpected program behavior, like array bound violations or infinite loops.

Moreover, TMs satisfy some liveness properties, which vary from one TM algorithm to another. The three liveness properties for TMs are *obstruction freedom* [HLM03], *livelock freedom* [AKH03], and *wait freedom* [Her91]. Obstruction freedom states that if a transaction executes in isolation, it eventually commits. Livelock freedom states that some transaction eventually commits. Wait freedom states that every transaction eventually commits.

1.1.5 Performance Measures

There are various ways of evaluating the performance of a TM. We briefly mention the common measures. The *throughput* of a TM is the number of committed transactions per unit of time. Another measure of performance, *number of aborts per commit* is used to quantify the amount of wasted work done by a TM. To improve performance, we maximize throughput, while we minimize the number of aborts per commit.

The performance of transactional memories is a critical issue for their adoption into mainstream parallel computing. Research has shown that in high contention scenarios, fine-grained locking outperforms TMs. Various strategies have been proposed and developed to boost the performance of TMs in such scenarios. The most common strategy studied in the TM literature is the notion of a contention manager.

1.2 Contention Managers

When a thread discovers an inconsistent access in a TM, either the thread facing the inconsistency, or the one that caused the inconsistency has to abort, so that the program executes correctly. To efficiently make this decision based on the runtime circumstances, a transactional memory employs a separate module called a *contention manager*. Extensive research has been done on various contention management schemes [SS05, GHP05, GHP06]. Here, we list down a few of these schemes.

- **Polite.** Polite contention manager [SS05] applies an exponential backoff scheme to resolve contention. Upon a conflict for an address (or object), a thread spins for a period of time that is exponential in the number of retries. On exceeding a maximum limit on the retries, the conflicting transaction is unconditionally aborted.

- **Karma.** This contention manager [SS05] tries to resolve conflicts based on the amount of work done by the transactions at the time of conflict resolution. The number of objects opened by a transaction is used as an estimate of the work done by the transaction. Threads gain cumulative priority values, as they open objects during a transaction. On commit of a transaction, the priority is reset to zero. On a conflict between two transactions, their priority values are compared, and the transaction with a lower priority value gets aborted. A transaction that aborts preserves its priority that accumulated during its run, so that it has higher chances of success after multiple aborts. Another contention manager, called *Polka*, is obtained by combining the Polite and Karma contention managers.

- **Greedy.** This contention manager [GHP05] works on the basis of timestamps. Each transaction gets a unique timestamp on its start. The timestamps are generated such that they increase monotonically over time. A transaction X aborts transaction X' on the following two conditions. (i) the timestamp of X is less than timestamp of X', or (ii) X' is waiting for another transaction.

- **Serializer.** The serializer contention manager [DHS08] resolves a conflict by removing a conflicting transaction from the core where it was running, and scheduling the removed transaction on the core of the other transaction involved in the conflict. The serializer contention manager ensures that two transactions never conflict more than once.

Although contention managers boost the performance of a TM, they play their role only after inconsistencies have been detected. So contention managers still do not avoid wasted work done by threads which are doomed to abort. Neither do contention managers allow threads to speculate conflicts and yield a processor to another non-conflicting thread.

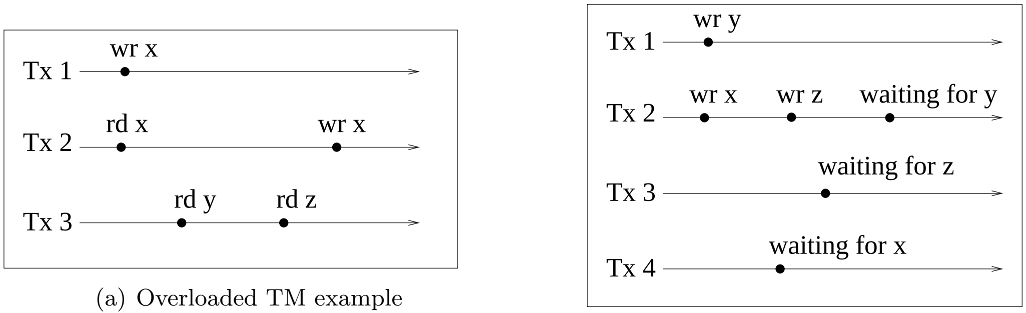
We propose a prediction based approach that predicts conflicts and avoids them before they actually happen. We give examples where prediction shall help to increase the performance of a TM.

1.2.1 Motivation for Prediction in TM

In this thesis, we distinguish between overloaded and underloaded TMs. We define a TM as *overloaded* if the number of threads in the TM is higher than the number of cores. Moreover, we define a TM as *underloaded* if the number of threads in the TM is less than or equal to the number of cores.

Consider the situation of an overloaded TM, as illustrated in Figure 1.1(a). Let the number of cores be two, with three threads running their transactions, Tx 1, Tx 2, and Tx 3. Transactions Tx 1 and Tx 2 are running on one core each, leaving no core available for Tx 3 to run. Tx 2 finds out later during its execution that it conflicts with Tx 1 on the write of variable x. If the transaction Tx 3 had been scheduled in place of either Tx 1 or Tx 2, it would not have conflicted with the other running transaction, resulting in a better TM performance.

A similar situation can be observed in an underloaded system (shown in Figure 1.1(b)). Transactions Tx 1, Tx 2, Tx 3, and Tx 4 are running on a 4-core machine. Thus, there are enough cores for running all 4 transactions concurrently. Transaction Tx 1 is writing to variable y, whereas transaction Tx 2 is writing to variables x and z. Tx 2 later wants to write to variable y, when it detects its conflict with Tx 1 on y. Also, Tx 3 and Tx 4 cannot run due to their conflicts with Tx 2, as Tx 2 has written to x and z. This situation could be avoided if Tx 1, Tx 3, and Tx 4 were scheduled to run in parallel, whereas Tx 2 were not scheduled.



(a) Overloaded TM example

(b) Underloaded TM example

Figure 1.1: Motivating examples for prediction in TM

In both these examples, a contention manager would act only after a conflict has occurred. Indeed, a scheduling technique which could predict a conflict and avoid it from occurring shall perform better. This thesis introduces such a scheduling policy, called the Dynamic Prediction based Scheduler (DPS). DPS is an online scheduling scheme that *learns* about the potential conflicts in a TM and acts apriori to prevent those conflicts.

1.3 Dynamic Prediction based Scheduler

We introduce a novel prediction technique, called *temporal locality*, in TMs. The principle of temporal locality has been widely studied and applied to different domains of computer science [Den68, DS72, Smi82, ABCdO96]. Specifically for TMs, the property of temporal locality states that similar addresses[2] are accessed across multiple transactions of a thread. This implies that addresses which are frequently accessed in the past few transactions of a thread are more likely to be accessed in future transactions of that thread. We empirically observe high temporal locality in read accesses of a thread. So, DPS predicts read sets using temporal locality. DPS maintains the read set of past few committed transactions of a thread in a set of Bloom filters. Then, DPS checks membership of an address in these Bloom filters [Blo70], and uses a confidence measure to predict whether the address shall be read in future transactions. To predict write sets, DPS uses the write set of an aborted transaction to predict the write set of the next restarting transaction of the thread. Note that the write set of an aborted transaction may not provide an exact prediction. This is because the underlying data structure, which the transactions concurrently work on, might change between the abort and the subsequent restart of a transaction. Moreover, in some cases transactions could be non-deterministic. But our empirical results on various TM benchmarks show sufficient accuracy of the write set prediction.

[2] In this thesis, we use the term *address* for words in word-based TMs, and for objects in object-based TMs

DPS uses these predicted read and write sets in conjunction with the information of the currently executing transactions to prevent conflicts. DPS checks whether any address in the predicted read and write sets of the starting transaction is being written by any other currently executing transaction. In case an address is being written, the scheduler serializes the starting transaction. Otherwise, the transaction is allowed to execute, as it would, without the scheduler. Note that DPS can be integrated with any TM that uses visible writes, that is, other threads know when a particular thread writes to an address.

To keep DPS competitive in low contention scenarios and in underloaded cases, it is important that DPS serializes threads only if contention is high. To detect when a thread faces high contention, DPS maintains a success rate of every thread, and activates the prediction and serialization technique only if the success rate of the thread falls below a certain threshold. Another interesting heuristic we use in our work is called *serialization affinity.* We observe that serializing a transaction with a probability proportional to the contention in the TM provides better performance. This heuristic is based on the observation that serializing a transaction is relatively more helpful when a large number of threads access similar addresses and compete for a small number of cores. This is obvious, as the lack of proper scheduling causes many conflicts in these cases.

We integrate DPS with two state-of-the-art STMs, SwissTM and TinySTM. The motivation behind DPS has been to increase the throughput of the state-of-the-art STMs in overloaded cases, while not degrading the performance of those STMs in underloaded cases. Our scheme proves successful. For overloaded cases, we obtain throughput gains of up to 50% for SwissTM, and around 3000% for TinySTM[3] on different workloads of STMBench7. On the other hand, the overhead of prediction techniques incurs a performance loss of 5-8% in underloaded cases, on STMBench7. In a way, DPS approximates an optimal scheduler

[3] We use SwissTM with preemptive waiting, and TinySTM with busy waiting.

by scheduling transactions which do not face any conflict, while postponing the execution of transactions which are likely to conflict.

Preliminary TM schedulers in the literature rely on coarse measures to activate scheduling. For example, Yoo et al. [YL08] use a measure of contention intensity in their ATS scheme which is similar to our measure of success rate. However, in ATS, when the contention intensity of a thread increases beyond a threshold, the thread is forced to serialize. We show examples in Chapter 2 that such coarse serialization inhibits parallelism. We discuss other TM schedulers [RHP$^+$07, DHS08] in Chapter 5.

1.4 Organization of the thesis

Chapter 2 introduces basic TM schedulers, and the problems associated with them. We present our variant of a basic TM scheduler, called the Pool scheduler. We also discuss examples why basic scheduling policies are too coarse. Chapter 3 explains our dynamic prediction based scheduler. We first describe different prediction strategies of DPS, followed by empirical results that show the accuracy of the predictions. Later, we formally present our DPS scheme. Chapter 4 covers the implementation details and the experimental results. We show how we integrate DPS with SwissTM and TinySTM. We give results of SwissTM and TinySTM running with our DPS scheme, on different benchmarks: STMBench7, STAMP, and microbenchmarks. Chapter 5 covers the related work in this field, and Chapter 6 concludes the thesis.

Chapter 2

A Basic TM Scheduler

Transactional memories are highly performance-oriented, and rely on speculative execution to boost throughput. Speculative execution often causes conflicts, which a TM resolves using a contention manager. Although contention managers have been very successful in boosting TM performance, they have a downside that they act only after a conflict has been detected. This wastes effort of one of the conflicting transactions, as it needs to be aborted.

TM schedulers offer a powerful means to boost the performance of TMs. A *transactional memory scheduler* is a policy that decides when a particular transaction executes in a TM. Preliminary TM schedulers [YL08] proposed in the literature defer a transaction from running, if it has aborted very often in the recent past. Some TM schedulers [DHS08, ALK$^+$09] reschedule a transaction on the core of the other conflicting transaction in order to avoid repeated conflicts. TM schedulers have shown performance benefits over contention managers [YL08, DHS08]. A TM scheduler especially helps in overloaded cases, i.e., when the number of threads is higher than the number of cores available to the TM. Generally, the performance of TMs (with or without contention managers) does not scale well in overloaded cases. We argue that future practical systems will tend to be overloaded, as these systems will have additional tasks running on the multiprocessor machine, which shall use the same cores. Moreover, the granularity of parallelism is not restricted to the level of transactions, and systems are already being built that exploit parallelism within a transaction. This concept of intra-transactional parallelism has been formalized as *parallel nesting* [AFS08]. If we assume that the number of threads in a TM is upper bounded by the number of cores, we shall hinder parallel nesting. Thus, to scale TM to practical systems and allow features like parallel nesting, we need strategies like TM schedulers to boost performance in overloaded scenarios. We now present a basic TM scheduler, and discuss the drawbacks of such schedulers.

2.1 Pool Scheduler

We first describe a basic TM scheduler, called the *Pool scheduler*, which serializes the transactions of threads facing high contention. The Pool scheduler is similar to the ATS scheme [YL08] in the literature.

We briefly discuss the algorithm for our Pool scheduler. We define *success rate* of a thread as a measure of the ratio of commits of the thread, to its aborts. We measure the success rate of a thread as the moving average over successive commits and aborts of the thread. The threads whose success rate is lower than a particular threshold are added to a pool of waiting threads, which wait for a common mutex. When the mutex is available, one of the threads grabs the mutex and continues execution. Note that the waiting threads form a pool of threads, rather than a queue [YL08], which saves the overhead of inserting a thread at a particular position in the queue. We illustrate the Pool scheduler algorithm in Algorithm 1.

We present examples to show that it is not always beneficial to serialize threads in the

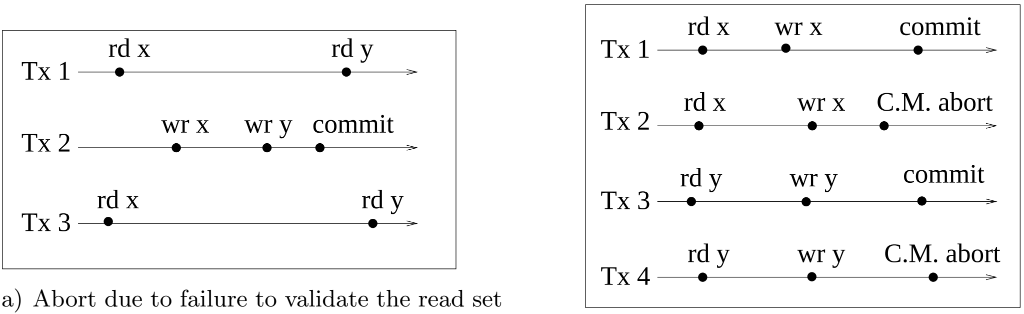
Algorithm 1 *Pool Scheduler*

```
Variables: global_lock is a lock variable shared between threads. succ_threshold and
success are constant integers, succ_rate is an integer variable per thread.
On transaction start
    if succ_rate < succ_threshold then
      lock global_lock
    endif
On transaction commit
    succ_rate := (succ_rate + success)/2
    if own global_lock then
      unlock global_lock
    endif
On transaction abort
    succ_rate := succ_rate/2
    if own global_lock then
      unlock global_lock
    endif
```

face of contention. In Figure 2.1(a), both transactions Tx 1 and Tx 3 read variable x and then continue their executions. Transaction Tx 2 writes to variables x and y, after the reads of x by Tx 1 and Tx 3, and then Tx 2 commits. Tx 1 and Tx 2 each read y next, and hence are bound to abort due to an inconsistency in the values read. A coarse serialization policy would serialize Tx 1 and Tx 3, which are not conflicting, and thus hinder parallelism.

Another example is that of a TM with a contention manager, as shown in Figure 2.1(b). Transactions Tx 1 and Tx 2 conflict, as they both write to the variable x. The contention manager decides to abort Tx 2, whereas allows Tx 1 to commit. Similarly, the contention manager aborts Tx 4 and allows Tx 3 to commit, when they conflict on variable y. A basic scheduling policy would serialize Tx 2 and Tx 4, though the accesses of Tx 2 and Tx 4 are completely independent and they could run in parallel, when they restart.

Based on the above examples, we now argue that though the Pool scheduler (or any basic serializing TM scheduler) serializes a thread only when its success rate falls below a certain threshold, such a scheduling policy is fairly coarse. Threads may abort due to different reasons, like failure to validate the read set, or failure to lock an object, or due to a decision of a contention manager. However, the decision to serialize a thread is justified only in the case when the thread aborts due to a failure to obtain a lock. Failure to validate the read set occurs due to the commit of another transaction, a scenario which may not repeat again. Similarly, the decision of a contention manager to abort a thread may be based on conditions that may not repeat again. The decision of the Pool scheduler to serialize transactions in these cases may hinder performance. Moreover, we observe that once a thread gets into the thread pool for serialization, it can only undergo execution via the path of mutex acquisition. If the cause of the conflict ceases to exist in future, the thread can very well exploit more parallelism than in the case of serial execution. On the other hand, a scheme that uses the information of the currently executing transactions to dynamically serialize transactions, could decide to normally execute the aborting transactions. This further motivates the need of prediction and dynamic serialization in TMs.



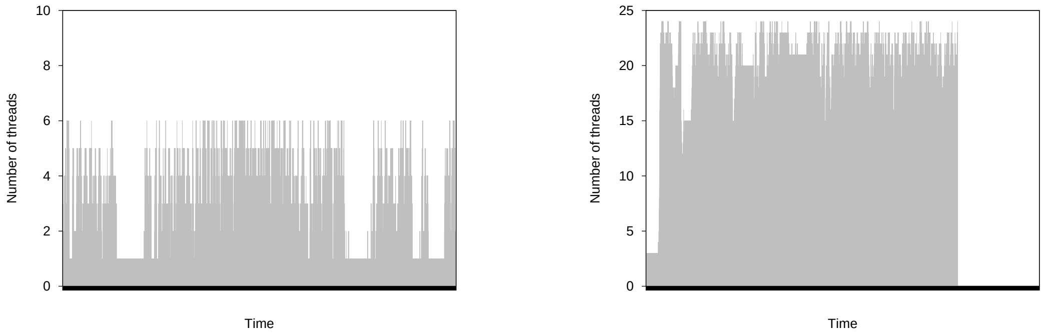
(a) Abort due to failure to validate the read set

(b) Abort due to contention manager

Figure 2.1: Motivating examples for dynamic serialization in a TM

2.2 Analysis of serializing behavior

Apart from the above examples, we also observe that basic TM schedulers serialize transactions even in low contention scenarios. To understand the performance tradeoff, we



(a) Underloaded system (6 threads on 8 cores) (b) Overloaded system (24 threads on 8 cores)

Figure 2.2: Serializing behavior in Pool-SwissTM

integrated the Pool scheduler in SwissTM[1] and analyze the performance on the STMBench7 benchmark. Performance results show that the Pool scheduler-enabled SwissTM (Pool-SwissTM) outperforms the base SwissTM in highly overloaded cases by up to 80%. However, the Pool-SwissTM suffers performance loss of up to 25% in underloaded cases, as compared to base SwissTM. This performance comparison suggests that serialization is more useful in highly overloaded TMs. In Figure 2.2, we show the serialization behavior of Pool-SwissTM in underloaded and overloaded cases. In Figure 2.2(a), we show how the number of threads waiting for serialization varies over time when the Pool-SwissTM is executed with 6 threads on an 8 core machine. We observe that the number of waiting threads is mostly between 4 and 6. In this case, the performance of the base SwissTM is better than the performance of Pool-SwissTM. Hence, we discourage serialization in underloaded TMs. In Figure 2.2(b), we show the serialization behavior in Pool-SwissTM when 24 threads are executed on an 8 core machine. We observe that the number of waiting threads remains very high, from 20 to 24 most of the time. In such highly overloaded cases, the Pool-SwissTM performs better than the base SwissTM. Thus, we encourage serialization in such scenarios. This points us to a heuristic called *serialization affinity*, which states

[1] SwissTM and STMBench7 are described in Chapter 4. The performance results are shown in Figure 4.1, 4.2, and 4.3.

that the probability of serializing a transaction should be proportional to the amount of contention. We use this heuristic in DPS to get good overall performance.

Chapter 3

Dynamic Prediction based Scheduler

We develop novel prediction techniques, like temporal locality and locality across repeated transactions. Using these prediction techniques, we predict access sets of future transactions. We develop a scheduling scheme which uses the predicted access sets, the heuristic of serialization affinity, and the information of currently executing transactions to dynamically serialize transactions. We show that our scheduling scheme improves the throughput of existing STMs in highly overloaded cases, while causing minimal losses due to overheads in underloaded cases.

A *clairvoyant* TM is one which knows apriori, which addresses a transaction will access. Developing an optimal scheduler for even a clairvoyant TM is an NP-hard problem. Of course, TMs are not clairvoyant, as transactions work upon complex data structures, which change over time. So, we try to perform two approximations: first, we predict accesses of the transactions of a TM to approximate a clairvoyant TM, and second we try to develop a scheduling policy that approximates the optimal TM scheduler.

3.1 Predicting accesses in a TM

We observe that a thread does not access a completely random set of addresses across its transactions. We explore this access behavior of each thread, and approximate clairvoyance in the TM by predicting the transactional access sets in our scheduler. We categorize the idea of prediction into read set prediction and write set prediction.

3.2 Read set prediction

We introduce the notion of *temporal locality* to predict the read sets of future transactions. Temporal locality is well-understood and utilized to boost the performance of the underlying systems in various domains of computer science, for example, memory hierarchies [Den68, DS72, Smi82], and web-caches [ABCdO96].

3.2.1 Temporal locality

Our empirical observations on several TM benchmarks, like STMBench7 and STAMP, show that multiple consecutive committed transactions of a thread access similar addresses. We believe that this is a result of the design of data structures. Data structures usually have a fixed way of traversal, which makes some memory addresses frequently accessed across transactions. Although data structures change over time due to new additions and deletions, temporal locality can indeed be observed across a significant window of adjacent transactions.

We define the *predicted read set* of a transaction as the set of addresses predicted to be read by the transaction. This prediction is based on the read accesses of the last few transactions.

We validate the idea of temporal locality by evaluating the accuracy of our predicted read set. Let T be the set of threads. Let X_t be the set of transactions of thread $t \in T$. For a transaction $x \in X_t$, let $pred_x$ be the predicted read set and rs_x be the actual read set of x. Then, the prediction accuracy of $pred_x$ is measured by the formula

$$\frac{\sum_{t \in T} \sum_{x \in X_t} |pred_x \cap rs_x|}{\sum_{t \in T} \sum_{x \in X_t} |pred_x|}$$

We run our experiments on STMBench7 benchmark with SwissTM. However, the idea of temporal locality is general, and can be observed in any benchmark with any TM. We show the results under three workload types[1] : read-dominated, read-write, and write-dominated, in Figure 3.1. We observe that the accuracy of read prediction is fairly high for read-dominated workloads, and decreases as the workload contains a higher proportion of writes. This is because, as the number of writes increase, the data structure is more likely to change across transactions, which decreases the temporal locality. Moreover, we observe that, in a given workload, the accuracy of the read prediction does not vary as the number of threads increases.

3.3 Write set prediction

The idea of temporal locality works with significant accuracy for predicting the read sets, but does not help in predicting the write sets. This is due to the fact that transactions have large read sets, but small write sets. The chance that a thread writes to the same addresses across multiple consecutive transactions is low. So, we need a different prediction strategy for the write set.

We use the fact that when a transaction repeats, its write set mimics the write set of

[1] SwissTM and STMBench7 are described in Chapter 4. The three workload types for STMBench7 specify the percentage of read-only operations, which is 90% for read-dominated, 60% for read-write, and 10% for write-dominated workloads.

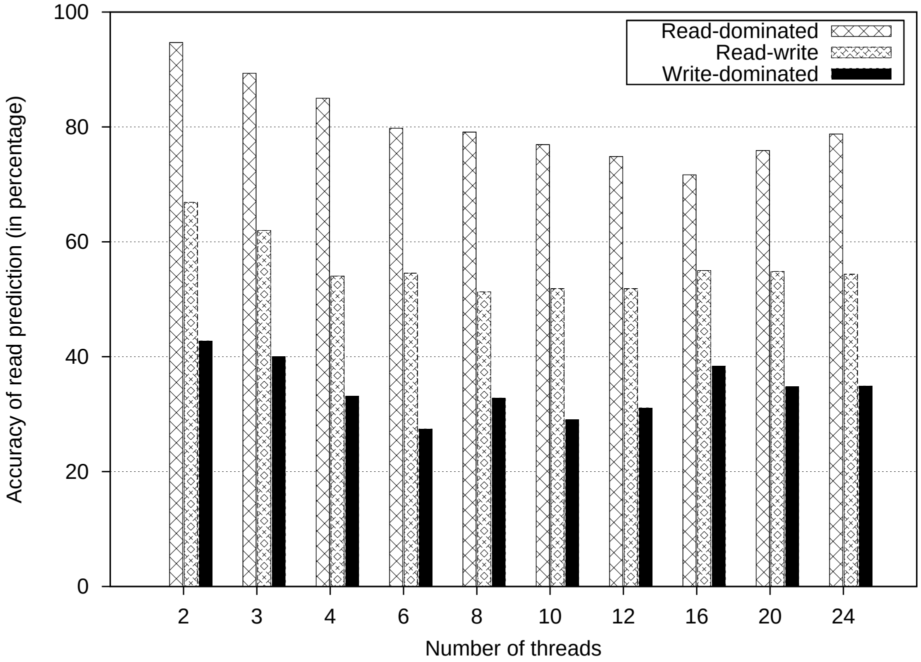


Figure 3.1: Accuracy of read prediction of DPS when measured on SwissTM on STMBench7, across the workload types in STMBench7

the immediately previous aborted transaction. As the underlying benchmark is unlikely to undergo drastic changes before the aborting transaction restarts execution, the writes of the aborting transaction form a good prediction for the write accesses of the restarting transaction. This property allows us to predict, upon an abort of a transaction, what addresses will be accessed in the next attempt of the transaction. We investigate this property in different benchmarks and show the results in Figure 3.2. Our experiments reveal that the write set prediction is fairly accurate across all workloads and with any number of threads.

Note that the idea of temporal locality allows read set prediction to work across committed and aborted transactions. On the other hand, write set prediction works solely across aborted transactions.

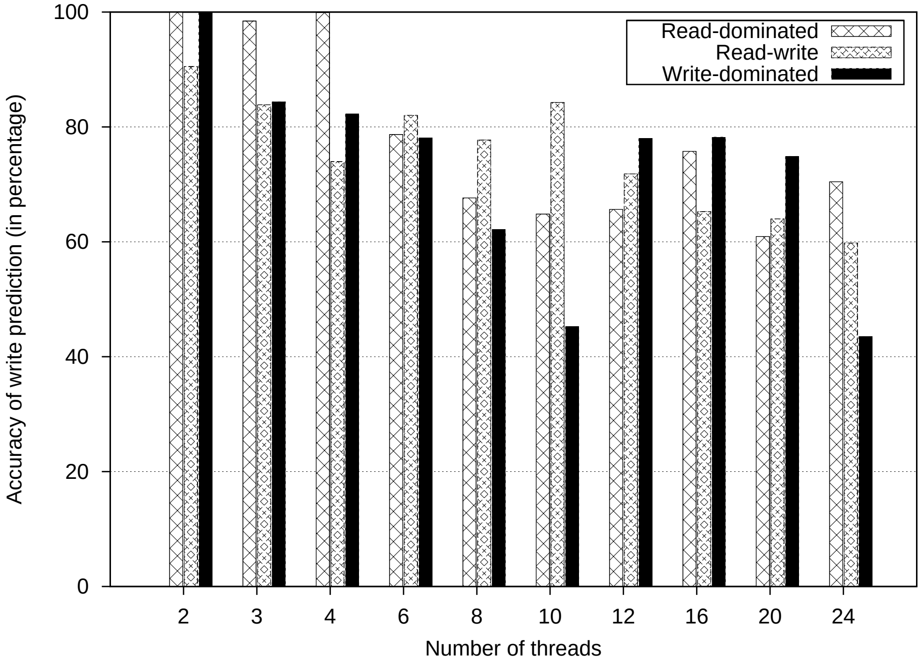


Figure 3.2: Accuracy of write prediction of DPS when measured on SwissTM on STMBench7, across the workload types in STMBench7

3.4 DPS scheme

DPS uses the above prediction techniques to predict the read and write sets. Using these predictions with the information of the currently executing transactions and the current contention, DPS dynamically serializes transactions.

3.4.1 Algorithm

We now present the algorithm for the dynamic prediction based scheduler. The schematic for DPS is shown in Figure 3.3. Like the Pool scheduler, DPS maintains the success rate of each thread. The *success rate* succ_rate is a measure of the ratio of the number of committing to the number of aborting transactions. The success rate of a thread is

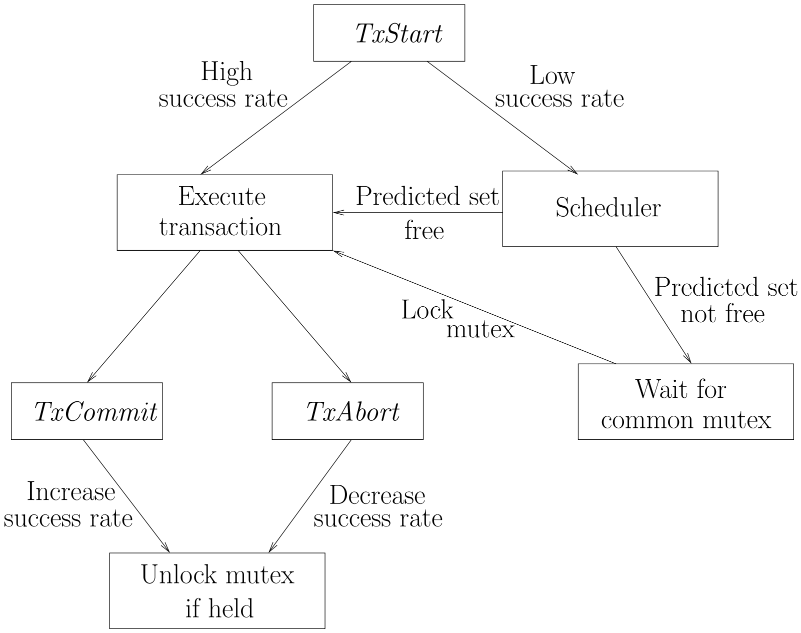


Figure 3.3: Schematic of the DPS scheduler

modified on commit and abort of a transaction. When the success rate of a thread falls below a certain threshold, the serialization policy of DPS gets activated.

The DPS scheduler uses a set of Bloom filters per thread to represent the thread's read set of last few transactions in a conservative manner. Bloom filters allow a fast means to insert addresses, and to check the membership of an address. When a transaction executes and reads an address, the scheduler checks the membership of the address in the Bloom filters corresponding to the past few transactions of the thread. If the address has been frequently accessed in the past, then the address is added to the predicted read set of the next transaction by the thread. We use a parameter called locality_window, which specifies the number of previous transactions that are used for prediction. Formally, let T_0 be a transaction that is currently running. Let T_{-i} be the last i^{th} transaction that committed before T_0. Let w be the locality window. Then, the scheduler maintains a set of bloom filters $\{\mathsf{bf}_1, \ldots, \mathsf{bf}_w\}$ for each thread, where bf_i is used to store the read set of transaction T_{-i}. With each bloom filter bf_i, we associate a confidence weight $\mathsf{confidence}_i$. During the execution of a transaction T_0 of a thread, we compute the predicted read set $\mathsf{pred\_read\_set}$ of T_1 as follows. On a read access of an address a by T_0, we compute the confidence of a as $\sum_{0<i\leq w} \mathsf{confidence}_i.\delta_i$, where $\delta_i = 1$ if a is in bf_i, and $\delta_i = 0$ if a is not in bf_i. All addresses whose confidence exceeds a threshold value are added to the predicted read set of T_1.

For the write set prediction, the DPS scheduler uses the write set of an aborting transaction. Thus, if T_0 aborts, then the write set of T_0 becomes the predicted write set $\mathsf{pred\_write\_set}$ of the next transaction T_1 of the thread. If T_0 commits, then T_1 starts with an empty predicted write set. Note that the scheduler is active only when the success rate has been below a certain threshold. Thus, the fact that the first attempt of a particular transaction has no predicted write set is not a major concern.

We now describe how the DPS scheduler acts for a thread, when the thread starts a transaction in the case that the success rate of the thread is less than a certain threshold. First, the DPS scheduler applies the serialization affinity heuristic. The scheduler first observes the number of threads $\mathsf{wait\_count}$ waiting for serialization, and decides to use the prediction scheme with probability proportional to $\mathsf{wait\_count}$. If the scheduler decides to forgo the prediction scheme, then the thread starts the transaction normally, as the base STM would. In case the scheduler decides to use the prediction scheme, the thread first checks whether some address in the predicted read set or the predicted write set is being written by any other thread. If there is indeed such an address, then the scheduler decides to serialize the starting transaction. In such a case, the thread waits to lock the global mutex $\mathsf{global\_lock}$ before starting the transaction. The modifications to the start procedure have been shown in Algorithm 2, and the modifications to the read, write, commit, and abort procedures have been shown in Algorithm 3.

Algorithm 2 *Dynamic Prediction based Scheduling*

Variables: global_lock is a lock variable shared between threads. succ_threshold is a constant integers, succ_rate is an integer variable per thread. read_pred and write_pred are addresses, pred_read_set is the predicted read set per thread, pred_write_set is the predicted write set per thread, and wait_count is an integer (initially 0).

```
On transactional start
    if succ_rate < succ_threshold
      generate a random number $r$ between 1 and 32
      if $r <$ wait_count then
        for each address read_pred in pred_read_set
          if read_pred is being written by some other thread then
            lock global_lock
            atomically increment wait_count
            break
          endif
        endfor
        if not owner of global_lock then
          for each address write_pred in pred_write_set
            if write_pred is being written by some other thread then
              lock global_lock
              atomically increment wait_count
              break
            endif
          endfor
        endif
      endif
    endif
    if last transaction was committed then
      remove all addresses from pred_read_set
    endif
    remove all addresses from pred_write_set
```

Algorithm 3 *Dynamic Prediction based Scheduling (contd.)*

Variables: bf_i is the bloom filter of the last i^{th} transaction, locality_window is the size of the set of bloom filters per thread, confidence is an integer (initially 0), confidence_threshold is the threshold confidence value, success is a constant integer, and *addr* and last_conflict are addresses. (other variables as in Algorithm 2).

```
On transactional read of $addr$
    if ($addr \notin \mathsf{bf}_0$) then
      add $addr$ to $\mathsf{bf}_0$
      $i = 1$
      while $i <$ locality_window do
        if ($addr \in \mathsf{bf}_0$) then
          confidence = confidence + $c_i$
        endif
      endwhile
      if confidence $\geq$ confidence_threshold, then
        add $addr$ into pred_read_set
      endif
    endif
On transactional write of $addr$
    if going to abort because $addr$ being written by some other thread then
      last_conflict := $addr$
    endif
On transactional commit
    succ_rate := (succ_rate + success)/2
    if own global_lock then
      unlock global_lock
      atomically decrement wait_count
    endif
On transactional abort
    copy write set of transaction into pred_write_set
    if last_conflict is not empty then
      add last_conflict to pred_write_set
      set last_conflict to empty
    endif
    succ_rate := succ_rate/2
    if own global_lock then
      unlock global_lock
      atomically decrement wait_count
    endif
```

Chapter 4

Implementation and Results

We implemented our dynamic prediction based scheduling scheme on top of two STMs, SwissTM and TinySTM. We evaluated the performance of the DPS-enabled SwissTM and DPS-enabled TinySTM, and compared them to the respective base STMs. DPS considerably improves the performance of these STMs in overloaded scenarios. Moreover, DPS is a general scheme which can be easily implemented in various TMs.

4.1 Evaluation framework

We chose two STM systems to demonstrate the performance gain obtained with our DPS scheme. By applying DPS in two systems, we attempt to show that the DPS scheme indeed improves the throughput of STMs, while it is not tightly coupled to any particular STM. Although, we have chosen STMs to demonstrate DPS, the scheme can indeed be implemented in hardware TMs, as we show in Section 4.6.

We performed our experiments on a 4 processor dual-core AMD Opteron 8216 2.4 GHz with 1024 KB cache, which gives us 8 cores for our experiments. We worked with several benchmarks to illustrate that coupling a TM with DPS generally boosts performance.

4.1.1 Microbenchmarks

Microbenchmarks have long been used to test STM performance. The red-black tree is a popular TM benchmark that provides simple transactional operations like insert, lookup, and delete on the red-black tree data structure. Although microbenchmarks do not provide a good representative of real world applications, they do help to analyze the complexity of the TM. We utilize this simple microbenchmark to prove that our scheme does not exhibit significant overheads over the conventional STMs.

4.1.2 STMBench7

STMBench7 is an object-based benchmark, which provides complex and realistic workloads for analyzing STM performance. STMBench7 comprises of a large number of operations on a shared data structure, and intends to represent complex real world concurrent applications. The spectrum of operations range from very short operations to very long ones, and from read-only to update operations. The data structure used by STMBench7 is many orders of magnitude larger than typical STM benchmarks. Three types of workloads, *read-dominated*, *read-write*, and *write-dominated* can be modeled in STMBench7. We perform experiments on all these workload types. STMBench7 allows long traversals to be turned on or off. We turn off long traversals for our experiments.

4.1.3 STAMP benchmark suite

STAMP benchmark suite has been designed with the objective of representing a variety of application domains that may benefit from TMs. STAMP consists of eight such applications from various domains: computational biology, security, engineering, and machine learning. Different benchmarks are characterized by their transaction length and the size of access sets. We briefly look at the various benchmarks in the STAMP benchmark suite.

Bayes implements an algorithm for Bayesian network learning. This benchmark has a high amount of contention due to frequently changing sub-graphs, and long transactions with relatively large read and write sets. *Genome* is an application for gene sequencing, which matches numerous DNA segments to construct the original source. The transactions, read and write sets are all moderate sized. The execution time is predominantly transactional, with little contention. *Intruder* is a security application for detecting intrusion in networks. It matches the packets to a set of known signatures. It has relatively short transactions, but the contention varies from moderate to high. *Kmeans* implements the kmeans-algorithm to cluster a set of objects into a set of partitions.The transactions in *Kmeans* are small with small read and write sets. The benchmark has little contention due to less execution time spent in transactions. *Labyrinth* implements a variant of Lee's algorithm, where the threads route paths in a three dimensional grid between a start and an end point. It has almost the whole execution as transactional, with long read and write sets for its transactions, resulting in high levels of contention. *SSCA2* is a scientific application for efficient graph construction, using adjacency arrays and auxiliary arrays. The transactions in SSCA2 are small, with small read and write sets and little contention. *Vacation* is an application from online transaction processing domain that emulates a travel reservation system. The transactions are of medium length, with moderate read and write sets, and most of the execution is transactional. *Yada* is based on Ruppert's algorithm for Delaunay mesh refinement. Yada spends most execution time in transactions, which are relatively longer.

4.1.4 DPS Settings

We experimented with different values of the parameters of DPS. Based on those experiments, we have chosen the success threshold succ_threshold as 0.5, the locality window locality_window to be 4, c_1 to be 3, c_2 to be 2, and c_3 to be 1. We choose the confidence threshold to be 3. We have implemented the lock variable global_lock using the pthread mutex variable.

4.2 Integration with SwissTM

SwissTM is a lock based STM algorithm, highly optimized for throughput. SwissTM has been implemented in C++. Each address in SwissTM has two locks - a read-lock and a write-lock. A writing transaction acquires the write-lock while writing, to prevent other transactions from concurrently writing. A writer acquires a read lock at the time of commit. This essentially provides SwissTM with a hybrid conflict detection scheme, where read-write conflicts are lazily detected and write-write conflicts are eagerly detected. As the read-lock is only acquired at commit time, read-write sharing is allowed. The fact that SwissTM detects read-write conflicts lazily, is believed to be critical to the performance of SwissTM. Our implementation of the DPS scheme in SwissTM follows the description in Algorithm 2 and Algorithm 3. During the start of a transaction, for all addresses in the predicted read set, we check whether the read lock has been acquired, and for all addresses in the predicted write set, we check whether the write lock has been acquired. Thus, like the base SwissTM, DPS-enabled SwissTM allows read-write sharing.

4.3 Analysis of SwissTM results

We now provide experimental results of the different scheduler techniques applied to SwissTM, and compare them against the base SwissTM. We set the preemptive waiting flag in SwissTM to true, as it shows to perform better than busy waiting. We start with a throughput analysis, and then analyze the number of aborts per commit of different scheduling schemes in SwissTM.

4.3.1 Throughput results on STMBench7

We compare the performance in three different workload types. We compare all scheduler variants of SwissTM: Pool-SwissTM, ATS-SwissTM, and DPS-SwissTM.

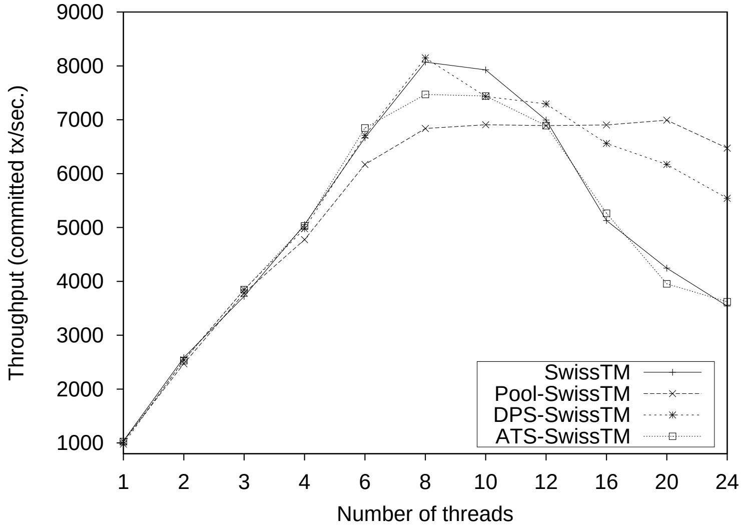


Figure 4.1: Throughput results for SwissTM on STMBench7 under read-dominated workload, with various schedulers.

Read-dominated workloads. In Figure 4.1, we show the throughput results for the read-dominated workloads of STMBench7. The throughput across all schemes has been averaged over 20 executions. The results show that base SwissTM and DPS-SwissTM have comparable throughput in underloaded cases, i.e., up to 8 threads, but DPS-SwissTM gradually outperforms base SwissTM as the system becomes more overloaded. For example, DPS-SwissTM performs 55% better than base SwissTM when 24 threads are spawned. Pool-SwissTM outperforms base SwissTM in overloaded cases gaining a maximum throughput of 80% in 24 threads, but at the same time, Pool-SwissTM suffers a performance penalty of up to 15% compared to the base SwissTM in the underloaded scenario with 8 threads. ATS-SwissTM does not gain as much as DPS-SwissTM or Pool-SwissTM in overloaded cases. ATS-SwissTM shows comparable performance to the base SwissTM in underloaded systems up to 6 threads, but shows a performance degradation between 6 threads and 12 threads, with a maximum throughput loss of around 8% in the case of 8 threads. We note that in general, serialization helps as a TM becomes overloaded.

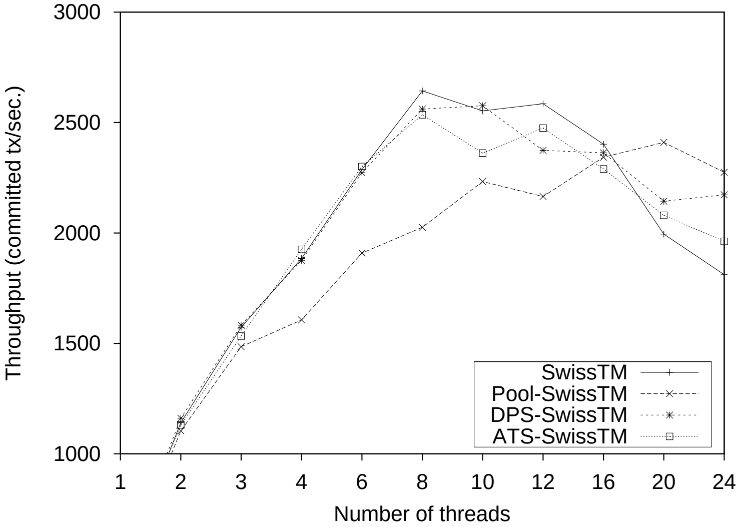


Figure 4.2: Throughput results for SwissTM on STMBench7 under read-write workload, with various schedulers.

Read-write workloads. Figure 4.2 shows the throughput of SwissTM with different schedulers in read-write workload of STMBench7. Here, DPS-SwissTM achieves a maximum throughput gain of 20% for 24 threads. For underloaded scenarios, base SwissTM, ATS-SwissTM, and DPS-SwissTM show comparable performance (within ±5%), while Pool-SwissTM pays a penalty for its coarse serialization. Although, Pool-SwissTM outperforms others when the number of threads exceeds 16, the loss in throughput in less loaded scenarios (thread count less than 12) is as high as 25%, and thus unacceptable. ATS-SwissTM and

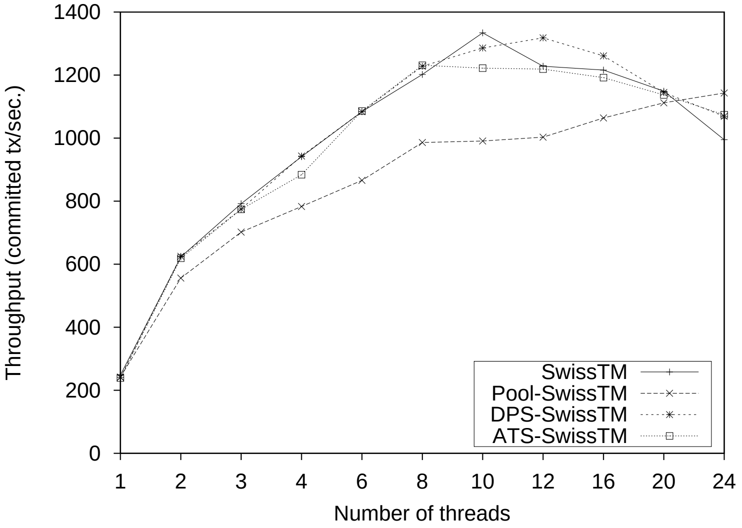


Figure 4.3: Throughput results for SwissTM on STMBench7 under write-dominated workload, with various schedulers.

DPS-SwissTM perform fairly similar, with DPS-SwissTM gaining performance in highly overloaded scenarios. We note that this workload brings forward a distinction in the ATS and Pool schedulers. Although they both serialize threads facing contention, the queueing up in ATS helps it to outperform Pool in underloaded scenarios.

Write-dominated workloads. For write-dominated workloads, as shown in Figure 4.3, DPS-SwissTM achieves a maximum gain of 8% over the base SwissTM in the case of 24 threads. Pool-SwissTM performs poorly in underloaded scenarios (performance loss of 20%), as well as in overloaded scenarios up to 20 threads. But, it outperforms all other schemes in the case of 24 threads. The DPS-SwissTM always exhibits throughput gain of around 5-10% over ATS-SwissTM in overloaded cases. In underloaded cases, DPS-SwissTM and base SwissTM have comparable performance, while ATS suffers a performance loss in the case of 4 threads.

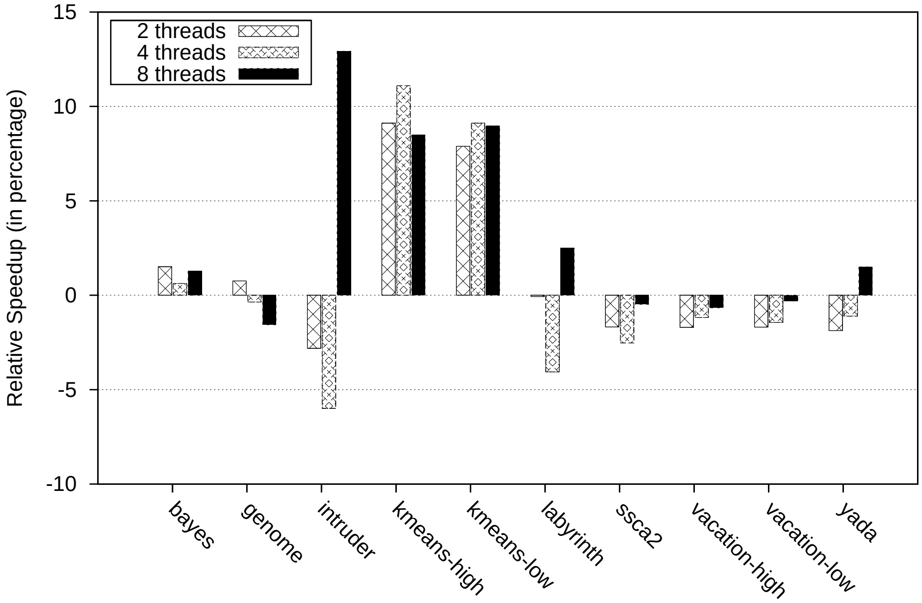
To summarize the results for the three workload types on STMBench7, we observe that

Pool-SwissTM is the best choice in highly overloaded TMs (24 threads). This is obvious, as the high contention in such scenarios makes the coarsest scheduling scheme the most efficient. Pool-SwissTM outperforms ATS-SwissTM as the latter suffers the overhead of queueing transactions in a particular order. DPS-SwissTM does not serialize in every case while base SwissTM never serializes a transaction, which make their respective throughput lower than Pool-SwissTM.

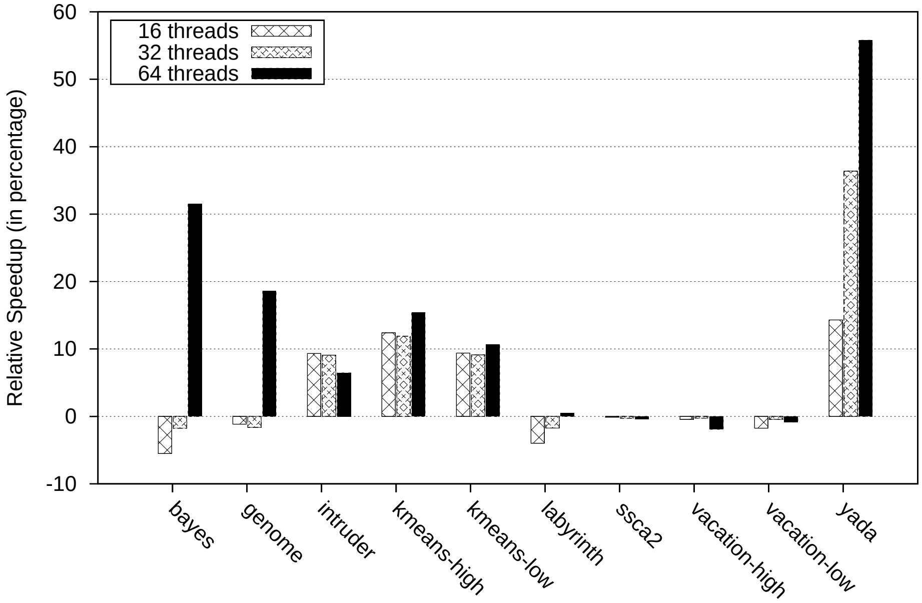
For less overloaded or underloaded cases, DPS-SwissTM consistently performs comparable or better than all other schemes. Although Pool-SwissTM is the best for highly overloaded cases, it suffers performance losses of up to 25% in most of the underloaded scenarios. On the other hand, DPS-SwissTM suffers a rare penalty of 5-8%, while giving significant performance gains up to 55% in some scenarios.

4.3.2 Performance results on STAMP

We now present our analysis of DPS-SwissTM on the workloads in the STAMP benchmark suite. Experimental results are listed in Figure 4.4(a) for underloaded TMs (number of threads at most 8), and Figure 4.4(b) for overloaded TMs (number of threads more than 8). For the case of 1 thread, DPS-SwissTM and base SwissTM differ by less than 1% across all workloads, and hence we do not plot these results. These figures show the relative speed up (in percentage) achieved by DPS-SwissTM over the base SwissTM in the STAMP benchmarks. A positive relative speed up means that DPS-SwissTM runs faster, while a negative relative speed up means that the base SwissTM is faster. For all workloads, we performed 20 executions and averaged them to obtain the results. We get the best results from DPS-SwissTM in the case of *kmeans* benchmark. DPS-SwissTM attains a relative speed up of around 10% in underloaded cases, and up to 15% in overloaded cases. We attribute this performance gain to the small size of the transactions in *kmeans*.



(a) Relative speedup in underloaded scenarios



(b) Relative speedup in overloaded scenarios

Figure 4.4: Relative speedup (in percentage) of DPS-SwissTM over base SwissTM across STAMP workloads

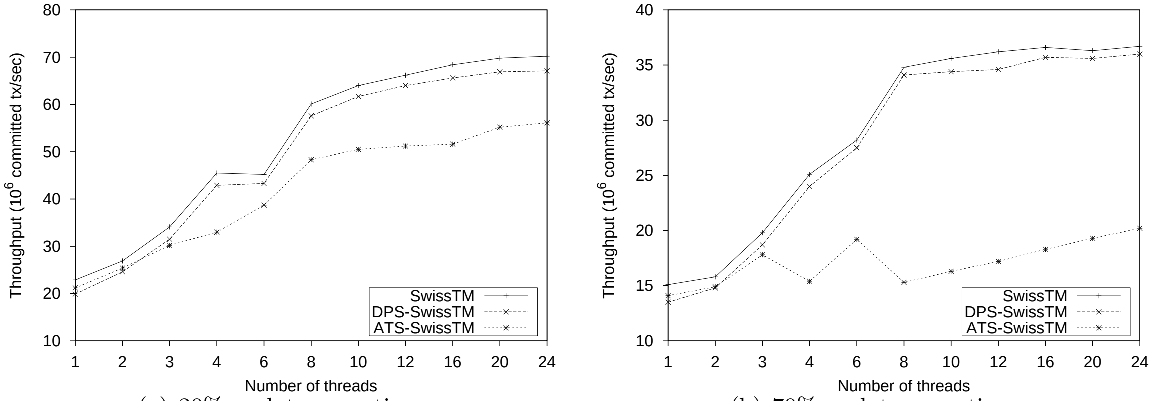
The DPS policy asks transactions whose access set is locked, to serialize: such a policy is naturally inclined towards benefiting small transactions more. This is because the idea of serialization in the cases when the access set is locked, may not be an optimal decision for long transactions. With long transactions, it may happen that even if a transaction's access set is locked when the transaction started, the access set becomes free later by the time the transaction actually accesses the addresses. At the same time, we note that the scheduling policy does introduce some overheads in starting a transaction, which tend to affect smaller transactions more than longer ones. So, if the prediction technique does not help, the benchmarks with smaller transactions shall suffer more due to overheads. This is evident from the performance losses of up to 3% in *ssca2* benchmark. For most other benchmarks, we generally face a performance degradation of up to 3% in underloaded scenarios, and performance improvements in overloaded scenarios.

An interesting case is the *intruder* benchmark, where DPS-SwissTM gains significantly (10-15%) in 8 threads and overloaded scenarios. In the *intruder* benchmark, a high number of transactions dequeue elements from a single queue. Thus, the serializing nature of DPS helps DPS-SwissTM to outperform the base SwissTM.

The maximum performance improvement is obtained in the *yada* benchmark, where we speed up performance by up to 35% in 32 threads, and 55% in 64 threads. This suggests that the prediction and the serialization techniques help reduce contention in the *yada* benchmark.

4.3.3 Throughput results on red-black tree microbenchmark

Figure 4.5 shows the performance of base SwissTM, DPS-SwissTM, and ATS-SwissTM on the red-black tree microbenchmark. Although microbenchmarks are not a representative of the realistic applications, they help evaluate any overheads introduced in our scheme



(a) 20% update operations

(b) 70% update operations

Figure 4.5: Throughput results for SwissTM and DPS-SwissTM on the red-black tree microbenchmark

due to the time needed for prediction or serialization of transactions. We perform our experiments on red-black tree benchmark, under 20% and 70% update operations and integer set range of 16384. The results are similar across the two cases. We observe that the prediction techniques of DPS-SwissTM incur a performance loss of around 13% in the case of 1 thread, and this performance loss gradually decreases as the number of threads increases. For example, with 8 threads, DPS-SwissTM incurs a performance loss of 4% with 20% update operations (Figure 4.5(a)) and a performance loss of 2% with 70% update operations (Figure 4.5(b)). On the other hand, ATS-SwissTM exhibits significantly higher overheads. With 8 threads, ATS-SwissTM performs 20% worse than base SwissTM when there are 20% update operations.

4.3.4 Number of aborts per commit

Apart from the throughput measurements, we also give the performance results in terms of number of aborts per commit (APC). APC provides a measure of the wasted work performed by an STM. A higher value of APC means that the STM aborts very often, and

Table 4.1: Throughput and number of aborts per commit (APC) in different variants of SwissTM on the STMBench7 benchmark. The experiments are performed on an 8-core machine (4 processor dual-core AMD Opteron 8216 2.4 GHz, with 1024 KB cache)

#threads	Base SwissTM		Pool-SwissTM		DPS-SwissTM		ATS-SwissTM	
	Throughput	APC	Throughput	APC	Throughput	APC	Throughput	APC
Read-dominated workloads								
1	1037	0.00	1030	0.00	979	0.00	1024	0.00
2	2583	0.34	2471	0.37	2542	0.37	2528	0.34
3	3722	0.87	3784	0.39	3844	0.61	3845	0.43
4	5054	1.16	4774	0.45	4979	0.97	5028	0.60
6	6665	1.80	6169	0.64	6710	1.46	6847	1.00
8	8070	2.21	6838	0.49	8147	2.47	7469	1.08
10	7926	3.16	6906	0.40	7433	1.86	7441	0.99
12	6994	4.72	6891	0.21	7294	1.82	6892	0.67
16	5128	12.76	6902	0.17	6560	2.43	5263	0.68
20	4244	29.52	6993	0.14	6169	1.71	3954	0.86
24	3541	62.62	6472	0.14	5542	1.60	3620	0.90
Read-write workloads								
1	444	0.00	430	0.00	458	0.00	471	0.00
2	1141	3.13	1104	2.56	1161	3.22	1131	2.54
3	1573	4.72	1485	2.16	1582	4.78	1533	3.43
4	1884	7.13	1606	1.93	1876	7.01	1926	3.77
6	2287	11.52	1909	1.71	2273	10.47	2301	5.59
8	2643	17.13	2026	0.94	2561	13.96	2535	7.18
10	2553	17.55	2233	0.69	2576	12.86	2362	6.07
12	2585	21.50	2165	0.57	2374	8.71	2475	5.70
16	2402	33.20	2343	0.52	2363	6.30	2290	6.78
20	1995	54.04	2410	0.30	2144	5.96	2080	5.71
24	1811	82.40	2275	0.24	2173	2.66	1962	2.86
Write-dominated workloads								
1	248	0.00	242	0.00	241	0.00	239	0.00
2	623	8.76	556	5.96	625	9.07	619	6.02
3	792	13.17	702	5.15	775	13.62	774	8.89
4	941	17.88	783	3.80	943	18.47	884	11.36
6	1084	30.17	866	3.16	1084	26.48	1086	16.80
8	1202	42.95	986	1.73	1228	36.69	1231	18.93
10	1334	40.06	991	1.43	1286	41.96	1222	17.06
12	1228	48.03	1003	1.81	1318	43.58	1219	15.44
16	1216	63.44	1064	1.29	1261	45.49	1192	13.84
20	1149	93.88	1112	0.88	1145	46.29	1137	11.14
24	995	100.45	1143	0.45	1069	35.15	1074	8.33

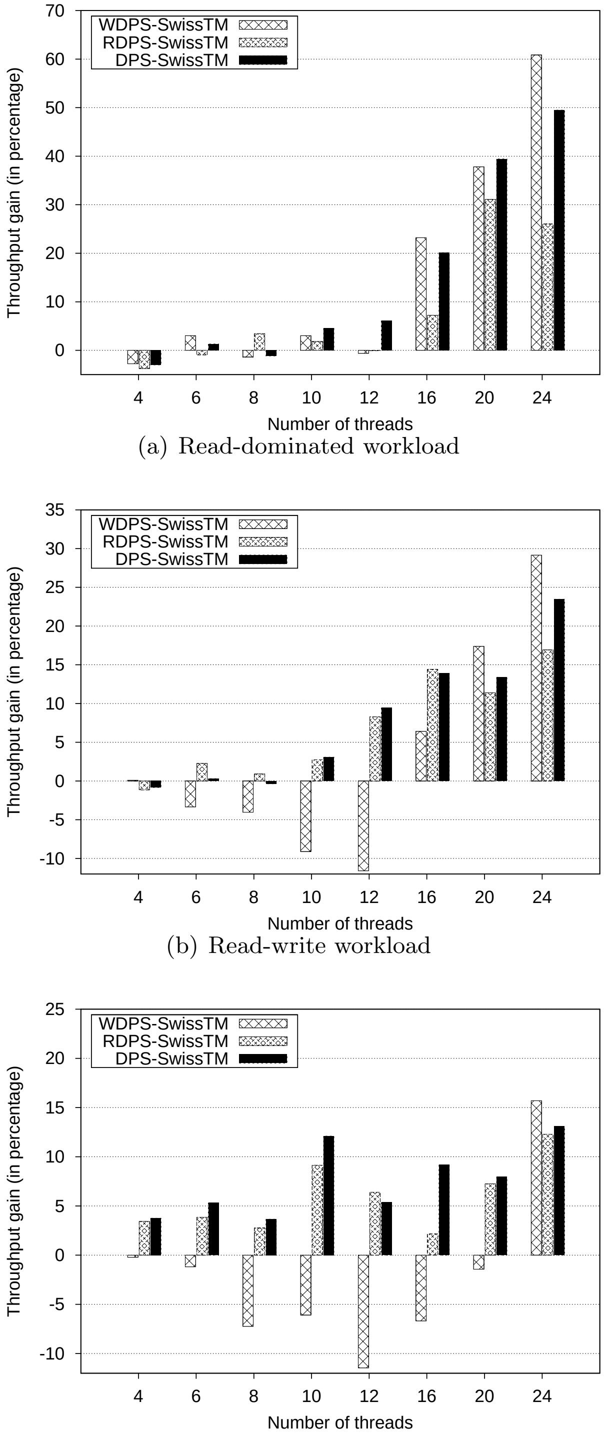
wastes a lot of work. So, according to the APC measure, an STM with a lower value of APC is better.

We analyze the APC of different variants of SwissTM (base SwissTM, Pool-SwissTM, DPS-SwissTM, and ATS-SwissTM) on different STMBench7 workloads. We show the results in Table 4.1. Our results demonstrate how serialization can drastically improve STMs on the measure of APC. We observe that Pool-SwissTM outperforms all other variants of SwissTM by a wide margin. Across all workload types, the APC of Pool-SwissTM does not go above 6, and is around 1 on the average. APC is maximum in write-dominated workloads, and minimum in read-dominated ones. Interestingly, the APC value of Pool-SwissTM is maximum in underloaded scenarios, with the number of threads between 2 to 6 (depending on the workload type). This is due to the fact that overloaded systems face high contention, which forces serialization, which in turn, drastically reduces the number of aborts. On the contrary, the APC of the base SwissTM increases monotonically with the number of threads, in each workload type. This is obvious, as increasing the number of threads increases contention, which increases APC as the base SwissTM does not have a serialization scheme. The APC for the DPS-SwissTM is close to that of the base SwissTM for underloaded systems, in each workload type. But, the APC for DPS-SwissTM does not increase (and sometimes decreases) as the system becomes more overloaded. This is an interesting observation, which basically manifests the heuristic of serialization affinity. In underloaded cases, DPS-SwissTM is designed to behave similar to the base SwissTM, whereas for overloaded cases, DPS-SwissTM behaves like Pool-SwissTM. The APC for ATS-SwissTM is lower as compared to base SwissTM and DPS-SwissTM, but much higher than Pool-SwissTM. In general, the ratio of APC for ATS-SwissTM to the APC for Pool-SwissTM is around 2-3 for underloaded systems, and around 5-10 for overloaded systems. Comparing the APC for ATS-SwissTM and DPS-SwissTM, we observe that DPS-SwissTM faces a higher APC value due to the fact that DPS-SwissTM is more throughput-oriented by design.

Based on our analysis of the results on SwissTM, we conclude that, in general, DPS-SwissTM has higher throughput than all other variants of SwissTM. While it suffers overheads of up to 5-8% in rare cases, it often boosts the performance of base SwissTM, in some cases by up to 55%. So, by the measure of throughput, DPS-SwissTM offers the best choice. Moreover, the coarsest scheduler, Pool-SwissTM has lowest APC among all variants of SwissTM, but Pool-SwissTM suffers performance penalty of up to 25% in many cases. So, if one is ready to sacrifice performance to some extent for the sake of reducing wasted work, then Pool-SwissTM is the best choice.

4.4 Individual Prediction Throughput Analysis

To quantify the benefits of individual prediction schemes (read set prediction and write set prediction) in DPS-SwissTM, we switch-off the two predictions one by one and run the experiments to evaluate the throughput gain over base SwissTM. We refer to the DPS scheme variant which uses write prediction only, as WDPS, and similarly RDPS represents the DPS scheme variant using read prediction only. The results show that no individual prediction scheme (read or write) is strictly better than the other, and the DPS scheme tends to average between the throughput gains achieved by the individual prediction schemes. This can be thought of as follows. Both schemes have their overheads and performance benefits. In a given workload type, one scheme may be boosting performance, while the other one may be causing overheads. The DPS scheme incorporates both schemes and averages out the performance gain. The results of the experiments under the three workload types of STMBench7 are listed in Figures 4.6. WDPS-SwissTM seems to boost performance significantly, by up to 30-60% in highly overloaded cases, while causing performance penalties



(a) Read-dominated workload

(b) Read-write workload

(c) Write-dominated workload

Figure 4.6: Throughput gain (in percentage) over base SwissTM with individual prediction strategies

of up to 12% in some less overloaded cases with 12 threads. In general, WDPS-SwissTM improves performance when the number of threads is at least 16. The graphs show that the write set prediction is crucial for the performance gains of DPS in overloaded TMs on read-dominated workloads. On the other hand, RDPS-SwissTM gives moderate performance gains in both underloaded and overloaded cases, but never faces a performance loss beyond 2% across all workloads. It may be interesting to learn the utility of a given prediction scheme and turn it on or off on the fly.

4.5 Integration and Analysis with TinySTM

TinySTM is a lock-based STM, and has been implemented in C. Every address in TinySTM has a single lock, which is eagerly acquired by a transaction writing to the address. When a lock is acquired by a transaction, other transactions busy-wait for the lock. The reads are invisible in TinySTM. We integrate our DPS scheme in TinySTM version 0.9.5. We evaluate the benefit of scheduling in TinySTM on STMBench7 benchmark.

As we did for SwissTM, we compare the results on three workload types, read-dominated, read-write, and write-dominated. We set long traversals off. We measure throughput for 20 executions for base TinySTM and DPS-TinySTM, and average the results.

Read-dominated workloads. In read-dominated workloads, as shown in Figure 4.7, DPS-TinySTM shows performance comparable to the base TinySTM in underloaded cases (±5%). As the TM becomes overloaded, DPS-TinySTM outperforms base TinySTM (by 180% in 10 threads, and up to 1900% in 24 threads). The steep fall of the performance of TinySTM in overloaded systems can be attributed to its eager lock acquisition policy, where the transactions abort very often due to the high contention caused by the read-write conflicts. Moreover, due to the busy-waiting policy, a transaction waiting for a lock does not yield the processor. DPS-TinySTM mitigates this performance loss to some extent by

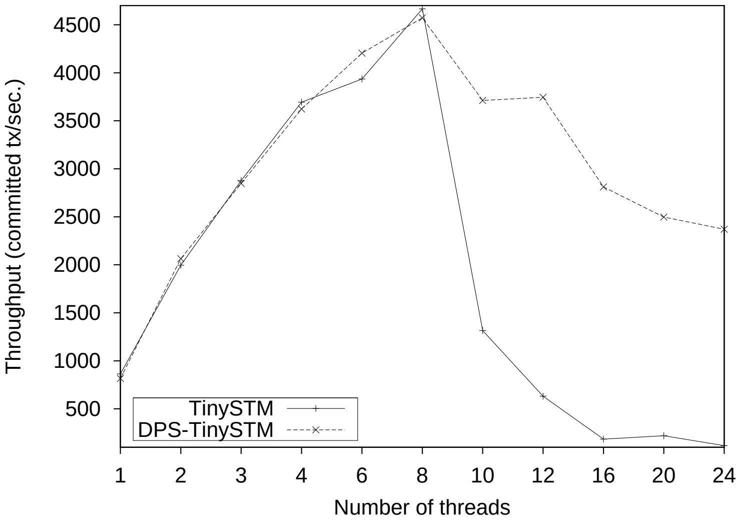


Figure 4.7: Throughput results for TinySTM on STMBench7 under read-dominated workloads

dynamically predicting such conflicts and avoiding them using serialization on the fly.

Read-write workloads. Under read-write workloads (Figure 4.8), we observe a trend similar to that of read-dominated workloads. In underloaded systems (up to 8 threads), DPS-TinySTM and base TinySTM have comparable performance. DPS-TinySTM performs within ±6% of the base TinySTM in such cases. As the system becomes overloaded, DPS-TinySTM maintains its throughput between 1200 and 1500 committed transactions per second. On the other hand, the throughput of base TinySTM drops sharply, yielding a small throughput of less than 100 above 12 threads. This results in a high performance gain of DPS-TinySTM in overloaded cases (around 400% in 10 threads and 2800% in 24 threads).

Write-dominated workloads. The experiments with the write-dominated workloads (Figure 4.9) show that DPS-enabled TinySTM performs strictly better than the base STM, across all thread counts. Below 6 threads, the throughput gain remains within 5%. The

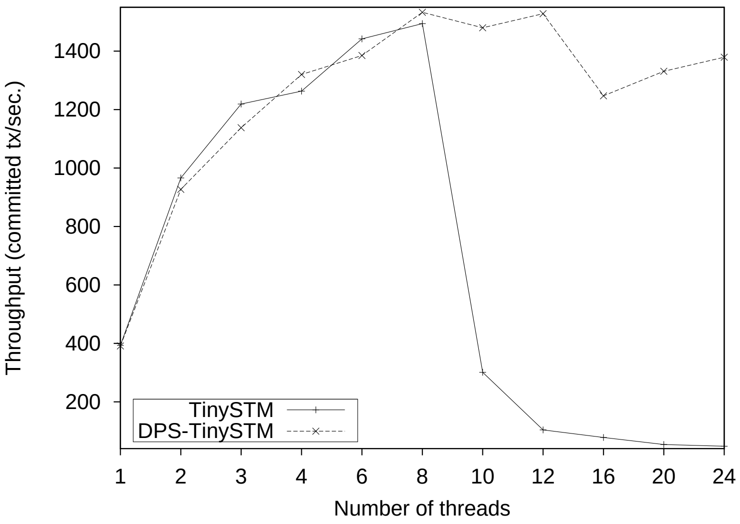


Figure 4.8: Throughput results for TinySTM on STMBench7 under read-write workloads

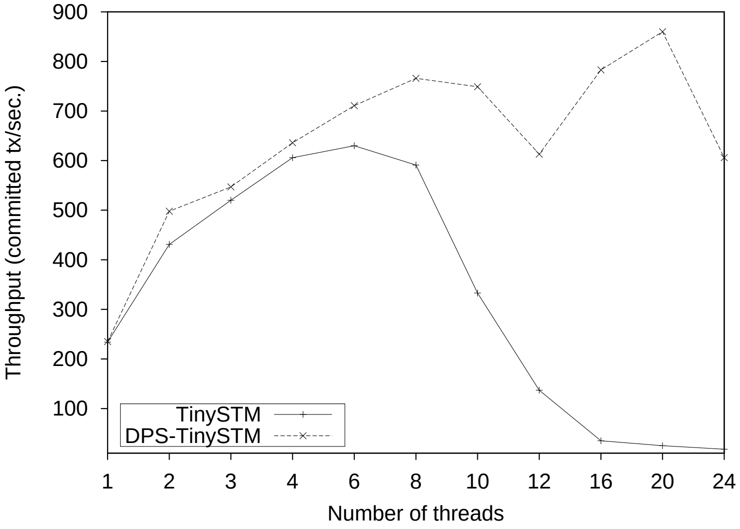


Figure 4.9: Throughput results for TinySTM on STMBench7 under write-dominated workloads

gain rises to around 12% in the case of 6 threads. As the thread count increases further, the throughput gain reaches the value of 3200% in 24 threads.

4.6 HTM integration

The simple design of DPS make DPS a suitable scheduling policy for integration with hardware TMs. DPS uses the variable, success rate, and the data structures in form of Bloom filters, and predicted read and write sets. We give an example of how DPS can be integrated with an HTM. Each processor uses a register to store the current success rate of the transactions running on the processor. Moreover, each processor uses a set of Bloom filters to store the access sets of past few transactions. For most HTMs, the predicted read and write sets can be stored similar to the read and write sets. For example, in LogTM, the read and write sets are stored in virtual memory. Some HTMs use a mark bit in the cache to represent read and write sets. For such cases, we can use a small processor buffer to store the predicted read and write sets. When the success rate is lower than a threshold, the processor checks, using the underlying conflict detection policy (usually cache coherence) whether an address in the predicted read or write set is being written by some processor. In that case, the processor waits until the particular address is free. The processor may get the notification, when a particular address is free, using the cache coherence mechanism.

Chapter 5

Related Work

Contention managers [SS05, GHP05] have been widely studied and used to boost performance of transactional memories in high contention scenarios. However, a TM system consults a contention manager only after a conflict is detected. This results in wasted work of at least one of the transactions.

There has been recent work in the direction of scheduling transactions in a TM. Initial work in this direction has proposed schemes to serialize transactions that face contention or reschedule a transaction on the core of a conflicting transaction. Yoo and Lee proposed adaptive transactional scheduling (ATS) [YL08], which detects threads that face high contention, and serializes them in a queue. The authors demonstrate the performance gain obtained with ATS in RSTM [MSH$^+$06] and LogTM [MBM$^+$06a] for overloaded systems. Our Pool scheduler works in a similar manner. The Pool scheduler forms a thread pool rather than a queue, and thus saves overheads of queue insertion. In this thesis we argued that Pool and ATS are coarse scheduling policies, and showed their performance results. Transactions may abort due to several factors, and serializing aborted transactions in every case may degrade performance. For example, a transaction may abort due to a failed validation, which occurs when another transaction with a conflicting write set commits. In this case, the aborting transaction should retry immediately rather than serializing. Ansari et al. proposed Steal-on-abort [ALK$^+$09], which avoids wasted work by runtime ordering of transactions on observing conflicts. An aborting transaction (say T) is *stolen* by its conflicting transaction (say T'). The transaction T is released after T' commits. The authors propose various strategies for the release of stolen transactions. Rossbach et al. proposed TxLinux [RHP$^+$07] as a variant of the Linux operating system, which provides HTM functionality in the Linux OS. The OS scheduler in TxLinux is transaction aware, and makes scheduling decisions such that transactions waste minimal work. All these TM schedulers use serialization to avoid conflicts, but none of them predicts a conflict before the conflict actually happens.

Another proposal, CAR-STM [DHS08] maintains a transaction scheduling queue per core, and introduces a serializing contention management for resolving conflicts by enqueuing conflicting transactions to the appropriate queue. The serializing contention manager does ensure that two transactions never conflict more than once. Still, as for any other contention manager, the serializing contention manager is reactive, and does not help to prevent conflicts. CAR-STM proposes yet another feature, called *proactive collision avoidance*, which uses a probabilistic measure of conflict between transactions, and tries to avoid executing those transactions concurrently, which are more likely to conflict. However, CAR-STM expects the application to provide the probability measure. This is generally not possible for an application to determine apriori. For this reason, most of the evaluation using CAR-STM has not used the proactive collision avoidance feature. Our prediction based scheduler essentially discovers the probability of conflict on the fly, and uses this probability to postpone the execution of transactions which are likely to conflict with currently executing transactions.

Indeed, prediction in TM has been studied for different purposes. For example, Waliullah and Stenstrom [WS08] use a prediction technique to insert a checkpoint before the first predicted conflicting access executes. This saves execution time up to the occurrence of first conflicting access, from the start of the atomic block. As this scheme preserves wasted work before the checkpoint, it gains performance due to reduced aborts.

Chapter 6

Conclusion

We introduced novel access set prediction techniques for transactional memories. We developed and used the idea of temporal locality for predicting read accesses of transactions. We used the write set information of aborted transactions for predicting write accesses of restarting transactions. We observed that existing serialization based TM schedulers perform better when the number of threads is high. Based on this observation, we developed a heuristic of serialization affinity, which encourages serialization of transactions when the number of threads is high. Using the prediction techniques and the serialization affinity, we built our dynamic prediction based scheduler (DPS), which dynamically serializes transactions based on the current contention and the likelihood of conflict with currently executing transactions.

We illustrate the performance improvement obtained with DPS by integrating DPS with different STMs, SwissTM and TinySTM. Unlike existing TM schedulers, DPS does not serialize transactions in every scenario, which allows DPS to perform comparable to base STMs when the number of threads is low. We motivate that the number of threads can exceed the number of cores in practical scenarios, due to multiple tasks executing on a limited number of cores, or to exploit parallel nesting. Our experiments on realistic benchmarks, like STMBench7, show that DPS boosts the performance in such cases by up to 55% in SwissTM and up to 3000% in TinySTM. Moreover, we observe that DPS incurs negligible overheads of prediction and serialization. In cases where contention is low, and the number of threads is less than the number of cores, DPS-enabled STMs perform 5-8% worse than the base STMs.

In general, DPS can be integrated with any STM or HTM. The prediction techniques are a step towards obtaining a clairvoyant TM, where the TM, at the start of a transaction, knows which addresses shall be accessed by the transaction.

Future work in this direction shall investigate how an OS scheduler can be used to refine the prediction scheme. One possibility is that the OS informs the TM how many cores are available to the application. The TM can then further use this information to better adapt the serialization of transactions. Moreover, our evaluation of individual prediction schemes (read set prediction and write set prediction) showed that under a given scenario, one scheme may benefit performance, while the other scheme incurs overheads. We believe that adapting DPS in a way that it can turn a specific scheme on or off, shall further boost the performance of DPS-enabled TMs.

Bibliography

[AAK$^+$05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *International Symposium on High-Performance Computer Architecture*, pages 316–327. IEEE Computer Society, 2005.

[ABCdO96] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the www. In *DIS '96: Proceedings of the fourth international conference on on Parallel and distributed information systems*, pages 92–107. IEEE Computer Society, 1996.

[AFS08] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 163–174. ACM, 2008.

[AKH03] J. H. Anderson, Y. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, pages 75–110, 2003.

[ALK$^+$09] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *International*

Conference on High Performance and Embedded Architectures and Compilers. Springer, January 2009.

[BGH$^+$08] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *International Symposium on Computer Architecture*. IEEE Computer Society, Jun 2008.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[Den68] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[DFL$^+$06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346. ACM, 2006.

[DGK08] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. Technical Report LPD-REPORT-2008-013, Ecole Polytechnique Federale de Lausanne, 2008.

[DHS08] Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 125–134. ACM, 2008.

[DS72] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, 1972.

[DSS06] David Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *International Symposium on DIStributed Computing*, pages 194–208. Springer, 2006.

[EGLT76] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, pages 624–633, 1976.

[GHP05] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *International Symposium on DIStributed Computing*, pages 303–323. Springer, 2005.

[GHP06] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Towards a theory of transactional contention managers. In *Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 316–317. ACM, 2006.

[GK08] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2008.

[Her91] Maurice Herlihy. Wait-free synchronization. *IEEE Transactions on Parallel and Distributed Systems*, pages 124–149, 1991.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. ACM, 2003.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. *International Conference on Distributed Computing System*, pages 522–529, 2003.

[HLMI03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 92–101. ACM, 2003.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pages 289–300. IEEE Computer Society, 1993.

[HMJH05] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM, 2005.

[HWC$^+$04] Lance Hammond, Vicky Wong, Michael K. Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *International Symposium on Computer Architecture*, pages 102–113. IEEE Computer Society, 2004.

[KBG97] Alain Kägi, Doug Burger, and James R. Goodman. Efficient synchronization: Let them eat qolb. In *International Symposium on Computer Architecture*, pages 170–180, 1997.

[KCH$^+$06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *ACM SIGPLAN Sym-*

posium on Principles and Practice of Parallel Programming, pages 209–220. ACM, 2006.

[LR07] J. R. Larus and R. Rajwar. *Transactional Memory.* Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[MBM+06a] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *International Symposium on High-Performance Computer Architecture*, pages 254–265. IEEE Computer Society, Feb 2006.

[MBM+06b] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370. ACM, New York, NY, USA, Oct 2006.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[MIS05] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *International Symposium on DIStributed Computing*, pages 354–368. Springer, 2005.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 267–275. ACM, 1996.

[MSH$^+$06] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006.

[MTC$^+$07] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *International Symposium on Computer Architecture*, pages 69–80. ACM, 2007.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, pages 631–653, 1979.

[RFF07] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, Jun 2007.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 294–305. ACM/IEEE, 2001.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17. ACM, Oct 2002.

[RHL05] Ravi Rajwar, Maurice Herlihy, and Konrad K. Lai. Virtualizing transactional memory. In *International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, 2005.

[RHP$^+$07] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *ACM Symposium on Operating System Principles*, pages 87–102. ACM, 2007.

[SATH$^+$06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197. ACM, Mar 2006.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[SMIS06] Michael F. Spear, Virendra J. Marathe, William N. Scherer Iii, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *International Symposium on DIStributed Computing*. Springer, 2006.

[SS05] William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 240–248. ACM, 2005.

[SSH$^+$07] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *International Symposium on Computer Architecture*, pages 104–115, New York, NY, USA, 2007. ACM.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[WS08] M. M. Waliullah and Per Stenström. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–11. IEEE Computer Society, 2008.

[YBM$^+$07] Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Feb 2007.

[YL08] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 169–178. ACM, 2008.