RELSTM: A Proactive Transactional Memory Scheduler *

David Sainz Hagit Attiya

Computer Science Department, Technion

dsainz,hagit@cs.technion.ac.il

Abstract

A major impediment to achieving high performance in transactional memory is repeated aborts due to conflicts among transactions. *Transaction scheduling* mitigates these effects by delaying a transaction until possible conflicting transactions have completed. The paper presents RELSTM, a new transaction scheduler that attempts to predict future conflicts on the basis of *second-hop conflicts*, namely, conflicts through an intermediate transaction. Our experimental evaluation shows that RELSTM provides performance benefits in high-contention workloads on many cores.

1. Introduction

Transactional memory (TM) has been widely explored as a convenient approach to concurrent programming, since it allows the programmer to abstract problematic details such as thread synchronization or data consistency. Instead of manually synchronizing shared data structures in critical sections, those sections are wrapped in transactions and coded without requiring synchronization, as non-concurrent programs. Transactions are then executed, aborted upon conflicts accessing shared data, and restarted again from the beginning until they finish without conflicts.

The strategies used to deal with conflicts possibly play a major role in the performance of transactional memories, as a transaction abort means wasted work. Several alternatives have been proposed [8, 13], which usually abort at least one transaction from the ones involved in a conflict, striving for a minimum performance deterioration. Transactional memories include a module called *contention manager* (CM) to manage those conflicts and decide which transactions to abort.

* The research leading to these results has received funding from the European Union's - Seventh Framework Programme (FP7/2007-2013]) under grant agreement number 238639, MC-TRANSFORM.

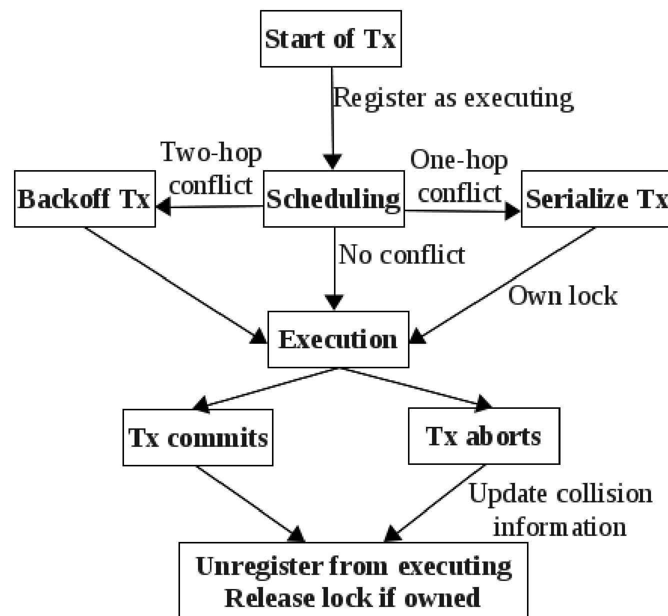


Figure 1. RELSTM flow chart.

Another appealing module studied to improve performance is a *transaction scheduler*, which controls the order and timing of transaction executions. It adds policies to decide when transactions can start, either for the first time or after an abort. This module has recently gained interest [4, 7] as a means to avoid aborts before they occur, hence improving the total performance. Dragojević et al. [7] propose an algorithm based on accesses of memory addresses (read sets and write sets), to measure potential conflicts between them, based on the assumption that transactions in a thread access similar memory addresses. Atoofian [4] devises an algorithm that schedules transactions based on how many times they have aborted already. CAR-STM [6] and LO-SER [10] are schedulers that serialize aborted transactions after the transactions they

conflicted with, therefore preventing the same conflicts from happening again.

Although several studies have been done on transaction schedulers, there are many possibilities yet to be explored. We propose a new point of view based on analyzing *relations* between transactions. These relations are mainly of *conflict* relations: two transactions are related if they conflict with each other. This idea can be extended to cover *second-hop conflicts*, namely *transactions* T_{1h} that conflicted directly with a specific transaction T_x and in turn, transactions T_{2h} (*second-hop transactions*) that conflicted with some transaction in T_{1h}. Recursively, these relations can be extended to n-hop relations. It is intriguing to analyze first and second-hop conflicts between transactions and their impact on transaction scheduling.

Following this idea, we propose RELSTM, a Transaction Scheduler that tracks conflict relations between transactions. Upon a conflict, each transaction registers its opponent as well as those transactions the opponent conflicted with. Upon abort, the transaction is backed off if the direct opponent or a predefined percentage of second-hop transactions are still in execution. After being blocked, the transaction resumes again. This algorithm is a *proactive scheduler* in the sense that it is able to activate the scheduling policy before a given conflict occurs. If a transaction T_a is a second-hop transaction of T_b and T_a is still in execution, RELSTM can proactively block T_b to prevent a future conflict. *Reactive schedulers*, on the other hand, only react to conflicts after they occur.

We have implemented RELSTM over TinySTM [12] and evaluated it against the benchmarks included in the STAMP suite [11]. This evaluation is compared with the same benchmarks against TinySTM and CAR-STM, to assess in which cases there is a performance improvement. Our findings reveal that RELSTM achieves more transaction throughput in tests that generate a significant number of aborts, especially when a high number of CPU cores is used.

The rest of this paper is structured as follows: Section 2 describes related work, Section 3 offers details about the implementation of RELSTM, Section 4 shows the evaluation results and Section 5 concludes with a discussion.

```
Upon transaction Tx start:
 if (Tx.conflicted is not NULL)
     if Tx.conflicted is registered in executing array
         lock on serial_lock
 else if (Tx.twohop is not empty)
     tx_executing: Number
     for each ID in Tx.twohop
         if (ID is registered in executing array)
             increment tx_executing
     if ((executing / size of Tx.twohop) >= TWOHOP_RATE)
         backoff during Tx.backoff time
         increment Tx.backoff for exponential backoff
 end if
```

Figure 2. RELSTM pseudocode for the transaction scheduling before a transaction starts. The code uses a global array to register and deregister transactions in execution, as well as a lock to serialize transactions.

2. Related Work

Along with the comprehensive studies on contention managers [1, 8, 13, 14, 16], there has been recent interest in transaction schedulers to improve conflict management. Yoo and Lee implemented an adaptive transaction scheduler (ATS) [17] which tracks the system contention level, and sends transactions to a scheduler to be serialized whenever that level surpasses a given threshold. The contention is measured counting the transaction aborts of each thread and contrasting it with past aborts. The formula can be configured to give more importance to either the abort history or the present conflict. Dragojević et al. [7] devised the SHRINK scheduler, which predicts conflicts between transactions by tracking the memory addresses they access. The algorithm is based on the concept of *temporal locality* [2, 15], which suggests that transactions executed recently before a given transaction T_x accessed a similar set of adresses than T_x. When transactions read, commit or abort, memory addresses are added to predicted read and predicted write sets. Upon start, a transaction checks whether its memory accesses belong to the sets. If so, transactions are serialized in a global queue. SHRINK also implements a contention detector that activates the prediction algorithm only under high contention cases. This contention detector is calculated by counting aborted and committed transactions. One of the main features of the algorithm is that it is a proactive scheduler that can apply a policy before a transaction aborts for the first time. The lock acquisition policy of transactions affects the performance of SHRINK, since in lazy lock acquisition conflicts are detected only at commit time and last for less time. This situation lim-

```
Transaction data:
emph{Other transaction information...}
 ID: Number
 Twohop: Array
                              \\list of transaction IDs that had a two-hop conflict
 Conflicted: Number
                              \\Transaction ID of the last one-hop lost conflict
 Killed: Number
                              \\Transaction ID of the last one-hop won conflict
 Backoff: Number
                              \\Current backoff time to wait

Upon conflict between T1 and T2:
 if (T2.conflicted is not NULL
 and T2.conflicted <> T1.ID)
     Add T2.conflicted to T1.twohop
 if (T2.killed is not NULL
 and T2.killed <> T1.ID
 and T2.killed <> T2.conflicted)
     Add T2.killed to T1.twohop
 if (T1.conflicted is not NULL
 and T1.conflicted <> T2.ID)
     Add T1.conflicted to T2.twohop
 if (T1.killed is not NULL
 and T1.killed <> T2.ID and T2.killed <> T2.conflicted)
     Add T1.killed to T2.twohop
 if (T1 won)
     T2.conflicted = T1.ID
     T1.killed = T2.ID
 else
     T1.conflicted = T2.ID
     T2.killed = T1.ID
 end if

Upon abort:
 Unegister transaction in executing array
 if (queue_lock.owned)
     Unlock queue_lock

Upon commit:
 Unregister transaction in executing array
 Empty twohop
 if (queue_lock.owned)
     Unlock queue_lock
```

Figure 3. RELSTM pseudocode for the commit, abort and conflict events of a transaction. The code uses a global array to register and deregister transactions in execution, as well as a lock to serialize transactions.

its the capacity of the the algorithm to detect a conflict when a transaction starts.

Steal-on-abort [3] is another scheduling technique in which aborted transactions are reordered and serialized after the winning transaction. This technique reduces the number of aborts when collisions between the same transactions are frequent, since a rolled back transaction is not able to restart until the winner is not in execution anymore.

CAR-STM [6] is a scheduler that ensures the same two transactions will not collide more than once. It maintains a queue for each core where transactions are run in the order they arrive. Upon collision, the aborted transaction is serialized in the queue of the winner core, effectively serializing the loser after the winner. CAR-STM also uses a conflict-probability prediction routine to compute collision probabilities before they occur. However, this routine relies on conflict information specifically given by the application, which in most cases is hard to provide.

LO-SER [10] is a transaction serialization mechanism which serializes conflicted transactions to avoid the occurrence of the same conflict. The scheduler is completed with a set of adaptive algorithms that apply LO-SER only when enough contention is detected, either after aborting a number of times or after reaching a certain general contention level. The LO-SER algorithm itself is in charge of transaction serialization. This approach avoids the use of queues and is based on condition variables. Each transaction owns a condition variable and after a conflict between two transactions, the aborted transaction waits on the condition variable of the winner. Due to the inherent properties of condition variables, several transactions can be serialized after the winner, being released after it commits.

Bimodal [5] is a scheduler that resorts to serialization in order to reduce repeated aborts of the same transactions. Bimodal is able to distinguish between reading and writing transactions, therefore being able to give different priorities depending on such types. In read-dominated workloads, attempting to serialize in every conflict may result in serializing read transactions that shouldn't wait for each other. To avoid this problem, a model is proposed in which each core has its own transactions queue, and share one general queue meant for read-only transactions. Bimodal alternates between 2 modes, or epochs, that differ in the priority that is given to read-only and write transactions. The algo-

Benchmark	Parameters
Bayes	-e8 -i2 -n10 -p40 -r4096 -v32
Genome	-g16384 -n16777216 -s64
Intruder	-a10 -l128 -n262144 -s1
Labyrinth	-i random-x512-y512-z7-n512.txt
SSCA2	-s20 -i1.0 -u1.0 -l3 -p3
Vacation High	-n2 -q90 -u98 -r1048576 -T4194304
Vacation Low	-n4 -q60 -u90 -r1048576 -T4194304
Kmeans High	-m40 -n40 -T0.00001 -i random-n65536-d32-c16.txt
Kmeans Low	-m15 -n15 -T0.00001 -i random-n65536-d32-c16.txt
Yada	-a15 -i ttimeu1000000.2

Table 1. Parameters used in the STAMP benchmark.

rithm serializes write transactions in each core and read transactions in the general queue. This mechanism allows concurrent execution of read transactions in read epochs.

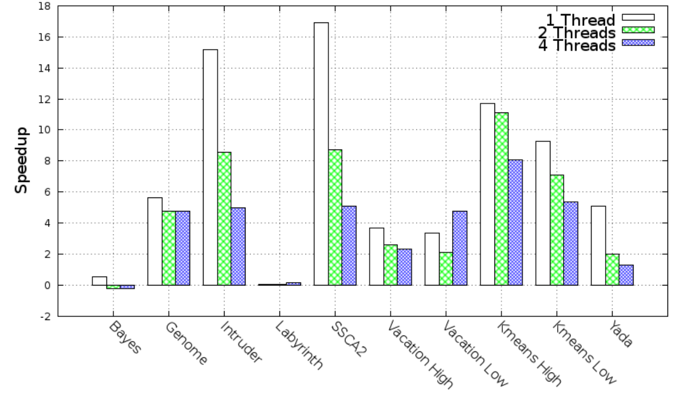
Speculative Contention Avoidance (SCA) [4] is another prediction algorithm based on how many times a transaction has aborted already. It relies on a contention predictor that tracks the contention level of each thread. This predictor acts as a counter and is incremented when a transaction aborts in the thread. When a transaction commits, the counter is reset. Once the predictor surpasses a given threshold, contention is assumed to be too high and transactions arriving to the thread are serialized. This technique reduces the number of aborts under high contention and can be applied before transactions are executed for the first time.

3. RELSTM implementation

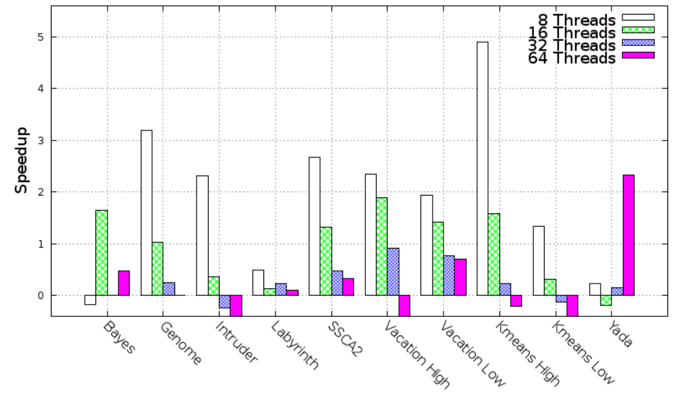
RELSTM aims to analyze how conflict relations between transactions impact the performance of STMs. We have implemented a first version of the algorithm in order to assess the benefits of such analysis. RELSTM uses first and second hop conflict information to schedule transactions, along with a general array to register and track transactions currently in execution.

Each transaction T_x is identified by a unique ID number, as well as different fields to track transactions that conflicted with it. The field *conflicted* stores the ID of the last transaction that collided and won against T_x. The field *killed* holds the ID of the last transaction that collided and lost against T_x. The array *twohop* keeps the two-hop transactions of the opponents in a conflict with T_x. Transactions can also access a global lock named *queue_lock* in order to be serialized.

Figure 1 depicts the flowchart of the scheduler algorithm. Figure 2 shows the RELSTM pseudocode for the algorithm part executed when a transaction starts. First, the transaction registers itself as executing and then it checks its conflicted field for one-hop conflicts. If there has been a previous collision with an opponent still in execution, the transaction blocks itself on the global lock in order to be serialized. Otherwise, it begins checking its two-hop conflicts and tracking which ones are still running. If the rate of executing two-hop transactions is greater than or equal to a given system constant *TWOHOP_RATE*, the transaction is backed off. The reason why the two types of collisions are not checked at the same time if they both exist is to avoid excessive waiting penalty to transactions that have multiple relations.



(a) Low number of threads



(b) High number of threads

Figure 4. Speedup of RELSTM over CAR-STM using the STAMP suite.

Figure 3 shows the RELSTM pseudocode for the conflict, abort and commit events of a transaction. When two transactions conflict, the collision is tracked and used in the scheduling algorithm. First, transactions add the opponent's two-hop relations. Then, the transaction that won the conflict adds the opponent to its killed field, while the transaction that lost populates its conflicted field.

Upon commit and abort, transactions deregister themselves in the executing array and proceed to release the global lock if they own it.

For the implementation we chose TinySYM 1.0.3 [12], adding the RELSTM code in C on it. This serves as an initial proof of concept to evaluate if this research direction helps improve the performance of an existing STM. We plan to add more optimizations in the future. We chose TinySTM for its ease of code modification:

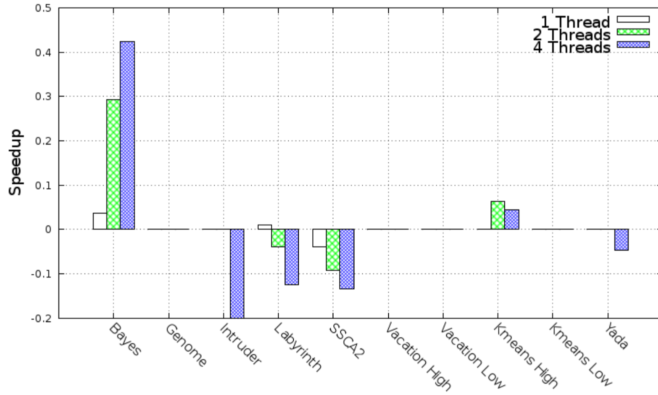


Figure 5. Speedup of RELSTM over TinySTM 1.0.3 using the STAMP suite. Scenarios with low thread number

it has documented code as well as an active support forum. Another reason that lead us to choose TinySTM is that many STMs and schedulers, including CAR-STM, have been built on it.

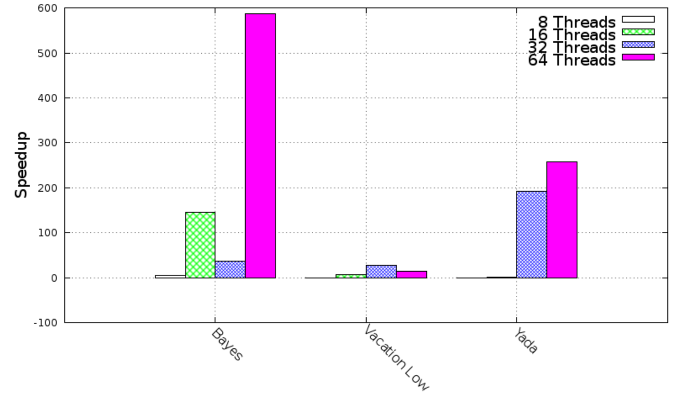
We added to TinySTM the extra transaction fields and tracked the 1-hop and 2-hop transactions in the code sections where conflicts are detected. After a rollback, we redirect the code flow to our own scheduler, written in a separate code file.

4. Evaluation

We have evaluated RELSTM against the different tests in the STAMP [11] benchmark and compared the results to the performance achieved by TinySTM 1.0.3 [12]. We have also compared our STAMP results to those of CAR-STM [6] built over TinySTM 1.0.3.

We tested the different STMs on a PowerEdge R815 machine, with a total of four 16-core processors AMD Opteron 6272 at 3,2 GHz, a total of 64 cores and 128 Gb of RAM to run the tests. The parameters used in the benchmarks appear in Table 2. TinySTM was configured using visible reads, modular contention manager and a write-back design with encounter-time lock acquisition.

The system parameter TWOHOP_RATE was chosen to be 0.3. After several experiments, it became apparent that a value less than 0.5 (T_x has half of its 2-hop transactions in execution) was enough to justify a backoff, since there are enough active transactions in the system for a collision. Values inferior to 0.2 (T_x has 20% of its 2-hop transactions in execution) proved to be more ineffective, as the backoff was too big a penalty for the minor chance of collision.



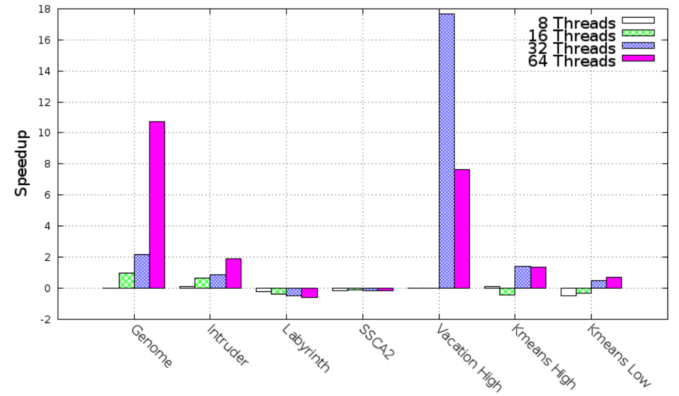


Figure 6. Speedup of RELSTM over TinySTM 1.0.3 using the STAMP suite. Scenarios with high thread number

To the best of our knowledge, TinySTM is one of the most efficient STMs available and has been continuously updated and optimized. We chose it to assess whether proactive transaction scheduling could improve the performance of an already optimized STM.

Since the STAMP results are not deterministic, we ran each test 10 times and averaged the results.

The graphs shown in this section represent speedup relative to RELSTM, namely, $\frac{(t_x - t_R)}{t_R}$ where t_R is the test completion time taken by RELSTM and t_x is the test completion time of the other STM in the comparison. Higher positive values are better, zero values represent no performance gain, whereas negative values represent performance drop.

Figure 4 shows the speedup achieved with RELSTM over CAR-STM. The figure is divided into low number of threads (from 1 to 4) and high number of threads (from 8 to 64). Figure 7 and figure 8 show the absolute benchmark completion times of both STMs. The figures show that for low number of threads, RELSTM achieves more throughput, being in some cases

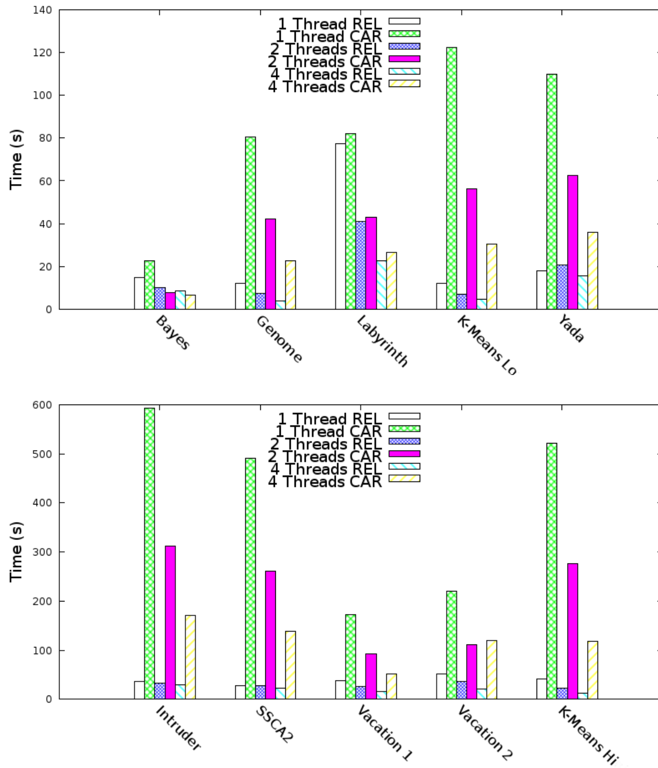


Figure 7. Benchmark completion time of RELSTM and CAR-STM with low thread number (using the STAMP suite).

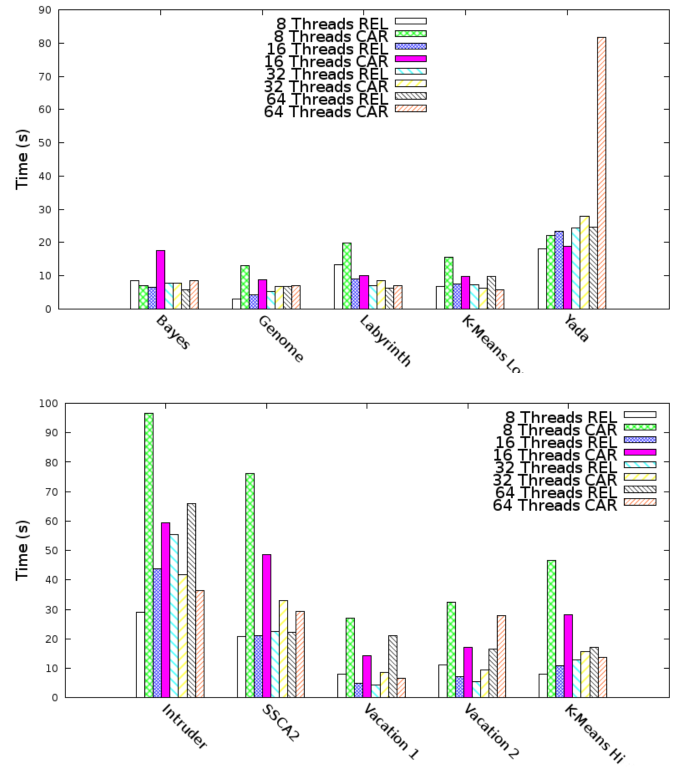


Figure 8. Benchmark completion time of RELSTM and CAR-STM with high thread number (using the STAMP suite).

(SSCA2) more than 16 times faster. The overhead CAR-STM imposes seems too harsh for low level of concurrency, and transactions take longer to finish. With higher number of threads the performance improvement of RELSTM, while still considerable, is smaller, suggesting that with a high enough number of cores both STMs would converge. CAR-STM reaches better completion times as concurrency and contention increase. Still RELSTM manages to complete the tests faster and has a performance gain regardless of the concurrency level, with the exception of some benchmarks with 64 threads where it is possible to see a small performance drop. An interesting case is the Yada test for 64 threads, where the tendency to converge with CAR-STM seems to break. Yada leads to a very high number of aborts, and CAR-STM seems to lose throughput in that highly concurrent case. We think that for CAR-STM the cause might be aborted transactions being serialized unevenly in a few queues. This would lead to a situation where most of the work is done by a small set of cores, hindering parallelism.

Figure 5 depicts the speedup of RELSTM over TinySTM version 1.0.3 with low number of threads (from 1 to 4). Figure 9 shows the same information in absolute benchmark completion times of each STM. The figures show a noticeable performance improvement for the Bayes and Kmeans High tests, while Genome, Vacation and Kmeans Low do not achieve a noticeable performance gain. In these last tests RELSTM did not manage to improve completion times but as shown, the extra code did not impose an overhead. Some performance degradation can be seen in some tests.

The speedup over TinySTM in high number of threads (8 to 64) is shown in Figure 6. Figure 10 shows the same information in absolute completion time of each STM. In this case a dramatic performance gain is achieved for the Bayes, Vacation Low and Yada tests. In these tests, TinySTM leads to a high number of aborts, while RELSTM mitigates them via serialization and backoff. Some cases obtain results from 200 to 600 times faster. The other tests yield a more moder-

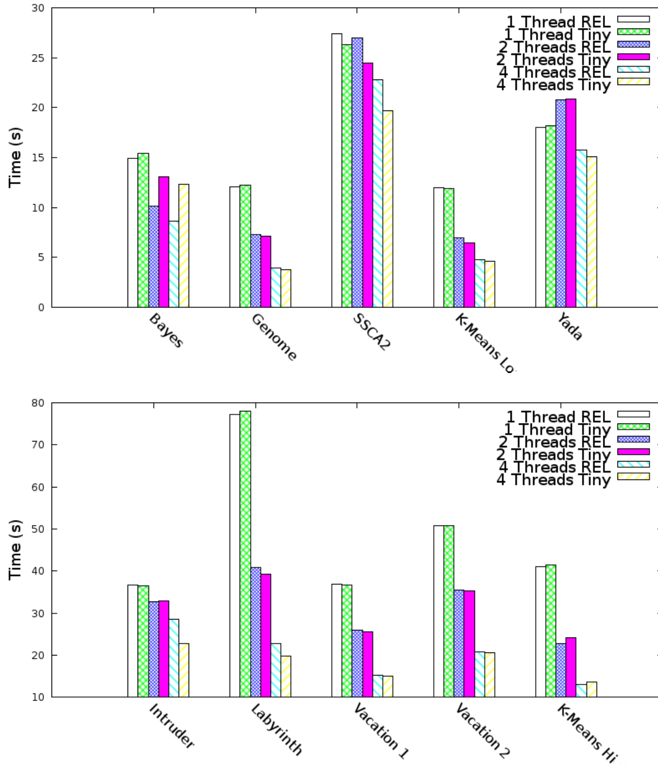


Figure 9. Benchmark completion times of RELSTM and TinySTM with low thread number (using the STAMP suite).

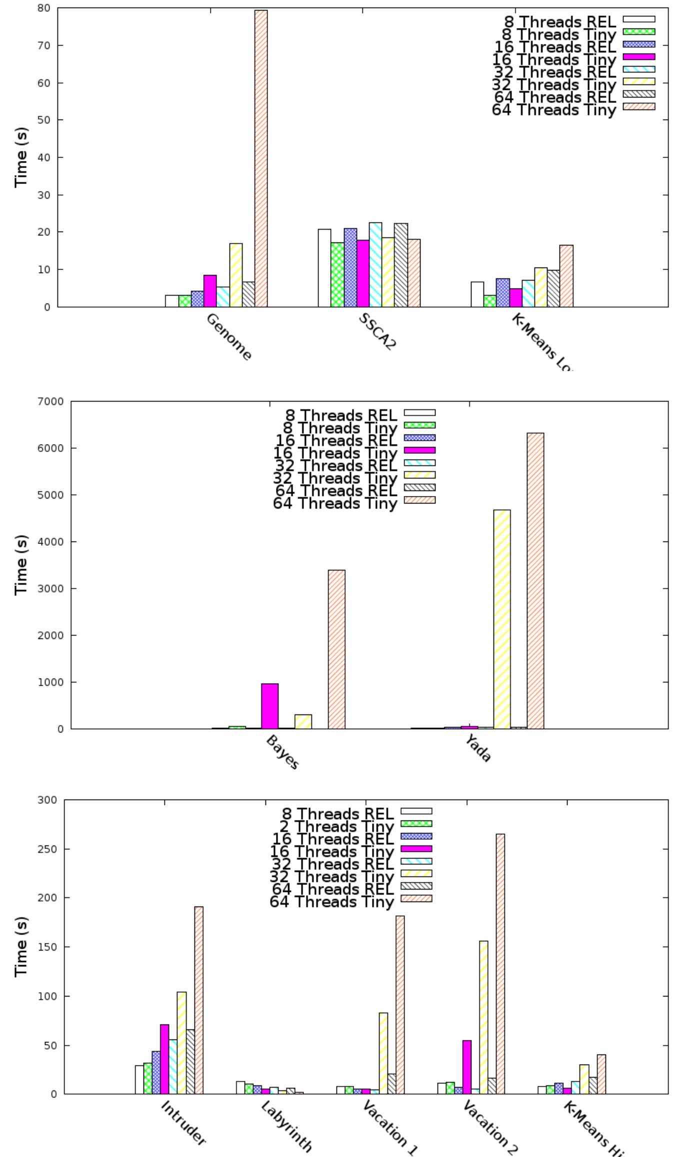


Figure 10. Benchmark completion times of RELSTM and TinySTM with high thread number (using the STAMP suite).

ate but still noticeable improvement, with the exception of Labyrinth and SSCA2 where there is a degradation. Those tests have a very low number of aborts even with high concurrency, and the RELSTM algorithm is not activated frequently. Nevertheless, the speedup over TinySTM still improves performance in most cases.

5. Conclusion

This paper suggests that a proactive approach, based on tracking two-hop relations between transactions, to transaction scheduling can improve performance in situations of high thread count and high abort rates.

There are several additional improvements we hope to explore. A natural optimization is to set a contention threshold for activating the scheduler. However, it seems that the overhead for tracking the number of transactions and aborts accounts for most of the cost of RELSTM; thus,significant improvement will necessitate a very fast mechanism for registering and deregistering transactions. We also plan to compare RELSTM with other transaction schedulers, in particular Shrink and LO-SER, as well as perform evaluations using STMBench7 [9].

Finally, it would be interesting to explore other proactive policies, going beyond two-hop conflicts.

Acknowledgements: The authors thank Adi Suissa for many helpful discussions.

References

[1] E. Akpinar, S. Tomić, A. Cristal, O. Unsal, and M. Valero. A comprehensive study of conflict resolution policies in hardware transactional memory. *6th*

ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), Jun 2011.

[2] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*, pages 92 –103, dec 1996.

[3] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 4–18, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] E. Atoofian. Speculative contention avoidance in software transactional memory. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1417–1423, 2011.

[5] H. Attiya and A. Milani. Transactional scheduling for read-dominated workloads. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 3–17, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] S. Dolev, D. Hendler, and A. Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 125–134, New York, NY, USA, 2008. ACM.

[7] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 7–16, New York, NY, USA, 2009. ACM.

[8] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In P. Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 303–323. Springer Berlin Heidelberg, 2005.

[9] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[10] T. Heber, D. Hendler, and A. Suissa. On the impact of serializing contention management on stm performance. *Journal of Parallel and Distributed Computing*, 72(6):739 – 750, 2012.

[11] C. C. Minh. Stamp: Stanford transactional applications for multi-processing. *IEEE IISWC*, pages 35–46, 2008.

[12] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[13] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[14] G. Sharma, B. Estrade, and C. Busch. Window-based greedy contention management for transactional memory. In *Proceedings of the 24th international conference on Distributed computing*, DISC'10, pages 64–78, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, sep 1982.

[16] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 141–150, New York, NY, USA, 2009. ACM.

[17] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 169–178, New York, NY, USA, 2008. ACM.