Steal-on-abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering

Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson

The University of Manchester
{ansari,mikel,kotselidis,jarvis,chris,watson}@cs.manchester.ac.uk

Abstract. In transactional memory, aborted transactions reduce performance, and waste computing resources. Ideally, concurrent execution of transactions should be optimally ordered to minimise aborts, but such an ordering is often either complex, or unfeasible, to obtain.
This paper introduces a new technique called *steal-on-abort*, which aims to improve transaction ordering at runtime. Suppose transactions A and B conflict, and B is aborted. In general it is difficult to predict this first conflict, but once observed, it is logical not to execute the two transactions concurrently again. In steal-on-abort, the aborted transaction B is stolen by its opponent transaction A, and queued behind A to prevent concurrent execution of A and B. Without steal-on-abort, transaction B would typically have been restarted immediately, and possibly had a *repeat conflict* with transaction A.
Steal-on-abort requires no application-specific information, modification, or offline pre-processing. In this paper, it is evaluated using a sorted linked list, red-black tree, STAMP-vacation, and Lee-TM. The evaluation reveals steal-on-abort is highly effective at eliminating repeat conflicts, which reduces the amount of computing resources wasted, and significantly improves performance.

1 Introduction

In the future, software will need to be parallelised to take advantage of the increasing number of cores in multi-core processors. Concurrent programming, using *explicit locking* to ensure safe access to shared data, has been the domain of experts, and is well-known for being challenging to build robust and correct software. Typical problems include data races, deadlock, livelock, priority inversion, and convoying. The move to multi-cores requires adoption of concurrent programming by the majority of programmers, not just experts, and thus simplifying it has become an important challenge.

Transactional Memory (TM) is a new concurrent programming model that seeks to reduce programming effort, while maintaining or improving execution performance, compared to explicit locking. TM research has surged due to the need to simplify concurrent programming. In TM, programmers mark those blocks of code that access shared data as *transactions*, and safe access to shared data by concurrently executing transactions is ensured *implicitly* (i.e. invisible to the programmer) by the TM implementation.

The TM implementation compares each transaction's data accesses against all other transactions' data accesses for conflicts. Conflicts occur when a transaction has a) read a data element and another transaction attempts to write to it, or b) written to a data element and another transaction attempts to read or write to it. If conflicting data accesses are detected between any two transactions, one of them is *aborted*, and usually restarted immediately. Selecting the transaction to abort, or *conflict resolution*, is based upon a policy, sometimes referred to as a *contention management policy*. If a transaction completes execution without aborting, then it *commits*, which makes its changes to shared data visible to the whole program.

Achieving scalability on multi-core architectures requires, amongst other things, the number of aborted transactions to be kept to a minimum. Aborted transactions reduce performance, reduce scalability, and waste computing resources. Furthermore, in certain (update-in-place) TM implementations aborted transactions require extra computing resources to roll back the program to a consistent state.

The order in which transactions concurrently execute can affect the number of aborts that occur, and thus affect performance. Although it may be possible to determine an optimal order (or schedule) that minimises the number of aborts given complete information a priori, in practice this is difficult to achieve. Often complete information is impractical to obtain, simply not available for some programs, e.g. due to dynamic transaction creation, or even if it is available, the search space for computing the optimal order may be unfeasibly large.

This paper presents a new technique called *steal-on-abort*, which aims to improve transaction ordering at runtime. When a transaction is aborted, it is typically restarted immediately. However, due to close temporal locality, the immediately restarted transaction may repeat its conflict with the original transaction, leading to another aborted transaction. Steal-on-abort targets such a scenario: the transaction that is aborted is not restarted immediately, but instead 'stolen' by the opponent transaction, and queued behind it. This prevents the two transactions from conflicting again.

Crucially, steal-on-abort requires no application-specific information or configuration, and requires no offline pre-processing. Steal-on-abort is implemented in DSTM2 [1], a Software TM (STM) implementation, and evaluated using a sorted linked list [2], red-black tree [2], STAMP-vacation [3], and Lee-TM [4, 5]. The evaluation shows steal-on-abort to be highly effective at reducing repeat conflicts, which lead to performance improvements ranging from 1.2 fold to 290.4 fold.

The paper is organised as follows: Section 2 introduces steal-on-abort, its implementation in DSTM2, the different steal-on-abort strategies developed, and related work. Section 3 evaluates steal-on-abort, presenting results for transaction throughput, repeat conflicts, and briefly examining steal-on-abort overhead. Finally, Section 4 presents the conclusions.

2 Steal-on-abort

In most TM implementations, aborted transactions are immediately restarted. However, we observed that the restarted transaction may conflict with the same opponent transaction again, leading to another abort, which we refer to as a *repeat conflict*. In general it is difficult to predict the first conflict between any two transactions, but once a conflict between two transactions is observed, it is logical not to execute them concurrently again (or, at least, not to execute them concurrently unless the repeat conflict can be avoided). Using steal-on-abort the aborter steals the abortee, and only releases its stolen transactions after committing. This prevents them from being executed concurrently, which reduces wasted work. However, steal-on-abort also aims to improve performance. When a transaction is abort-stolen, the thread that was executing it acquires a new transaction and begins executing it.

An advantage of steal-on-abort is that it complements existing contention management policies. Since steal-on-abort is only engaged upon abort, existing contention management policies can continued to be used to decide which transaction to abort upon conflict.

The remainder of this section explains the implementation of steal-on-abort in DSTM2, and then explores the steal-on-abort design space by suggesting several execution strategies. The implementation needs to support three key components of steal-on-abort. First, each thread needs to be able to store the transactions stolen by its currently executing transaction. Second, each thread needs to be able to acquire a new transaction if its current transaction is stolen. Finally, a safe mechanism for stealing active transactions is required.

2.1 Multiple Work Queue Thread Pool with Randomized Work Stealing

DSTM2, like other STMs [6–8], creates a number of threads that concurrently execute transactions. This is extended into a thread pool model, and application threads submit transactional jobs to a transactional thread pool. As shown in Figure 1, a work queue is added to each worker thread in the transactional thread pool (`java.util.concurrent.LinkedBlockingDeque`, a thread-safe deque) to store transactional jobs. A transactional job is simply an object that holds the information needed to execute a transaction (e.g., pointer to a function, and parameters). Multiple work queues are used as a single work queue would lead to high serialisation overhead, and submitted jobs are round robin distributed to work queues. Worker threads acquire transactions from the head of their own queue when their current transaction commits, or is abort-stolen.

In order to keep worker threads busy, randomised work stealing [9] is implemented as well. The terms *work-steal* and *abort-steal* are used to differentiate

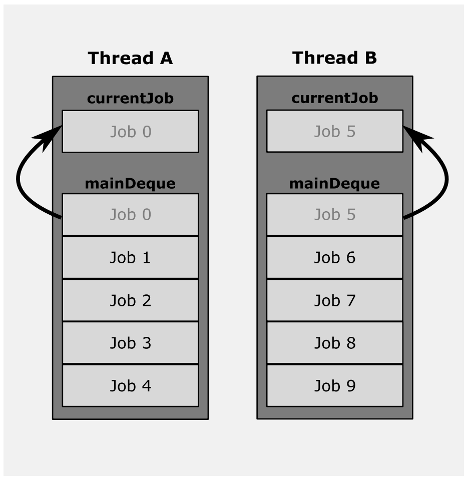

Fig. 1. Worker threads have per-thread deques that store transactional jobs. Worker threads take jobs from the head of their own deque.

between transactions being stolen due to work stealing, and due to steal-on-abort, respectively. As shown in Figure 2, if a worker thread's own work queue is empty, it randomly selects another worker thread, and attempts to work-steal a single transactional job from the tail of that thread's work queue. If all work queues are empty, the thread will attempt to work-steal from other worker threads' steal queues (described next).

2.2 Steal-on-abort Operation

A private steal queue (also a `java.util.concurrent.LinkedBlockingDeque`) is added to each worker thread to hold transactional jobs that are abort-stolen, as shown in Figure 3, which illustrates steal-on-abort in action. Each worker thread has an additional thread-safe flag, called `stolen`.

A stealing worker thread attempts to abort its victim worker thread's transaction. If this attempt is successful, the stealing thread takes the job stored in the victim thread's `currentJob` variable, and stores it in its own steal queue. After the job is taken, the victim thread's `stolen` flag is set. If a victim thread detects its transaction has been aborted, it waits for its `stolen` flag to be set. Once the flag is set, the victim thread obtains a new job, stores it in `currentJob`, and then clears the `stolen` flag. The victim thread must wait on the `stolen` flag, otherwise access to the variable `currentJob` could be unsafe.

2.3 Programming Model Considerations

There are two important programming model changes to consider when using a transactional thread pool. First, in our implementation, application threads

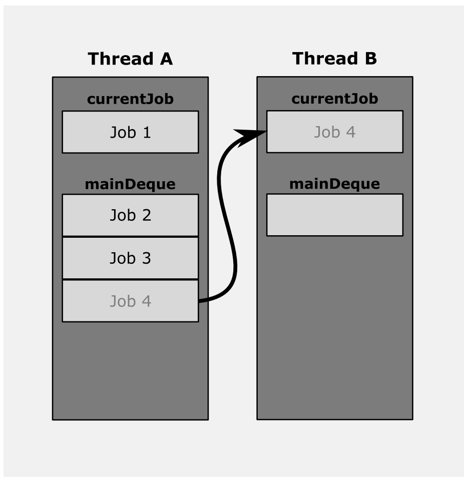

Fig. 2. If a worker thread's deque is empty, it work-steals a job from the tail of another randomly selected worker thread's deque.

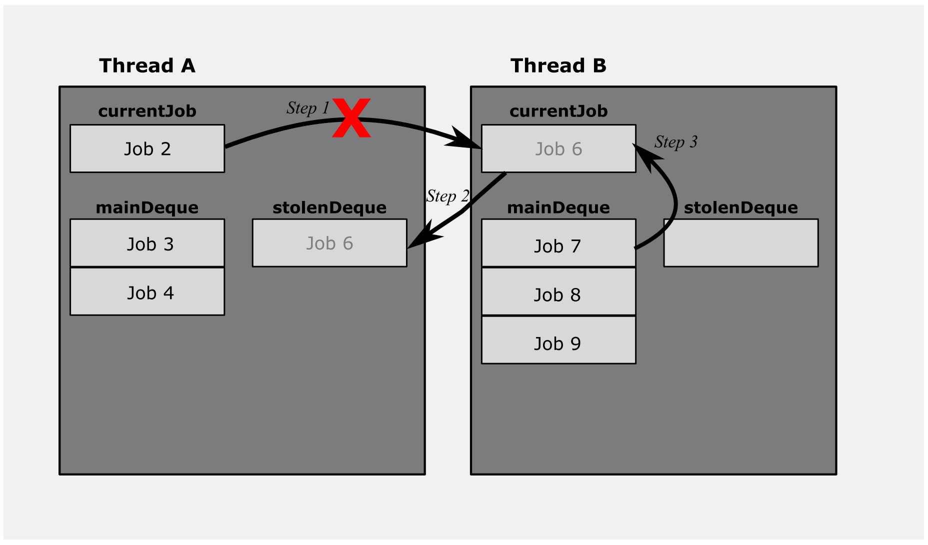

Fig. 3. Steal-on-abort in action. Worker thread A is executing a transaction based on Job 2, and worker thread B is executing a transaction based on Job 6. In step 1, thread A's transaction conflicts with, and aborts, Thread B's transaction. In step 2, thread A abort-steals thread B's job, and places it in its own steal queue. In step 3, after thread A finishes stealing, thread B gets a new job, and starts executing it.

submit transactional jobs to the thread pool to be executed asynchronously, rather than executing transactions directly. This requires a trivial change to the application code.

Secondly, application threads that previously executed a transactional code block, and then executed code that depended on the transactional code block (e.g. code that uses a return value obtained from executing the transactional code block), are not easily supported using asynchronous job execution. This dependency can be accommodated by using synchronous job execution; for example, the application thread could wait on a condition variable, and be notified when the submitted transactional job has committed. Additionally, the transactional job object could be used to store any return values from the committed transaction that may be required by the application thread's dependent code. This requires a simple modification to the implementation described already. The use of asynchronous job execution, where possible, is preferred as it permits greater parallelism: application and worker threads execute simultaneously.

2.4 Steal-on-abort Strategies

Four different steal-on-abort strategies constitute the design space investigated in this paper. Each strategy differs in the way stolen transactions are released. The properties of each one are explained below.

Steal-Tail inserts abort-stolen jobs at the tail of the `mainDeque` once the current transaction completes. This means the abort-stolen jobs will be executed last since jobs are normally taken from the head of the deque, unless other threads work-steal the jobs and execute them earlier. As mentioned earlier, jobs are created and distributed in a round-robin manner to threads' `mainDeque`s. Therefore, jobs created close in time will likely be executed close in time. Steal-Tail may benefit performance in a benchmark, for example, where a job's creation time has strong affinity with its data accesses, i.e. jobs created close in time have similar data accesses, which means they are likely to conflict if executed concurrently. Executing an abort-stolen job immediately after the current job may lead it to conflict with other concurrently executing transactions since they are likely to be those created close in time as well. Placing abort-stolen jobs at the tail of the deque may reduce conflicts by increasing the temporal execution distance between jobs created close in time.

Steal-Head inserts abort-stolen jobs at the head of the `mainDeque` once the current transaction completes. This means the abort-stolen jobs will be executed first. For benchmarks that do not show the affinity described above, placing jobs at the head of the deque may take advantage of cache locality and improve performance. For example, transaction A aborts and abort-steals transaction B. Upon completion of transaction A, transaction B is started. At least one data element is common to both transactions; the data element that caused a conflict between them, and is likely to be in the local cache of the processor (or core). The larger the data access overlap, the more likely performance is to improve.

Steal-Keep does not move abort-stolen jobs from a thread's `stealDeque` to its `mainDeque` once the current transaction completes. The thread continues to execute jobs from its `mainDeque` until it is empty, and then executes jobs from its `stealDeque` (when both are empty, work stealing is invoked as usual). The motivation of Steal-Keep is to increase the average time to an abort-stolen job's re-execution, as it will be executed last by the current thread, and only work-stolen by other threads if all other threads' `mainDeque`s are empty. Steal-Keep may reduce steal-on-abort overhead as it does not require jobs to be moved from the `stealDeque` to the `mainDeque` after every transaction finishes, however, it may increase the overhead of work stealing when the `mainDeque` of all threads is empty.

Steal-Block causes an abort-stolen job's second-order abort-stolen jobs to be taken as well (thus a block of transactions is stolen). The hypothesis is that in some benchmarks there is a strong data access affinity between aborted transactions that extends further down the directed graph of aborted transactions. In such benchmarks, Steal-Block aims to give greater performance improvements by reordering transactions faster. However, it also increases steal-on-abort overhead, as on every steal-on-abort operation the `stealDeque` must be traversed to take the second-order abort-stolen transactions.

2.5 Limitations

There are two important limitations to steal-on-abort. First, steal-on-abort is only useful when repeat conflicts occur, as queueing transactions eliminates the chance of repeat conflicts. If an application has significant numbers of conflicts, but they are mostly unique conflicts, then the benefit of steal-on-abort may be reduced.

Second, in order to detect repeat conflicts, the TM implementation must support visible accesses, either read, write, or both. Using invisible reads and writes only allow conflicts to be detected between an active transaction and a committed transaction. Repeat conflicts require the detection of conflicts between two active transactions, as then one may abort, restart, and again conflict with the same opponent, if the opponent is still active.

2.6 Related Work

Research in transaction reordering for improving TM performance has been limited. Bai *et al.* [10] introduced a key-based approach to co-locate transactions based on their calculated keys. Transactions that have similar keys are predicted to have a high likelihood of conflicting, and queued in the same queue to be executed serially. Their implementation also uses a thread pool model with multiple work queues, but they do not support work-stealing or abort-stealing.

Although their approach improves performance, its main limitation is the requirement of an application-specific formula to calculate the keys. This makes their technique of limited use without application-specific knowledge, and performance is dependent on the effectiveness of the formula. For some applications it may be difficult to create effective formulae, and in the extreme case ineffective formula may degrade performance. In contrast, steal-on-abort does not require any application-specific information.

Our recent work [11] attempts to reduce aborts by monitoring the percentage of transactions that commit over a period of time. If the percentage is found to deviate from a specified threshold then worker threads are added or removed from a transactional thread pool to increase or decrease the number of transactions executed concurrently. Although this work does not target repeat conflicts, it effectively schedules transactions to improve resource usage and execution performance.

Recent work by Dolev *et al.* [12], called CAR-STM, has similarly attempted to schedule transactions into queues based on repeat conflicts. CAR-STM also supports the approach by Bai *et al.* to allow an application to submit a function used to co-located transactions predicted to have a high likelihood of conflicting. Unlike steal-on-abort, CAR-STM does not support the use of existing contention management policies, does not implement work stealing to improve load balance and parallel performance, and does not investigate strategies such as those in Section 2.4.

Harris *et al.* [13] describe the retry mechanism, which allows an aborted transaction to block, and wait for the condition that caused it to abort to change, rather than restart immediately. However, retry must by explicitly called by the programmer, whereas steal-on-abort operates transparently.

3 Evaluation

The evaluation aims to investigate the performance benefits of the steal-on-abort strategies by executing several benchmarks using high contention configurations. In this section, the term *Normal* refers to execution without steal-on-abort. Steal-Tail, Steal-Head, Steal-Keep, and Steal-Block are abbreviated to Steal-T, Steal-H, Steal-K, and Steal-Blk, respectively. All execution schemes (including Normal) utilise the thread pool, and work stealing.

3.1 Platform

The platform used to execute benchmarks is a 4 x dual-core (8-core) Opteron 880 2.4GHz system with 16GB RAM, running openSUSE 10.1, and using Sun Hotspot Java VM 1.6 64-bit with the flags `-Xms4096m -Xmx14000m`. Benchmarks are executed using DSTM2 set to using the shadow factory, and visible reads. DSTM2 always uses visible writes. Although read and write visibility affect the amount of conflicts that occur, visible reads and writes are generally considered to give higher performance than invisible reads and writes when a large number of conflicts occur, and the benchmarks in this evaluation have large numbers of conflicts (see next). Experiments are executed with 1, 2, 4, and 8 threads, each run is repeated 9 times, and mean results are reported with ±1 standard deviation error bars.

The published best contention manager (CM), called Polka [14], is used in the evaluation. Upon conflict, Polka waits exponentially increasing amounts of time for a dynamic number of iterations (equal to the difference in the number of read accesses between the two transactions) for the opponent transaction to commit, before aborting it. Polka's default parameters are used for controlling the exponentially increasing wait times [14].

3.2 Benchmarks

The benchmarks used to evaluate steal-on-abort are a sorted linked list, red-black tree, STAMP-vacation, and Lee-TM. Hereafter, they are referred to as List, RBTree, Vacation, and Lee-TM, respectively. Evaluating steal-on-abort requires the benchmarks to generate large amounts of transactional conflicts. Below, the benchmarks are briefly described, along with the execution parameters used to produce high contention.

List and RBTree transactionally insert or remove random numbers into a sorted linked list or tree, respectively. List and RBTree are configured to perform 20,000 randomly selected insert and delete transactions with equal probability. Additionally, after executing its code block, each transaction waits for a short delay, which is randomly selected using a Gaussian distribution with a mean duration of 3.2ms, and a standard deviation of 1.0. The delays are used to simulate transactions that perform extra computation while accessing the data structures. This also increases the number of repeat conflicts.

Vacation is a benchmark from the STAMP suite (version 0.9.5) ported to DSTM2. It simulates a travel booking database with three tables to hold bookings for flights, hotels, and cars. Each transaction simulates a customer making several bookings, and thus several modifications to the database. High contention is achieved by configuring Vacation to build a database of 128 relations per table, and execute 256,000 transactions, each of which performs 50 modifications to the database.

Lee-TM is a transactional circuit routing application. Pairs of coordinates for each route are loaded from a file and sorted by ascending length. Each transaction attempts to lay a route from its start coordinate to its end coordinate in a three-dimensional array that represents a layered circuit board. Routing consists of two phases: *expansion* performs a breadth-first search from the start coordinate looking for the end coordinate, and *backtracking* writes the route by tracing back a path from the end coordinate to the start coordinate. For high contention, the Lee-TM-t configuration [5] is used (i.e., no early release) with the `mainboard.txt` input file, which has 1506 routes. This input file has relatively long transactions, and a only minority of them cause contention so repeat conflicts should be limited in comparison to the other benchmarks. Furthermore, later transactions are more likely to conflict with each other because of large amounts of data accesses, and Steal-Blk may offer better performance in such conditions.

3.3 Transaction Throughput

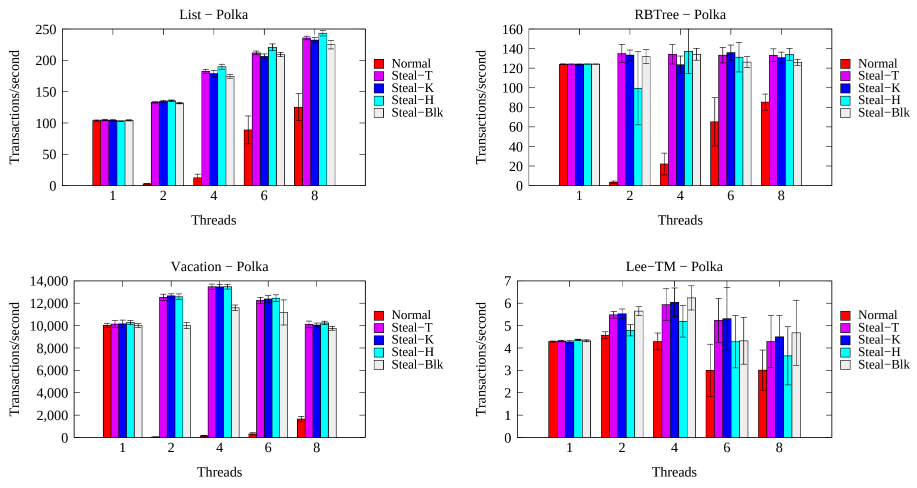

Fig. 4. Throughput results.

Figure 4 illustrates the throughput results. Cursory observation shows that steal-on-abort always improves throughput over Normal execution, sometimes by significant margins. Performance variance is generally minimal between the steal-on-abort strategies compared to the difference with Normal, but Steal-Blk is less effective in Vacation, and slightly more effective in Lee-TM, while Steal-H is less effective in Lee-TM. Furthermore, Steal-K and Steal-T are the most consistent performers, and thus for brevity the discussion will mainly focus on the performance benefits of Steal-T.

In List, Steal-T improves average throughput over Normal by 46.7 fold with 2 threads, 14.6 fold with 4 threads, 2.4 fold with 6 threads, and 1.9 fold with 8 threads. Similarly, in RBTree the improvements are 40.0 fold, 6.1 fold, 2.0 fold, and 1.6 fold respectively. In Vacation the improvements are 290.4 fold, 92.9 fold, 37.9 fold, and 6.1 fold, respectively.

Examining Lee-TM, Steal-T improves average throughput over Normal by 1.2 fold with 2 threads, 1.4 fold with 4 threads, and 1.3 fold with 8 threads. However, Lee-TM results have high standard deviations, which increase with the number of threads. This is caused by Lee-TM performance being sensitive to the order in which transactions commit. As there are only 1506 routes, and most of the contention due to the long transactions executed near the end, even aborting a few long transactions in favour of other long transactions that have performed less computation can significantly impact performance. As predicted, Steal-Blk generally improves performance the most for Lee-TM.

3.4 Repeat Conflicts

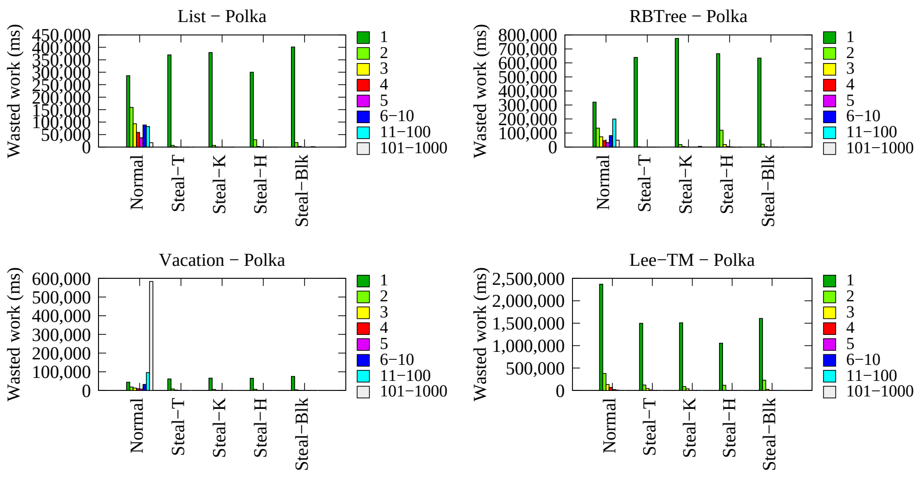

Fig. 5. Wasted work distribution by number of repeat conflicts.

Next, we examine the amount of time spent in repeat conflicts, and the effectiveness of the steal-on-abort strategies at reducing repeat conflicts. Figure 5 shows histograms of the distribution of wasted work [15] (i.e. the amount of time spent executing transactions that eventually aborted) for a given number of conflicts with the same transaction. As an example, consider a transaction A that is aborted seven times before it finally commits. Such a transaction has seven lots of wasted work. Four aborts occur through conflict with a transaction B, two with a transaction C, and one with a transaction D (seven in total). The four lots of wasted work caused by conflicting with, and being aborted by, transaction B are added to column '4', the two lots associated with C are added to column '2', and the one lot associated with D is added to column '1'. For brevity, only results from execution with eight thread results are discussed, although better performance improvements were observed previously with fewer threads (Figure 4).

Since steal-on-abort should targets repeat conflicts it should reduce the amount of time in all but the first column. This is confirmed by the results in Figure 5: Steal-T reduces time in the remaining columns (repeat conflicts) by 99% in List, 95% in RBTree, 99% in Vacation, and 58% in Lee-TM. Furthermore, the results show that repeat conflicts represent a significant proportion of the total wasted work for the high contention configurations used: 65% in List, 54% in RBTree, 96% in Vacation, and 17% in Lee-TM. The net reduction in wasted work using Steal-T with 8 threads is 53% in List, 18% in RBTree, 93% in Vacation, and 13% in Lee-TM.

However, steal-on-abort increases single conflict (non-repeat) wasted work for List, RBTree, and Vacation. This is because repeat conflicts are being reduced to single conflicts so their wasted work is allocated to the single conflict column. However, the increase in single conflict wasted work is far less than the decrease in repeat conflict wasted work. As a result, Lee-TM, which has far fewer repeat conflicts than the other benchmarks, actually sees a fall in single conflict wasted work. Thus, a side effect of steal-on-abort is to reduce the number of single (i.e., unique) conflicts that occur.

3.5 Committed Transaction Durations

Polka causes transactions to wait for their opponents, which increases the average time it takes to execute a transaction that eventually commits. Since steal-on-abort reduced the amount of time spent in repeat conflicts, it should also have reduced the total number of conflicts, which in turn should have reduced the average committed transaction's duration.

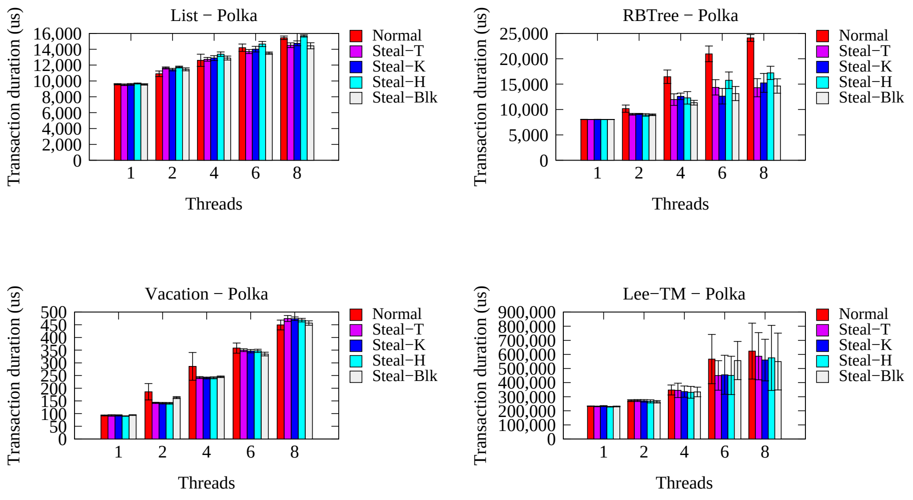

Fig. 6. Average Committed Transaction Duration (microseconds).

Figure 6 shows the results for the average committed transaction's duration, which includes the overhead of steal-on-abort operations, and confirms the hypothesis. Three of the four benchmarks reduce the average duration with 8 threads, except for List using Steal-H, which marginally increases the average duration. Only for Vacation do all the steal-on-abort strategies increase the average duration, although this is still largely within the standard deviations.

3.6 Steal-on-abort Overhead

We have not precisely measured the overhead of steal-on-abort as it consists of small code blocks, some of which execute within transactions, and some outside of transactions. However, as shown in Figure 6, Vacation's transactions have much shorter average durations than the other benchmarks, and consequently Vacation's increase in average duration in Figure 6 may be due to abort-stealing overhead, which would indicate that the overhead is in the tens of microseconds per transaction.

However, this overhead does not include the cost of moving transactions between deques, as that happens after a transaction completes. To measure that cost the in-transaction metric (InTx), which is the proportion of execution time spent in executing transactions, is presented in Figure 7. For the benchmarks used in this evaluation there are two sources of out-of-transaction execution: work stealing, and moving jobs from a thread's `stolenDeque` to its `mainDeque` after every transaction completes. Since Normal execution utilises work stealing, the difference between Normal and steal-on-abort execution should approximately represent the cost of moving jobs between the deques.

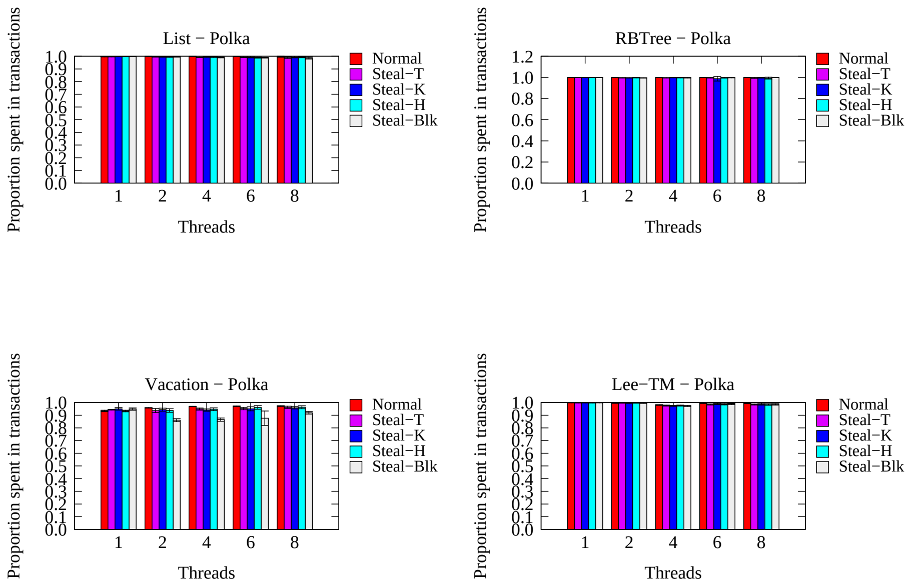

Fig. 7. Proportion of total time spent executing transactions.

Figure 7 identifies that there is negligible overhead in moving jobs between deques and work stealing in List, RBTree, and Lee-TM. However, in Vacation the overhead becomes visible, with most strategies observing an overhead of 3%. The average execution time of Vacation at 8 threads with Steal-T is 24.0 seconds, and given that 256,000 transactions are executed, the average overhead of moving jobs is 2.8 microseconds per transaction. However, this cost is related to the number of jobs moved between deques, and with Steal-T this averages to 2.2 jobs per completed transaction. Section 2.4 mention that Steal-Blk may have higher overhead, and Vacation's results identify Steal-Blk observing an overhead of 5-10%.

4 Conclusions and Future Work

In well-engineered, scalable, concurrently programmed applications it is expected that high contention conditions will occur only rarely. Nevertheless, when high contention does occur it is important that performance degrades as little as possible. It is also probable that some applications will not be as well-engineered as expected, and thus may suffer from high contention more frequently.

This paper presented steal-on-abort, a new runtime approach that dynamically reorders transactions with the aim of improving performance by reducing the number of repeat conflicts. Steal-on-abort is a low overhead technique that requires no application specific information or offline pre-processing.

Steal-on-abort was evaluated using the well-known Polka contention manager with two widely used benchmarks in TM: sorted linked list and red-black tree, and two non-trivial benchmarks: STAMP-vacation and Lee-TM. The benchmarks were configured to generate high contention, which led to significant amounts of repeat conflicts. Steal-on-abort was effective at reducing repeat conflicts: Steal-Tail reducing by almost 60% even when repeat conflicts only accounted for 17% of the total wasted work, and reducing by over 95% when repeat conflicts accounted for 55% or more of the wasted work. This led to performance improvements ranging from 1.2 fold to 290.4 fold.

We are encouraged by the results from the steal-on-abort evaluation, and we plan to continue our investigation of the design space. In particular, we wish to investigate the design of steal-on-abort when invisible reads and writes are used, and the implementation of steal-on-abort for HTMs.

References

1. Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 253–262. ACM Press, October 2006.
2. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101. ACM Press, July 2003.

3. 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 *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 69–80. ACM Press, June 2007.
4. Ian Watson, Chris Kirkham, and Mikel Luján. A study of a transactional parallel routing algorithm. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 388–400. IEEE Computer Society Press, September 2007.
5. Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Lee-TM: A non-trivial benchmark for transactional memory. In *ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing*. LNCS, Springer, June 2008.
6. Virendra Marathe, Michael Spear, Christopher Herio, Athul Acharya, David Eisenstat, William Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. In *TRANSACT '06: First ACM SIGPLAN Workshop on Transactional Computing*, June 2006.
7. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*. LNCS, Springer, September 2006.
8. Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246. ACM Press, February 2008.
9. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.
10. T. Bai, X. Shen, C. Zhang, W.N. Scherer, C. Ding, and M.L. Scott. A key-based adaptive transactional memory executor. In *IPDPS '07: Proceedings of the 21st International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, March 2007.
11. Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *EUROPAR '08: Fourteenth European Conference on Parallel Processing*, August 2008.
12. Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: Scheduling-based collision avoidance and resolution for software transactional memory. In *PODC '07: Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, August 2008.
13. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.
14. William Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 240–248. ACM Press, July 2005.
15. Cristian Perfumo, Nehir Sonmez, Adrian Cristal, Osman Unsal, Mateo Valero, and Tim Harris. Dissecting transactional executions in Haskell. In *TRANSACT '07: Second ACM SIGPLAN Workshop on Transactional Computing*, August 2007.