Preventing versus Curing: Avoiding Conflicts in Transactional Memories

Aleksandar Dragojević   Rachid Guerraoui
Anmol V. Singh   Vasu Singh

EPFL

Abstract

Transactional memories are typically speculative and rely on contention managers to *cure* conflicts. This paper explores a complementary approach that *prevents* conflicts by *scheduling* transactions according to predictions on their access sets.

We first explore the theoretical boundaries of this approach and prove that (1) a TM scheduler with an accurate prediction can be 2-competitive with an optimal TM scheduler, but (2) even a slight inaccuracy in prediction makes the competitive ratio of the TM scheduler of the order of the number of transactions.

We then show that, in practice, there is room for a pragmatic approach with good average case performance. We present *Shrink*, a scheduler that (a) bases its prediction on the access patterns of the past transactions from the same threads, and (b) uses a novel heuristic, which we call *serialization affinity*, to schedule transactions with a probability proportional to the current amount of contention. Shrink obtains roughly 70% accurate predictions on STMBench7 and STAMP. For SwissTM, Shrink improves the performance by up to 55% on STMBench7, and up to 120% on STAMP. For TinySTM, Shrink improves the performance by up to 30 times on STMBench7 and 100 times on STAMP.

This paper is submitted as a regular paper. Except Rachid Guerraoui, all authors are full-time students. Anmol V. Singh could kindly be considered for the best student paper award.

Contact author: Anmol V. Singh
Address: EPFL IC, Station 14, 1015 Lausanne, Switzerland.
Email address: anmol.tomar@epfl.ch

1 Introduction

Transactional memory (TM) is an appealing abstraction 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. Transactions in a TM execute speculatively: they typically commit if they do not encounter any conflicts and abort otherwise. Generally, due to the speculative execution, TMs exhibit better performance than lock based programming in low contention scenarios [**?**, 11]. But, in high contention, TMs can suffer from repetitive aborts and result in poor performance [20].

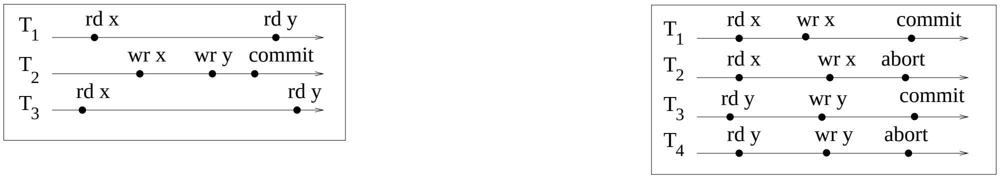
Various strategies have been developed to boost the performance of TMs in high contention scenarios. These strategies are ususally encapsulated in a dedicated module called a *contention manager* (CM) [11, 17], which resolves a conflict between two transactions, sometimes based on the past history of the transactions. The practical impact and the theoretical significance of CMs have been widely explored [9, 10, 17]. *Polite*, *Karma*, and *Greedy* are examples of CMs used to boost TM performance. On the theoretical front, it has been proved that Greedy has a competitive ratio of $O(s)$ with respect to an optimal offline scheduler [4], where s is the number of objects accessed in the TM. Another recent proposal is *Serializer* [6], a CM which resolves a conflict by removing a conflicting transaction T from the core where it was running, and scheduling T on the core of the other transaction involved in the conflict. Serializer ensures that two transactions never conflict more than once. Although CMs boost the performance of a TM, they play their role only after conflicts have been detected. CMs do not avoid wasted work performed by threads which are doomed to abort, neither do they speculate conflicts and yield a processor to another non-conflicting thread. Generally, the performance of TMs (with or without contention managers) does not scale well in overloaded cases[1] [6, 20].

TM schedulers [6, 20] offer an appealing complementary approach to boost the performance of TMs in overloaded scenarios. Basically, a TM scheduler is a software component encapsulating a policy that decides when a particular transaction executes. So far, TM schedulers that have been proposed in the literature relied on coarse measures to activate scheduling. These might not always be beneficial. To illustrate this, consider a simple serialization strategy applied to the scenario of Figure 1(a). Both transactions T_1 and T_3 read variable x. Transaction T_2 writes variables x and y, after the reads of x by T_1 and T_3. Then transaction T_2 commits. Transactions T_1 and T_3 each read y next, and hence are bound to abort due to an inconsistency in the values read. Serializing T_1 and T_3, which are not conflicting, hinders parallelism. As another example, consider the ATS strategy[2] [20] applied to the example of Figure 1(b). Transactions T_1 and T_2 conflict, as they both write x. Only one of them can commit. Assume that the TM aborts T_2 and commits T_1. Similarly, the TM may abort T_4 and allow T_3 to commit, as they conflict on variable y. ATS would serialize T_2 and T_4, though the accesses of T_2 and T_4 are completely independent and they could run in parallel after restarting. In short, the decision to serialize a thread is justified only when the reason to abort is likely to repeat. If the cause of the conflict ceases to exist in the future, the thread can very well exploit more parallelism than in the case of serial execution. It is intriguing to investigate *prediction* techniques that help distinguish such scenarios and *dynamically* serialize transactions.

More generally, the motivation of this paper is to explore the power of prediction-based TM schedulers. We present theoretical bounds for traditional serialization techniques: namely, we

[1] We define a system as *overloaded* if the number of threads exceeds the number of cores. Further, we define a system which is not overloaded as *underloaded*.

[2] ATS measures the contention intensity of a thread, and when this increases beyond a threshold, ATS forces the serialization of the thread.



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

(b) Abort due to a read-write conflict

Figure 1: Examples where coarse serialization hinders parallelism

prove that Serializer and ATS are $O(n)$-competitive with the optimal offline scheduler, where n is the number of transactions. Then, we study the benefit of prediction from a theoretical perspective. As defined by Motwani et al. [14], an *online clairvoyant TM scheduler*[3] is a scheduler which has complete information about already released transactions. We present an online clairvoyant TM scheduler, *Restart*, and show that *Restart* is 2-competitive with respect to the optimal offline scheduler, which has complete information about all transactions. *Restart* is of independent interest as it closes the open problem of finding the optimal competitive ratio for an online clairvoyant scheduler [14]. The previous best scheduling algorithm [14] known for the online clairvoyant scheduling problem was 3-competitive, and Motwani et al. [14] have shown that an optimal online clairvoyant scheduler cannot have a competitive ratio less than 2.

At first glance, identifying an online clairvoyant scheduler that is 2-competitive opens a new way to improve TM performance. But, this has to be taken with a big grain of salt. Indeed, we show that the performance in the worst case is very sensitive to the accuracy of the prediction. Namely, we prove that an inaccurate prediction by the online clairvoyant TM scheduler, *Restart*, makes the competitive ratio $O(n)$.

Nevertheless, we show that scheduling transactions based on dynamic prediction can provide excellent average case performance. We introduce a new scheduler, called *Shrink*, which predicts the future accesses of a thread based on the past accesses, and dynamically serializes transactions based on the prediction to prevent conflicts. Shrink is based on two ideas: (1) locality of reference and (2) serialization affinity.

1. We use locality in two forms. To predict *transactional read sets*, we use the notion of *temporal locality* [2, 18] which, in the context of TMs, stipulates that addresses[4] which are frequently accessed in the past few transactions of a thread are more likely to be accessed in future transactions of that thread. Shrink maintains the read set of past few committed transactions of each thread in a set of Bloom filters. Then, Shrink checks membership of an address in these Bloom filters and uses a confidence measure to predict whether the address shall be read in future transactions. To predict *transactional write sets*, Shrink uses the locality across repeated transactions. We empirically observe that locality gives an average accuracy of around 70% in predicting the read and write accesses of a thread in the biggest benchmarks we know of, STMBench7 [8] and STAMP [5]. Shrink uses the predicted access sets, in conjunction with the information of the currently executing transactions, to prevent conflicts. Specifically, Shrink checks whether any address in the predicted read and write sets of the starting transaction is being written by any other currently executing transaction. If this is the case, Shrink serializes the starting transaction. Otherwise, the transaction executes normally.

[3] This was referred to as a *non-clairvoyant* scheduler by Motwani et al.

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

2. To be competitive in low contention scenarios and underloaded cases, Shrink serializes threads only if contention is high. To detect when a thread faces high contention, Shrink maintains a parameter, *success rate*, for every thread, and activates the prediction and serialization techniques for a thread only if its success rate falls below a certain threshold. We observe that serializing a transaction with a probability proportional to the contention in the TM provides better performance. This heuristic, which we call *serialization affinity*, is based on our observation that serializing a transaction is more helpful when a large number of threads access similar addresses and compete for a small number of cores. This is not surprising, as the lack of proper scheduling causes many conflicts in these cases.

Shrink can be integrated with any TM that uses visible writes[5] Most of the TMs we know of [**?**, 7, 11, 15] use visible writes. To evaluate its performance, we integrated Shrink with two state-of-the-art STMs, SwissTM [7] and TinySTM [15]. Besides their availability, these STMs are among the most performant according to latest published results. Shrink proves successful in increasing the throughput of these STMs in overloaded cases, while not degrading performance in underloaded cases. For SwissTM, Shrink obtains performance gains of up to 55% on STMBench7, and up to 120% on STAMP in overloaded cases. For TinySTM, Shrink increases the performance by up to 30 times on STMBench7, and up to 100 times on STAMP. Generally, Shrink has no noticeable overhead with STMBench7 and STAMP in underloaded cases. To identify the overhead in such cases, we considered the red-black tree microbenchmark. We observe that in SwissTM, Shrink incurs a performance drop of about 13% with 1 thread, and about 3% with 24 threads. In TinySTM, we get a similar performance drop with 1 thread, but Shrink drastically outperforms base TinySTM with more than 8 threads.

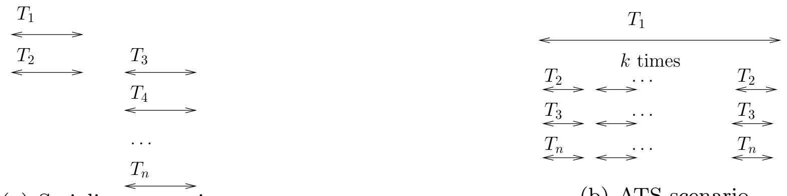
The paper is organized as follows. In Section 2, we investigate prediction in TM schedulers from a theoretical perspective. Section 3 describes our dynamic prediction based TM scheduler, Shrink. In Section 4, we present experimental results obtained with Shrink on different benchmarks with different STMs. Section 5 discusses the related work and Section 6 concludes the paper. Due to space limitations, we postpone some of the experimental results in the appendix.

2 Theoretical Power and Limitations of Prediction

We study the theoretical bounds of different TM schedulers. We first show that traditional serialization techniques, like Serializer and ATS are $O(n)$-competitive, where n is the number of transactions. We show that with an online prediction of the information of transactions, we can get a 2-competitive scheduler. But, with a slight inaccuracy in prediction, the scheduler becomes $O(n)$-competitive.

Framework. We consider a model similar to that of the non-clairvoyant scheduling problem with dependent jobs pioneered by Motwani et al. [14]. Let $T_1 \dots T_n$ be a set of n transactions (called jobs). Let the release time of transaction T_i be R_i, and the execution time of transaction T_i be E_i. Let $R_m = \max_i R_i$ be the latest time when a transaction is released. Let $E_m = \max_i E_i$ be the execution time of the longest transaction. Motwani et al. model the *general distributed scheduling problem* as a graph where the vertices represent transactions, and two vertices are adjacent if the corresponding transactions are in conflict. The processing environment has an infinite number of processors, and a centralized scheduler assigns transactions to processors. A transaction may be preempted and restarted from the point of preemption. Also, a transaction may be aborted and

[5] A TM uses *visible writes* if all threads know whenever a particular thread writes to an address.



(a) Serializer scenario

(b) ATS scenario

Figure 2: Lower bound cases

restarted from the beginning[6]. A preemption and an abort require zero time. The constraint on scheduling is that two conflicting transactions cannot be executed simultaneously, that is, either (i) one transaction preempts if a conflicting transactions starts, or (ii) if two conflicting transactions have executed in parallel, then one of them aborts. We consider the makespan of all transactions as the performance measure. An *offline optimal scheduler* has complete information (conflict relations, execution times, and release times) of all transactions, including those which will appear in the future. Given OPT, the makespan of the offline optimal scheduler, we obviously have $OPT \geq R_m$ and $OPT \geq E_m$.

Bounds on existing CM and TM schedulers. Guerraoui et al. [10] showed that Greedy has a competitive ratio of $O(s^2)$, where s is the number of addresses shared by transactions. Attiya et al. [4] improved the result and showed that Greedy has a competitive ratio of $O(s)$. Further, they showed that the competitive ratio of Greedy is lower bounded by $O(s)$. On similar lines, we prove how Serializer and ATS are $O(n)$-competitive.
Serializer [6]: A transaction is executed as soon as it is available. Upon a conflict between two transactions T_1 and T_2, one of the transactions is scheduled after another.
ATS [20]: A transaction is executed as soon as it is available. ATS maintains a sequence Q of transactions. The transactions in Q are scheduled one after another. When a transaction T aborts k times (where k is a constant) and T is not in Q, then T is added to Q.

Theorem 1. Serializer and ATS are both $O(n)$-competitive.

Proof. (i) *Upper bound for Serializer.* We observe that for a set of n transactions, Serializer can face at most $n-1$ aborts. Then, the makespan of Serializer is upper bounded by $R_m + 2 \cdot n \cdot E_m$. As $OPT \geq R_m$ and $OPT \geq E_m$, we obtain that Serializer is $O(n)$-competitive.
Lower bound for Serializer. Consider a set $\{T_1 \ldots T_n\}$ of transactions as shown in Figure 2(a). All transactions have execution time 1. Let T_1 and T_2 be released at time 0, and all other transactions be released at time 1. Transaction T_1 runs in parallel with transaction T_2. As they conflict, Serializer schedules T_2 after T_1. Then, the TM attempts to run transactions $\{T_3 \ldots T_n\}$ in parallel with T_2. Let all transaction $\{T_3 \ldots T_n\}$ conflict with T_2, and no pair of transactions in $\{T_3 \ldots T_n\}$ conflict. Serializer schedules all transactions T_3 to T_n after T_2. Thus, Serializer has a makespan of n. An optimal offline scheduler could execute T_2 first, and then all remaining transactions at the same time, giving $OPT = 2$. In this instance, Serializer has a makespan of $O(n) \cdot OPT$.

(ii) *Upper bound for ATS.* We observe that after time R_m, any $(k+1)^{th}$ abort of a transaction T overlaps with a commit of a transaction T' or with the m^{th} abort of T' such that $m \leq k$. Thus, the total makespan of ATS is at most $R_m + k \cdot n \cdot E_m + n \cdot E_m$. As $OPT \geq R_m$ and $OPT \geq E_m$

[6] Our model differs from the job scheduling model of Motwani et al. in the respect that the latter does not allow aborts

and k is a constant, ATS is $O(n)$-competitive.
Lower bound for ATS. Consider a set $\{T_1 \dots T_n\}$ of transactions as shown in Figure 2(b). Let all transactions be released at time 0. Let $E_1 = k$ and for $i \geq 2$, let $E_i = 1$. All transactions $T_2 \dots T_n$ conflict with T_1, and no pair of transactions in $\{T_2 \dots T_n\}$ conflict. As $T_2 \dots T_n$ face k aborts, all of them are added to Q. Thus, ATS has a makespan of $k+n-1$, whereas the optimal offline scheduler would require $k+1$ time. As k is a constant, ATS requires $O(n) \cdot OPT$ time in this case.

Online clairvoyant scheduler. We define an *online clairvoyant scheduler* as a scheduler which has the complete information (execution time, release time, and conflict relation) about the transactions which have already been released. We now introduce an online clairvoyant scheduler, which we call *Restart*, and we prove that it has a competitive ratio of 2. *Restart* is a slight variation of the Greedy scheduler described by Motwani et al. [14].
Restart: Let Γ be the set of transactions executing at time t, and let Γ' be the set of released transactions unfinished at time t. When a new transaction T is released at time t, *Restart* aborts the set Γ of currently executing transactions, and executes the transactions $\Gamma' \cup \{T\}$ according to the optimal schedule.

Theorem 2. *Restart is 2-competitive.*

Proof. As *Restart* aborts and restarts the remaining transactions according to the optimal schedule at the release of every job, the makespan of *Restart* is at most $R_m + OPT$. As $OPT \geq R_m$, the competitive ratio of *Restart* is 2.

Note that our result of 2-competitiveness holds even for the original problem described by Motwani et al. [14] where transactions cannot abort, but are allowed to preempt and continue from the point of preemption. In that model, whenever a new job is released (say, at time t), the unfinished transactions at time t shall be scheduled according to the new optimal schedule. *Restart* indeed solves an open problem described by Motwani et al. [14]. The authors established that no deterministic online clairvoyant[7] scheduler is c-competitive for any constant $c < 2$. The authors gave a scheduler, Greedy, which is 3-competitive, and left the problem of determining the optimal competitive ratio open. As the competitive ratio of *Restart* matches the lower bound, we know that no online clairvoyant scheduler can have a better performance ratio than *Restart*. That is, *Restart* gives the optimal competitive ratio.

So, online clairvoyance can make a TM scheduler 2-competitive with the optimal offline scheduler. Nevertheless, the competitive ratio of 2 relies on an optimal scheduling with an accurate prediction of the access sets. Consider a scheduler *Inaccurate* which works like *Restart*, but has an inaccurate prediction of the conflict relation. Also, assume that after time R_m, *Inaccurate* satisfies the *pending commit* property [4], which states that of all transactions running at a given time, atleast one of the transactions commits. We show that *Inaccurate* is $O(n)$-competitive.

Theorem 3. *Inaccurate is $O(n)$-competitive.*

Proof. *Upper bound. Inaccurate* takes time at most $R_m + n \cdot E_m$ time units (due to the fact that *Inaccurate* satisfies the pending commit property after R_m). Thus, we get that *Inaccurate* is $O(n)$ competitive.
Lower bound. Consider a set $\{T_1 \dots T_n\}$ of transactions, which are all released at time 0, and have execution time 1. The transaction T_i accesses only resource R_i for all $i \leq n$. Clearly, the optimal offline scheduler requires 1 time unit. On the other hand, consider an inaccurate prediction, where

[7] referred to as non-clairvoyant by Motwani et al.

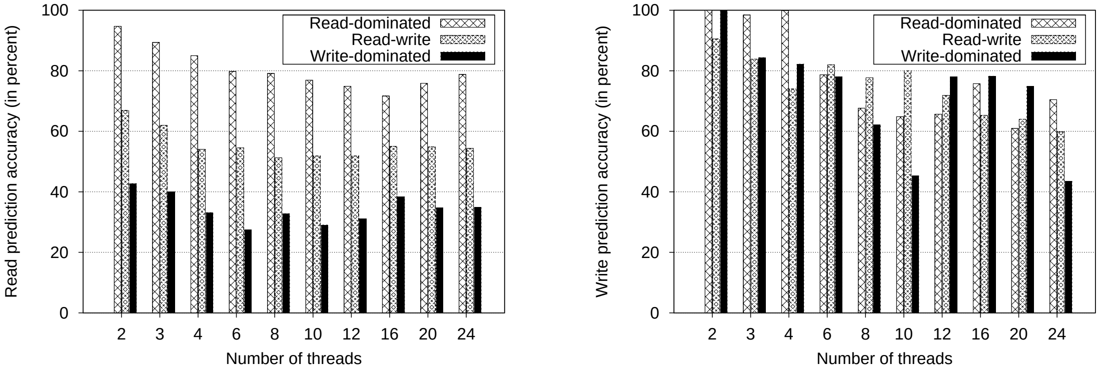
Inaccurate believes that the transaction T_i accesses resources R_i and R_1 for all i. As *Inaccurate* schedules released transactions like an optimal scheduler, *Inaccurate* requires time n.

In retrospect. Although the worst case of *Inaccurate* is $O(n)$, the competitiveness of *Restart* opens hope that prediction in TMs can provide performance benefits in the average case. The first question is how to predict access sets in TMs. Static analysis or compiler assisted methods have limited use in TMs because the accesses are made on dynamic data structures, which change over time. We propose a TM scheduler, *Shrink*, which uses novel techniques to predict the access sets of future transactions. Further, Shrink dynamically serializes transactions based on the predicted access sets and the information of the currently executing transactions.

3 The Shrink

The prediction techniques in Shrink are based on locality of reference across transactions. We describe these techniques, followed by how Shrink uses these predictions. Then, we present the serialization affinity heuristic in Shrink.

Read set prediction. Our empirical observations on several TM benchmarks, like STMBench7 and STAMP, show that multiple consecutive committed transactions of a thread access similar addresses. We refer to this phenomenon as the temporal locality of access sets in threads. We believe that temporal locality in TM is a result of fixed traversals of data structures, 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. Thus, based on the read accesses of the last few transactions, we can predict the read accesses of future transactions.



(a) Read set prediction on STMBench7

(b) Write set prediction on STMBench7

Figure 3: Accuracy of access set predictions by SwissTM

Write set prediction. Temporal locality works with significant accuracy for predicting the read sets, but does not help predict the write sets. This is due to the fact that transactions typically 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 adopt a different prediction strategy for

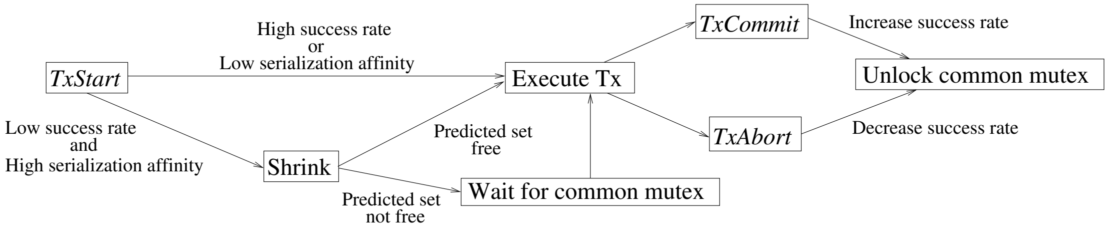


Figure 4: The Shrink flowchart

the write set: we use the fact that when a transaction repeats, its write set mimics the write set of the immediately previous aborted transaction. As the underlying benchmark is unlikely to undergo drastic changes before the aborting transaction restarts execution, the writes of an aborting transaction form a good prediction for the write accesses of the next restarting transaction. 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.

We validate the idea of locality by evaluating the accuracy of our predicted read and write sets. We run our experiments on STMBench7 [8] and STAMP [5] with SwissTM [7], and present our results in Figure 3. For STMBench7, 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. For STAMP, the read prediction is, on an average, 70% accurate. The write set predictions are fairly high across STMBench7 and STAMP benchmarks.

Shrink uses the above prediction techniques to predict the transactional read and write sets. For a starting transaction, Shrink checks whether some address in the transaction's predicted read or write set is being written by some other thread. In that case, Shrink may decide to serialize the starting transaction.

Serialization affinity. So far, TM schedulers serialized transactions even in low contention scenarios [6] and underloaded cases [20]. To understand the performance tradeoff associated with serialization, we built a simple TM scheduler that serializes all threads that face contention. We call this the Pool scheduler. Our experiments show that serialization helps when the number of threads facing contention is high. This points us to a heuristic called *serialization affinity*, which stipulates that the probability of serializing a transaction should be proportional to the amount of contention.

Algorithm. The flowchart for Shrink is shown in Figure 4. Shrink maintains for every thread, the *success rate* as a measure of the ratio of the number of committing to the number of aborting transactions of the thread. The success rate of a thread is modified on commit and abort of a transaction. When the success rate of a thread falls below a certain threshold, the serialization policy of Shrink gets activated.

The modifications needed to integrate Shrink into an STM are shown in Algorithm 1. Shrink uses a set of Bloom filters per thread to represent the thread's read set of last few transactions in

Algorithm 1 *The Shrink Algorithm*

Variables: global_lock is a lock variable shared between threads. success, succ_threshold and confidence_threshold are constants, succ_rate is an integer variable per thread. read_pred and write_pred are addresses, pred_read_set and pred_write_set are the predicted read and write sets per thread, wait_count and confidence are integers (initially 0), bf_i is the bloom filter of the last i^{th} transaction, c_i is a constant representing the confidence value of the last i^{th} transaction, locality_window is the size of the set of bloom filters per thread, and *addr* is an address.

```
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
            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
    if last transaction was committed then remove all addresses from pred_read_set
    remove all addresses from pred_write_set
On transactional read of $addr$
    if $(addr \notin \mathsf{bf}_0)$ then
        add $addr$ to $\mathsf{bf}_0$
        for $(i = 1; i < \mathsf{locality\_window}; i := i + 1)$ do
            if $(addr \in \mathsf{bf}_i)$ then $\mathsf{confidence} = \mathsf{confidence} + c_i$
        if $\mathsf{confidence} \geq \mathsf{confidence\_threshold}$, then add $addr$ into pred_read_set
On transactional commit
    $\mathsf{succ\_rate} := (\mathsf{succ\_rate} + \mathsf{success})/2$
    if own global_lock then
        unlock global_lock
        atomically decrement wait_count
On transactional abort
    copy write set of transaction into pred_write_set
    $\mathsf{succ\_rate} := \mathsf{succ\_rate}/2$
    if own global_lock then
        unlock global_lock
        atomically decrement wait_count
```

a conservative manner. Bloom filters provide a fast means to insert addresses, and to check the membership of an address. When a transaction reads an address, Shrink 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. A locality window parameter locality_window specifies the number of previous transactions that are used for prediction.

For the write set prediction, Shrink uses the write set of an aborting transaction. Thus, if a transaction aborts, its write set becomes the predicted write set of the next transaction of the thread. Note that Shrink 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 Shrink acts for a thread with a low success rate. First, Shrink applies the serialization affinity heuristic by observing the number of threads wait_count waiting for serialization, and using the prediction scheme with a probability proportional to wait_count. If Shrink decides not to use the prediction scheme, then the thread starts the transaction normally, as the base STM would. In case Shrink 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 Shrink serializes the starting transaction. In such a case, the thread waits to lock the global mutex global_lock before starting the transaction.

4 Experimental evaluation

We experimented Shrink with two STMs: SwissTM [7] and TinySTM [15].[8] For our experiments we choose the following values for the Shrink parameters: success = 1, succ_threshold = 0.5, locality_window = 4, confidence_threshold = 3, c_1 = 3, c_2 = 2, and c_3 = 1. Shrink was implemented in C/C++ (the same as STMs it was integrated with). We have implemented the lock variable global_lock using the pthread mutex variable.

All experiments were performed 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 (STMBench7 [8], STAMP [5] as well as the red-black tree microbenchmark) to illustrate that coupling a TM with Shrink generally boosts performance. All the results are averaged over 20 runs to reduce variation in collected data. Some of the experimental results, namely STMBench7 results with SwissTM using busy waiting, and STAMP and red-black tree with TinySTM in the appendix.

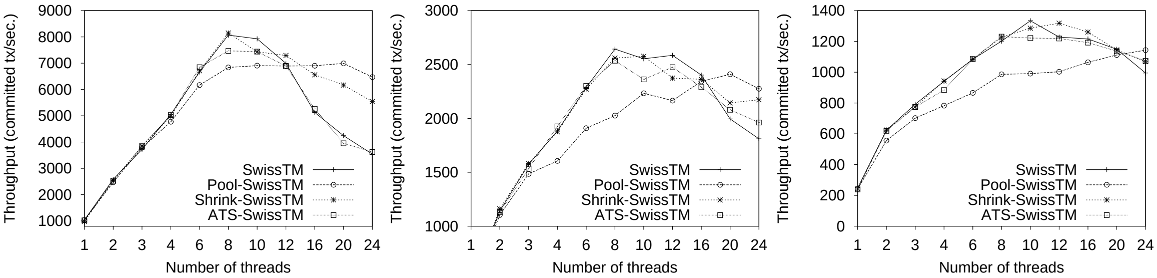
4.1 Shrink with SwissTM

We now provide experimental results of the different scheduler techniques applied to SwissTM, and compare them against the base SwissTM. Pool-SwissTM serializes all transactions that face contention. ATS-SwissTM uses the ATS serialization scheme [20]. We consider ATS to be the representative for the various coarse serialization schemes in the literature, like CAR-STM [6] and Steal-on-abort [3]. We set the preemptive waiting flag in SwissTM to true, as it shows to perform better than busy waiting when a system is overloaded.

STMBench7. We compare the performance in all three STMBench7 workloads with long traversals turned off. We compare all scheduler variants of SwissTM: Pool-SwissTM, ATS-SwissTM, and Shrink-SwissTM, as well as base SwissTM with preemptive waiting. We also analyze the performance of Shrink-SwissTM with busy waiting in the appendix.

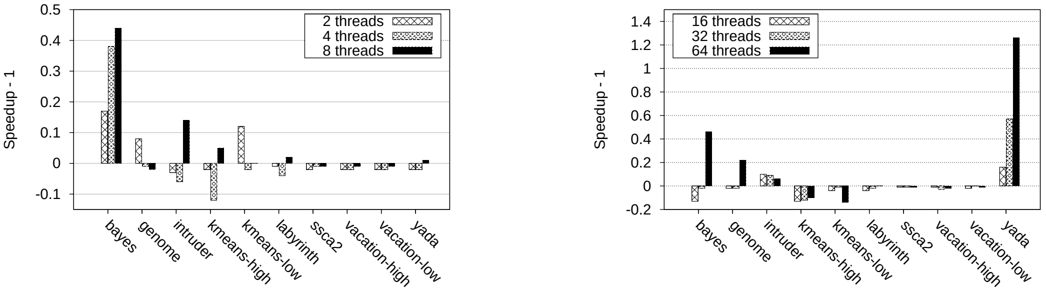
In Figure 5, we show the throughput results for the different workloads of STMBench7. Shrink-SwissTM and base SwissTM variants have comparable throughput in underloaded cases, i.e., up to 8 threads, but Shrink-SwissTM gradually outperforms base SwissTM as the system becomes more heavily overloaded. For example, in read-dominated workloads, Shrink-SwissTM performs 55% better than SwissTM when 24 threads are spawned. ATS-SwissTM incurs high overheads when system is underloaded, while still not matching Shrink-SwissTM throughput for overloaded cases. We note that in general, serialization helps as a TM becomes overloaded.

[8] Although, we have chosen STMs to demonstrate benefits of Shrink, the scheme could be implemented in hardware TMs as well.



(a) Read dominated (b) Read-write (c) Write dominated

Figure 5: SwissTM throughput on STMBench7



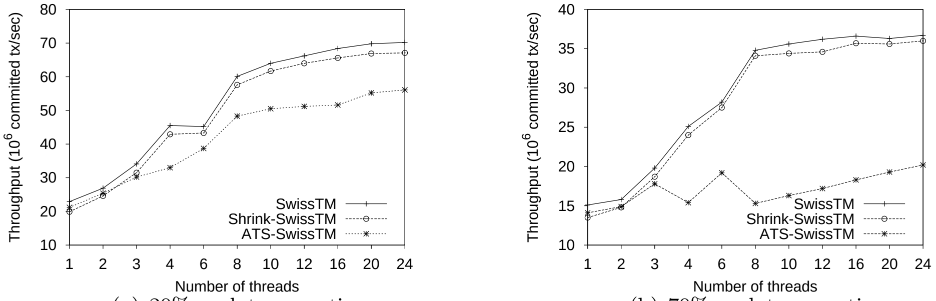
(a) Speedup in underloaded scenarios (b) Speedup in overloaded scenarios

Figure 6: Speedup of Shrink-SwissTM over base SwissTM in STAMP

STAMP. We now present our analysis of Shrink-SwissTM on the workloads in the STAMP benchmark suite. Experimental results are listed in Figure 6(a) for underloaded system (number of threads at most 8), and Figure 6(b) for overloaded system (number of threads higher than 8). For the case of 1 thread, Shrink-SwissTM and base SwissTM differ by less than 1% across all workloads, and hence we do not show these results. Overall, we obtain speedup of up to 1.4 in underloaded cases, and up to 2.2 in overloaded cases. We face a performance degradation (a speedup of up to 0.9) in one of the underloaded scenarios.

An interesting case is the *intruder* benchmark, where Shrink-SwissTM gains significantly (a speedup of 1.1 to 1.15) 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 Shrink helps Shrink-SwissTM outperform the base SwissTM. The maximum performance improvement for overloaded system is obtained in the *yada* benchmark, where we obtain a speedup of up to 1.5 in 32 threads, and 2.2 in 64 threads, due to the ability of our prediction and serialization techniques to reduce contention in the *yada* benchmark.

Red black tree. Figure 7 shows the performance of base SwissTM, Shrink-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 due



(a) 20% update operations (b) 70% update operations

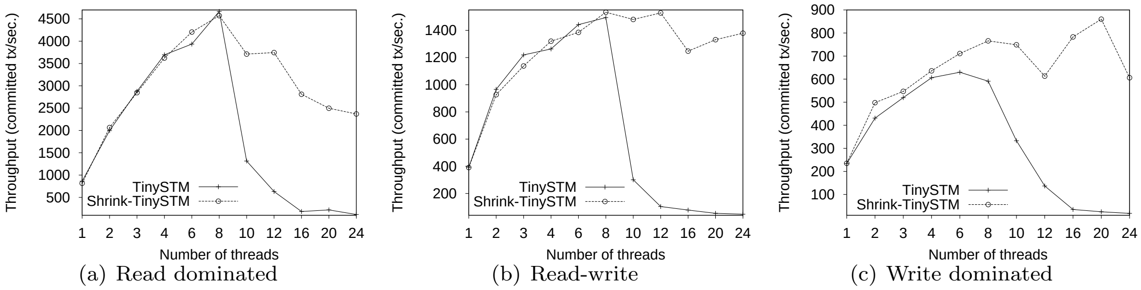
Figure 7: Throughput of SwissTM and Shrink-SwissTM red-black tree

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 Shrink-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, Shrink-SwissTM incurs a performance loss of 4% with 20% update operations (Figure 7(a)) and a performance loss of 2% with 70% update operations (Figure 7(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.

4.2 Shrink with TinySTM

We integrate our Shrink scheme in TinySTM version 0.9.5 with busy waiting. We compare the performance of Shrink-TinySTM with the base TinySTM under different benchmarks. We present the results on STMBench7 below, and the results on STAMP and the red-black tree microbenchmark in the appendix.

STMBench7. As we did for SwissTM, we compare the results on three workload types, read-dominated, read-write, and write-dominated. We also set long traversals off.



(a) Read dominated (b) Read-write (c) Write dominated

Figure 8: TinySTM throughput on STMBench7

We observe that in underloaded cases, Shrink-TinySTM shows performance comparable (±6%) to the base TinySTM. As the TM becomes overloaded, Shrink-TinySTM outperforms base TinySTM (by up to 32 times with 24 threads in write-dominated workloads). The steep fall of the performance of TinySTM in overloaded systems can be attributed to its eager lock acquisition policy and busy-waiting. 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. Shrink-TinySTM mitigates this performance loss to some extent by dynamically predicting such conflicts and avoiding them using serialization on the fly.

5 Related Work

Contention managers (CM) [9, 17] have been widely studied and used to boost the performance of transactional memories (TM). As CMs work only to resolve conflicts, they cause wasted work of at least one of the transactions. There has been recent work in the complementary direction of scheduling transactions in a TM. Yoo and Lee proposed adaptive transactional scheduling (ATS) [20], which detects threads that face high contention, and serializes them in a queue. The authors demonstrate the performance gain obtained with ATS in RSTM [12] and LogTM [13] for overloaded systems. In this paper, we argued that ATS serializes more than necessary, and showed that Shrink gives better performance than ATS in most scenarios. Ansari et al. proposed Steal-on-abort [3], which avoids wasted work by allowing transactions to steal conflicting transactions. The authors propose various strategies for the release of stolen transactions. Rossbach et al. proposed TxLinux [16] 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 unlike Shrink, none of them work to prevent conflicts.

Another proposal, CAR-STM [6] proposes Serializer, which ensures that two transactions never conflict more than once. As for any other CM, Serializer is reactive, and does not 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 hard to determine a priori because of the dynamic nature of the access sets of transactions. Most of the evaluation of 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.

Prediction in TM has also been studied for different purposes. For example, Waliullah and Stenstrom [19] use a prediction technique to insert a checkpoint before the first predicted conflicting access executes. This saves execution time up to the occurrence of the 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.

6 Conclusion

We explored the theoretical power and limitations of prediction-based TM schedulers. We introduced a novel pragmatic scheduler, Shrink, that uses locality to predict access sets in TMs. Shrink's underlying heuristic of serialization affinity encourages serialization of transactions only when the number of threads is high. Using the prediction techniques and the serialization affinity, Shrink dynamically serializes transactions based on the current contention and the likelihood of conflict with currently executing transactions. We illustrate the performance improvement obtained with Shrink with SwissTM and TinySTM. Our experiments on realistic benchmarks, like STMBench7 and STAMP, show that Shrink boosts the performance in such cases in SwissTM and TinySTM.

Our work on integrating prediction techniques with a TM opens several problems. On the theory front, a formalism to reason about the average case performance of TM schedulers shall help to theoretically justify the good performance of Shrink across various benchmarks and thread counts. On the practical front, integrating prediction techniques within the OS could provide better TM schedulers.

References

[1] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *DIS*, pages 92–107. IEEE Computer Society, 1996.

[2] 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 *HIPEAC*. Springer, January 2009.

[3] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC*, pages 308–315. ACM, 2006.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*. IEEE, 2008.

[5] David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *DISC*, pages 194–208. Springer, 2006.

[6] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *PODC*, pages 125–134. ACM, 2008.

[7] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI*. ACM, 2009. (to appear).

[8] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *EuroSys*, pages 315–324. ACM, 2007.

[9] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *DISC*, pages 303–323. Springer, 2005.

[10] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Towards a theory of transactional contention managers. In *PODC*, pages 316–317. ACM, 2006.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101. ACM, 2003.

[12] 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 *TRANSACT*. Jun 2006.

[13] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *HPCA*, pages 254–265. IEEE Computer Society, Feb 2006.

[14] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[15] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *SPAA*. ACM, Jun 2007.

[16] 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 *SOSP*, pages 87–102. ACM, 2007.

[17] William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248. ACM, 2005.

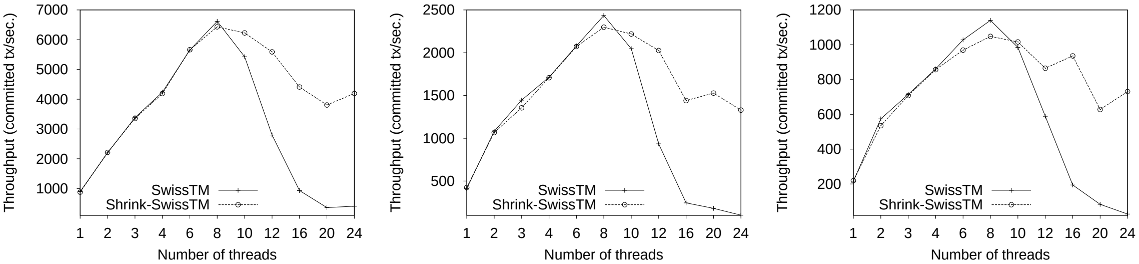
[18] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[19] M. M. Waliullah and Per Stenström. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *IPDPS*, pages 1–11. IEEE Computer Society, 2008.

[20] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA*, pages 169–178. ACM, 2008.

A Further experimental results

Shrink-SwissTM with busy waiting on STMBench7. Figure 9 compares the throughput of Shrink-SwissTM with the base SwissTM, with busy waiting, that is, threads that wait for a lock do not yield the processor. As in the case of TinySTM, we observe that Shrink improves the performance of SwissTM drastically in overloaded cases. This is because, as the number of threads increases, the performance of base SwissTM drops steeply due to busy waiting. On the other hand, Shrink-SwissTM maintains a good throughput in overloaded cases.

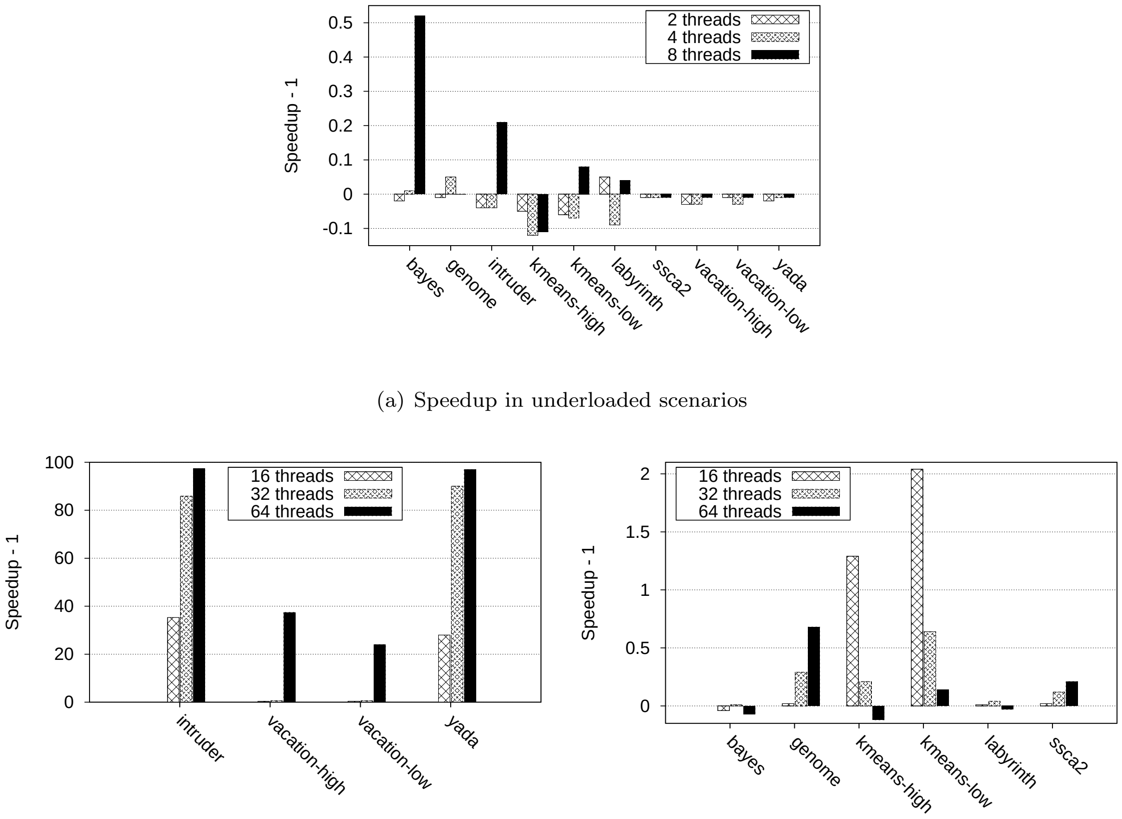


(a) Read dominated (b) Read-write (c) Write dominated

Figure 9: SwissTM (live waiting) throughput on STMBench7

Shrink-TinySTM on STAMP. Figure 10 shows speedup obtained by Shrink-TinySTM over the base TinySTM. Shrink-TinySTM improves the performance of most of the benchmarks in overloaded cases. For some benchmarks like *intruder*, *vacation*, and *yada*, we observed that as the number of threads increase, the performance of base TinySTM drastically suffers. In such cases, the serialization of Shrink-TinySTM helps to maintain good performance, giving up to about 100 times speedup. As with SwissTM, Shrink also improves the performance of *intruder* in 8 threads. Shrink-TinySTM suffers a performance degradation (a speedup of 0.9) in some underloaded cases.

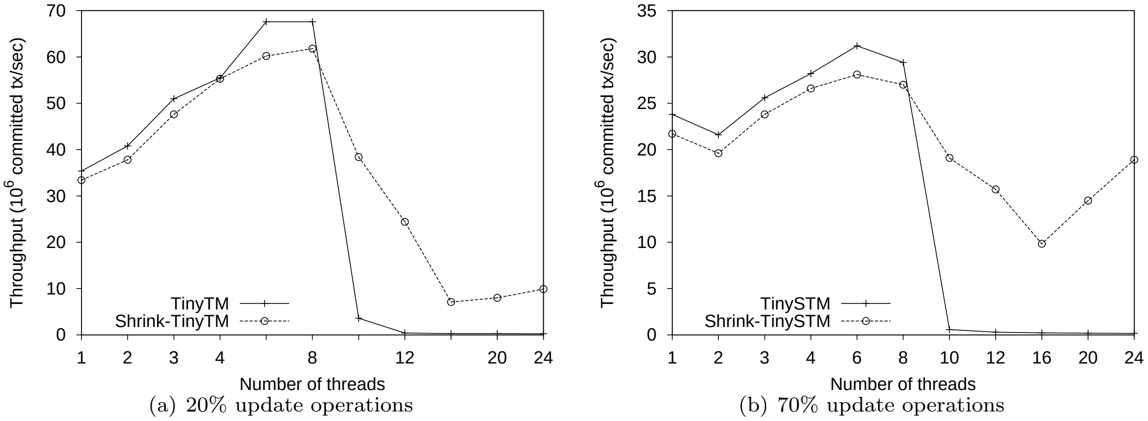
Shrink-TinySTM on red black tree. Figure 11 shows the performance of base TinySTM and Shrink-TinySTM on the red-black tree microbenchmark. We perform our experiments under 20% and 70% update operations and integer set range of 16384. We observe that the overheads of Shrink incur a performance loss of up to 10% in the case of 1 thread. But, as the TM becomes overloaded, the performance of base TinySTM drops sharply. The performance of Shrink-TinySTM also drops, but remains an order of magnitude higher than that of the base TinySTM. For example, with 12 threads and 20% update operations, base TinySTM has a throughput of 0.4 million transactions per second, whereas Shrink-TinySTM has a throughput of 24.4 million transactions per second.



(a) Speedup in underloaded scenarios

(b) Speedup in overloaded scenarios

Figure 10: Speedup of Shrink-TinySTM over base TinyTM in STAMP



(a) 20% update operations

(b) 70% update operations

Figure 11: Throughput of TinySTM and Shrink-TinySTM on red-black tree