Steal-on-abort: Dynamic Transaction Reordering to Reduce Conflicts in Transactional Memory

Mohammad Ansari
University of Manchester
ansari@cs.man.ac.uk

Mikel Luján
University of Manchester
mikel@cs.man.ac.uk

Christos Kotselidis
University of Manchester
kotselidis@cs.man.ac.uk

Kim Jarvis
University of Manchester
jarvis@cs.man.ac.uk

Chris Kirkham
University of Manchester
chris@cs.man.ac.uk

Ian Watson
University of Manchester
watson@cs.man.ac.uk

ABSTRACT

In Transactional Memory (TM) if any two concurrently executing transactions perform conflicting data accesses, one of them is aborted. The transaction to be aborted is usually selected by some form of contention manager. Aborted transactions waste computing resources, and reduce performance. Ideally, concurrent execution of transactions would be ordered to minimize aborts, but such an ordering is often either complex, or unfeasible, to obtain.

This paper presents introduces 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 same transaction that aborted it the first time, leading to another aborted transaction. In steal-on-abort, the aborted transaction is stolen by the non-aborted transaction, and queued behind it, thus preventing the two transactions from conflicting again.

Steal-on-abort operates at runtime, and requires no application-specific information or offline pre-processing. In this paper, steal-on-abort is considered for eager-validation TM systems, and evaluated using a sorted linked list, red-black tree, and STAMP-vacation with different contention managers. The evaluation reveals a range of improvements in throughput performance, and in the transactional metrics *wasted work*, and *aborts-per-commit* (APC).

1. INTRODUCTION

Recent progress in multi-core processor architectures, coupled with challenges in advancing uniprocessor designs, has led to mainstream processor manufacturers adopting multi-core designs. Modest projections suggest hundred-core processors to be common within a decade. Although multi-core has re-invigorated the processor manufacturing industry, it has led to an important change in software development.

The execution time of software has improved on successive generations of uniprocessors. However, on future multi-core processors this 'free' improvement will not materialize unless the software is multi-threaded, i.e., parallelized, and thus able to take advantage of the increasing number of cores. Furthermore, given the number of cores predicted in future processors, software will need to be parallelized to non-trivial levels.

Parallel (or 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. Parallel applications also usually take longer to build, and correcting defects is complicated by the difficulty in reproducing errors. However, the move to multi-cores requires adoption of parallel programming by the majority of programmers, not just experts, and thus simplifying it has become an important challenge.

Transactional Memory (TM) is a new parallel 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 parallel programming. In TM, programmers are required to 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., invisibly to the programmer) by a TM system. The TM system compares each transaction's data accesses against all other transactions for conflicts, also known as *conflict detection* or *validation*. 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.

In order to achieve good scalability on multi-core architectures, it is important that the number of aborted transactions is kept to a minimum. Aborted transactions reduce performance, reduce scalability, and waste computing re-

sources. Furthermore, certain (update-in-place) TM implementations require extra computing resources to roll back the program to a consistent state. The order in which transactions are executed concurrently can affect the number of aborts that occur, and given complete information a priori it may be possible to determine an optimal order (or schedule) that minimizes the number of aborts. However, in practice this is difficult to achieve because complete information is not available for many programs, e.g., due to dynamic transaction creation, or impractical to obtain. Additionally, even if complete information is available, the search space for computing the optimal order of transactions is likely to be infeasibly large.

This paper presents an initial design space exploration of a novel 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 the 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, but instead 'stolen' by the non-aborted transaction, and queued behind it, thus preventing the two transactions from conflicting again. Steal-on-abort requires no application-specific information or offline pre-processing, and it is considered for eager validation TM systems in this paper.

Steal-on-abort is implemented in DSTM2 [1], a Software TM (STM) implementation, that has been modified to employ random *work stealing* [2] to execute transactions. Steal-on-abort is evaluated with different contention managers using two widely used benchmarks in TM (sorted linked list [3], and red-black tree [3]), and a non-trivial benchmark (STAMP-vacation [4]). The evaluation reveals hundred-fold performance improvements for some contention managers, while negligible performance difference for others.

The remainder of this paper is organized as follows: Section 2 introduces steal-on-abort, the strategies developed, its implementation in DSTM2, and related work. Section 3 evaluates steal-on-abort, presenting results for execution time, and transactional execution metrics [5] *wasted work*, and *aborts-per-commit* (APC). Finally, Section 4 completes the paper with a summary and overview of future work.

2. STEAL-ON-ABORT

In all TM implementations, a data access conflict between two transactions requires conflict resolution to abort one of them. In most TM implementations, the aborted transaction is immediately restarted. However, we observed that the restarted transaction often conflicts with the same transaction again, and gets aborted again, 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 is avoided).

Steal-on-abort does not restart the aborted transaction immediately; it 'gives' the aborted transaction to the opponent transaction, and a new transaction is started in place

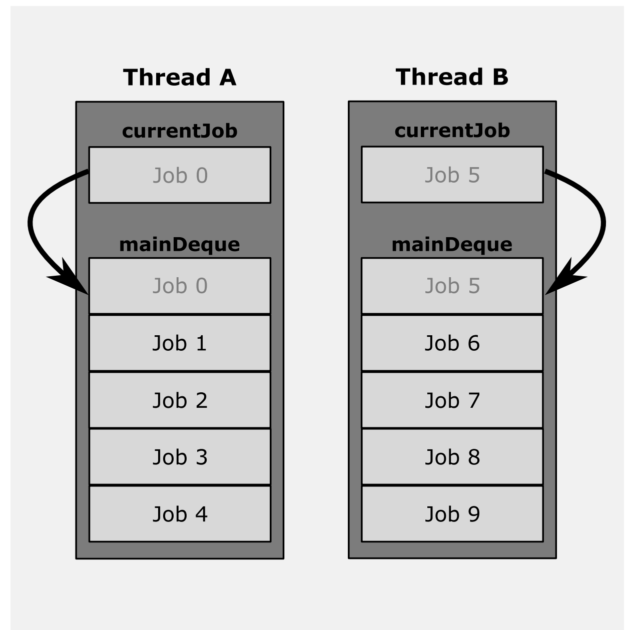

Figure 1: DSTM2 is modified to implement per-thread deques that store transactional jobs. Threads take jobs from the head of their own deque.

of the aborted transaction. This strategy aims to reduce the amount of temporally local aborts, and similar to a greedy search algorithm, expects that minimizing temporally local aborts will minimize the global number of aborts, and thus improve execution performance and efficiency of TM.

Steal-on-abort relies on removing repeat conflicts between pairs of transactions to improve performance. A repeat conflict occurs when an aborted transaction conflicts for a second time with a transaction that is still active. The more repeat conflicts that occur in an application, the more effective steal-on-abort is likely to be. In applications that have a high number of unique aborts, i.e., few of them are repeat conflicts, steal-on-abort may not improve results.

Finally, steal-on-abort reorders transactional jobs, and depending on the strategy (for examples, see below), the reordering may be significant. It is beyond the scope of this paper to provide a complete analysis of the impact of this reordering on fairness and responsiveness.

The remainder of this section details the concrete implementation of random work stealing, and steal-on-abort in DSTM2, and then goes on to explain the two design variants of steal-on-abort evaluated in this paper.

2.1 Implementation in DSTM2

DSTM2, like most other STM implementations [6, 7, 8], creates a number of threads that concurrently execute transactions. In order to implement steal-on-abort, we implemented random work stealing [2] in DSTM2, because once a thread's transaction is stolen (on abort), it must obtain another transaction to execute.

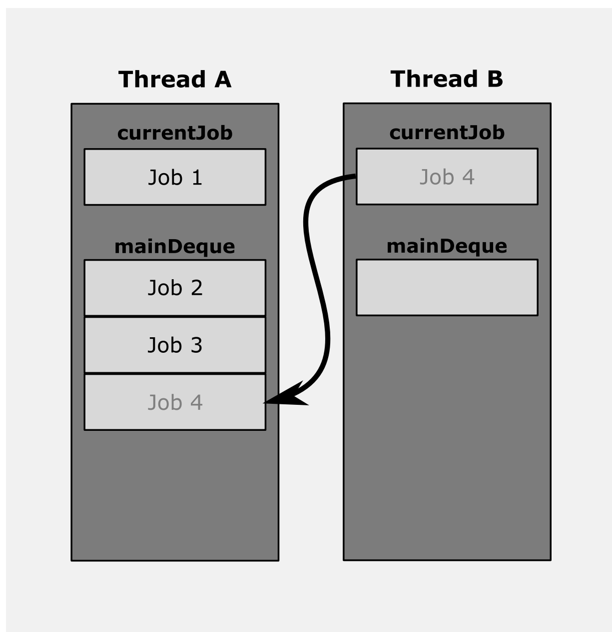

Figure 2: If a thread deque is empty, it steals work from the tail of another, randomly selected, thread's deque.

2.1.1 Adding Random Work Stealing to DSTM2

A `java.util.concurrent.LinkedBlockingDeque`, which is a thread-safe double-ended queue (deque), was added to each thread, and named `mainDeque`, as shown in Figure 1. The deque allows LIFO and FIFO operations, and is used to store transactional jobs. A transactional job is simply an object that holds parameters needed to execute a transaction. In the benchmarks used in this paper, these parameters were generated randomly. Threads execute transactions by removing a job from the head of their deque, placing it in the thread variable `currentJob` and executing the transactional code block using the parameters stored in the job. The benchmarks used in this paper were modified to load jobs onto each thread's `mainDeque` in a round-robin manner during benchmark initialization, which is excluded from the execution times reported in the evaluation in Section 3.

Figure 2 illustrates work stealing in action. Work stealing is implemented with the method `workSteal()`, which first attempts to retrieve a job from the head of the thread's own deque, and if that is empty, attempts to steal a job from the tail of another randomly selected thread's deque. In any one call to `workSteal()` the threads from which theft has already been attempted are recorded so that random selection is performed only over the remaining threads. If a job is returned by the call, then it is stored in the thread variable `currentJob`, and not in any deque.

2.1.2 Steal-on-abort Operation

To implement steal-on-abort, a second private deque, named `stolenDeque`, is added to each thread to hold jobs stolen by the transaction currently executing on a thread. Once a transaction completes executing (i.e., commits or aborts), the jobs in the `stolenDeque` are moved to the `mainDeque`. The second deque is necessary to hide stolen jobs otherwise they may be taken and executed concurrently by other threads that have no jobs, while the current transaction is still active, thus re-introducing the possibility of a repeat conflict.

Figure 3 illustrates steal-on-abort in action. Steal-on-abort is explained from the perspectives of the victim thread (the one from which the aborted transaction is stolen) and the stealing thread. Each thread has an additional flag, called `stolen`. If a victim thread detects its transaction has been aborted, it waits for its `stolen` flag to be set, and then calls `workSteal()` to obtain a new job if its `mainDeque` is empty, storing it in `currentJob`, and then clears the `stolen` flag. The victim thread must wait on the `stolen` flag, otherwise access to the variable `currentJob` would be unsafe.

The stealing thread operates as follows. In DSTM2, a transaction is aborted by using Compare-And-Swap (CAS) to change its status flag from `ACTIVE` to `ABORTED`. If the stealing thread's call to abort the victim thread's transaction in this manner is successful, it proceeds to steal the victim thread's job that is stored in its `currentJob` variable. After the job is taken, the victim thread's `stolen` flag is set.

2.2 Steal-on-abort Strategies

Two steal-on-abort strategies that differ in when they choose to re-execute a stolen job are described and evaluated. When an aborted job is stolen, and subsequently moved from the `stolenDeque` to the `mainDeque`, it can either be placed at the head of the `mainDeque`, or the tail.

Steal-Tail. If jobs are placed at the tail, the thread will execute the stolen jobs last, although the job may be executed earlier by another thread due to work stealing. As an example, the round-robin allocation of jobs means jobs that were created close in time will likely be executed close in time. For benchmarks with a close relationship between a job's creation time and its data accesses, executing a stolen job right after the current job may lead to conflicts with other transactions, therefore placing stolen jobs at the tail of the deque may reduce conflicts.

Steal-Head. If jobs are placed at the head, the thread will execute the stolen jobs first. For benchmarks that do not show the temporal locality described above, placing jobs at the head of the deque may take advantage of cache locality to improve performance. For example, data accessed by transaction A, which aborts and steals transaction B's job, is likely to have at least one data element (the data element that caused a conflict between the two transactions), in the processor's local cache.

2.3 Related Work

Limited research has been carried out in transaction reordering for improving TM performance. Bai *et al.* [9] introduced a key-based approach that colocates transactions based on their calculated keys. Although their approach improves performance, it requires an application-specific formula to calculate keys for transactions. Furthemore, per-

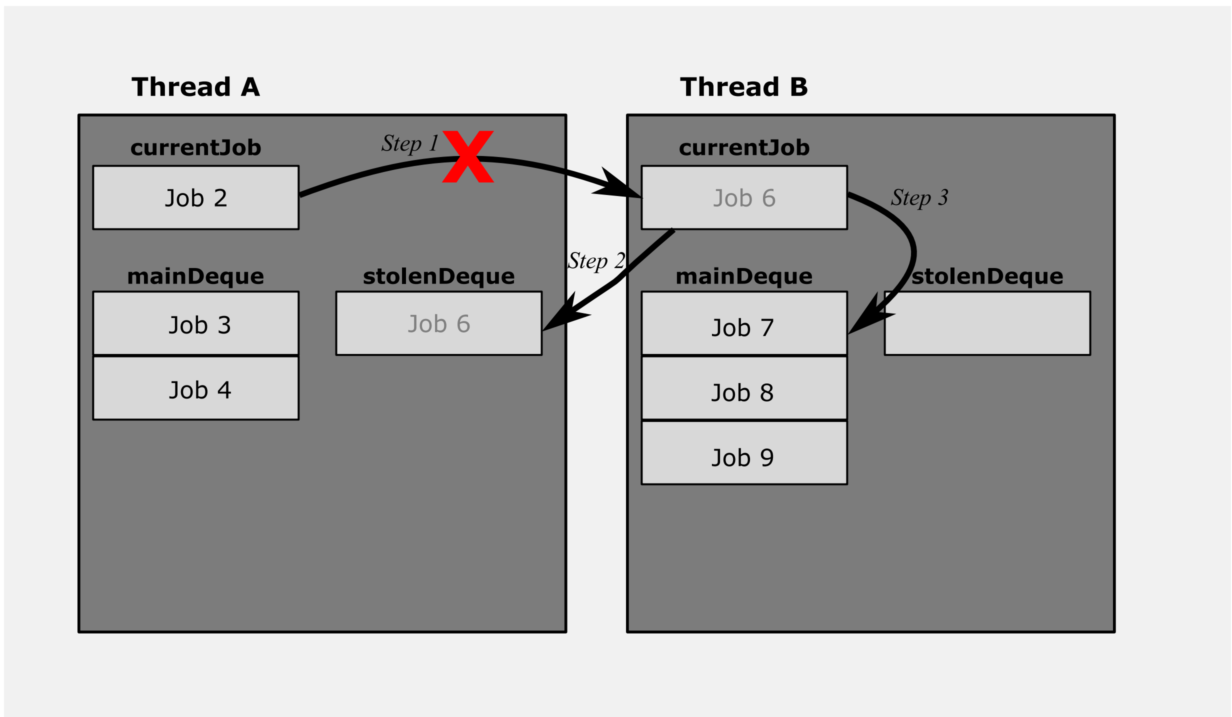

Figure 3: Steal-on-abort in action. Thread A is executing a transaction based on Job 2, and Thread B is executing a transaction based on Job 6. In step 1, thread A's transaction conflicts with, and aborts, Thread B's transaction. In step 2, thread A steals thread B's job, and places it in its own `stolenDeque`.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

formance is based on the effectiveness of the formula, and it may be difficult to generate such formulas for some applications. In contrast, our approach does not require any application-specific information.

3. EVALUATION

The experiments will test whether steal-on-abort provides performance improvements, and whether it reduces the wasted work and APC. In this section, Normal execution refers to execution without steal-on-abort, Steal-Head refers to steal-on-abort execution where stolen jobs are moved to the head of the `mainDeque`, and Steal-Tail refers to execution where stolen jobs are moved to the tail. All execution schemes utilize random work stealing as explained previously.

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 eager validation. Benchmarks are executed with 1, 2, 4, and 8 threads, each run is repeated 6 times. Mean results are reported with ± 1 standard deviation error bars.

3.2 Benchmarks

The benchmarks used to evaluate steal-on-abort are linked list [3], red-black tree [3], and STAMP-vacation [4]. Hereafter, they are referred to as List, RBTree, and Vacation, respectively. List and RBTree microbenchmarks transactionally insert or remove random numbers into a sorted linked list or tree, respectively. Vacation is a benchmark from the STAMP suite (version 0.9.5) ported to DSTM2 that 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. The number of threads used represents the number of concurrent customers.

Evaluating steal-on-abort requires the benchmarks to generate large amounts of transactional conflicts. Below, their execution configurations to produce high contention scenarios are described.

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 standard deviation of 1.0, and a mean duration of 3.2ms. The execution time of the average committed transaction in List is 4ms, and in RBTree is 0.2ms, before the delays were added. The delays are used to simulate transactions that perform extra computation while accessing the data structures. This also increases the number of repeat conflicts.

To induce high contention in Vacation, it is configured to build a database of 128 relations per table, and execute 1,024,768 transactions, each of which performs 50 modifications to the database.

3.3 Contention Managers

Aggressive [3], Polka [10], and Priority contention managers (CMs) are used to provide coverage of published CM policies. Aggressive always aborts the opponent transaction. Polka gives the opponent transaction time to commit before aborting it. Polka waits exponentially increasing amounts of time for a dynamic number of iterations (equal to the difference in the number of read accesses performed by the two transactions). The parameters for Polka are based on the defaults [10]. Priority immediately aborts the younger of the two transactions based on their timestamps. Without steal-on-abort, Priority gives the best performance, followed by Polka, and finally Aggressive gives the poorest performance with the high contention configurations used (see below).

3.4 Performance

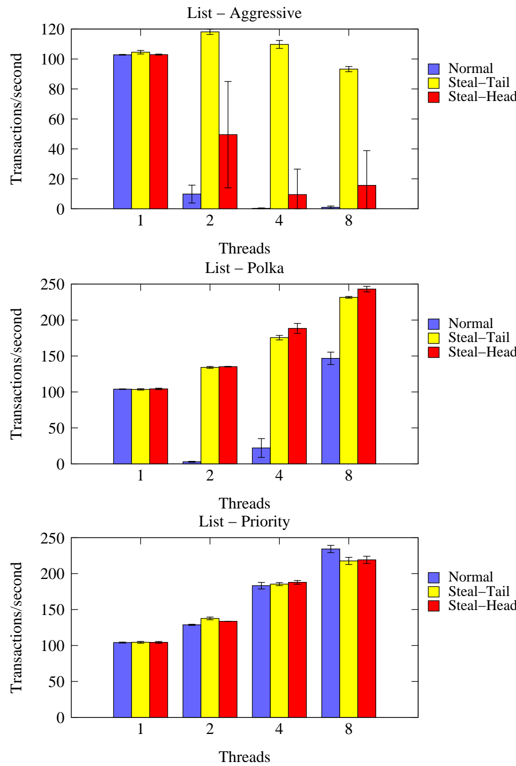

Figure 4: List throughput. Higher is better.

Figure 4 shows the transaction throughput results for List. The Aggressive CM, which gives the poorest performance benefits significantly with steal-on-abort. Steal-Tail gives a 422 times improvement at 4 threads, 99 times improvement at 8 threads, while Steal-Head gives 36 times and 16 times respectively. Using Polka the benefit is less pronounced, but still significant with both steal-on-abort strategies giving an 8 times improvement at 4 threads, and 1.6 times improvement at 8 threads. Priority, the best performing CM in these results, sees no benefit of using steal-on-abort, and performance actually degrades slightly at 8 threads.

Steal-Head improves Polka's performance such that it is close (within 3%) to the performance of Priority, the best performing CM. Steal-on-abort strategies also give Polka improving performance over single thread execution as the number of threads is increased. Furthermore, neither strategy is a clear winner, with Steal-Tail giving significantly better performance improvements using the Aggressive, and Steal-Head performing better with Polka and Priority. Finally, the large standard deviations in List with Aggressive are due to (data not shown here) Steal-Head execution either completing in either a short duration, or approximately 5 times greater duration than the short duration. This suggests that Steal-Head can be further improved to consistently execute in the shorter duration.

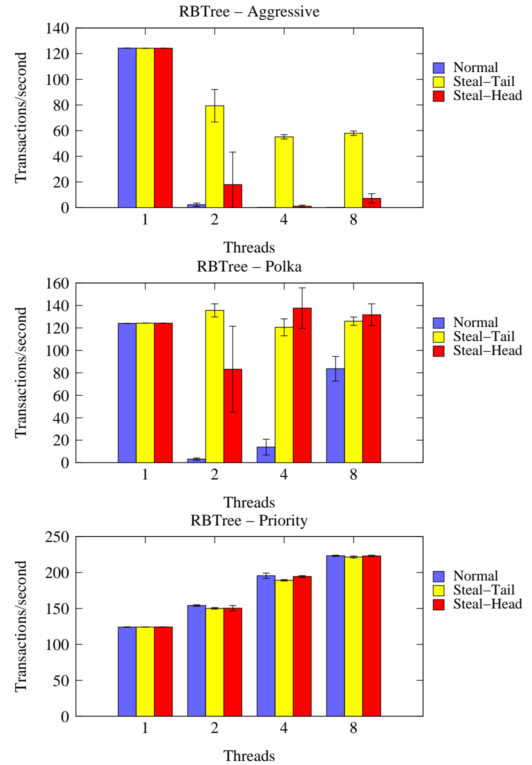

Figure 5: RBTree throughput. Higher is better.

Figure 5 shows the throughput results for RBTree. Aggressive again achieves significant performance improvements with steal-on-abort: Steal-Tail gives a 344 times improvement at 4 threads, and 1159 times improvement at 8 threads, while Steal-Head gives a 7 times and 144 times improvement respectively. Using Polka, again the benefit is less pronounced, but significant, with steal-on-abort strategies giving approximately 9 times improvement at 4 threads, and 1.6 times improvement at 8 threads. Finally, Priority again sees no benefit of using steal-on-abort.

The performance results with RBTree show many similar characteristics to those seen with List: e.g. Aggressive achieves significant improvements, Polka's improvements are near identical, and Priority sees no improvement. However, with RBTree steal-on-abort is not able to improve Polka's performance enough to match that of Priority's, and Steal-Head's large standard deviations now extend to Polka for the same reason described earlier. Finally, the performance improvements as more threads are added do not increase as seen in List.

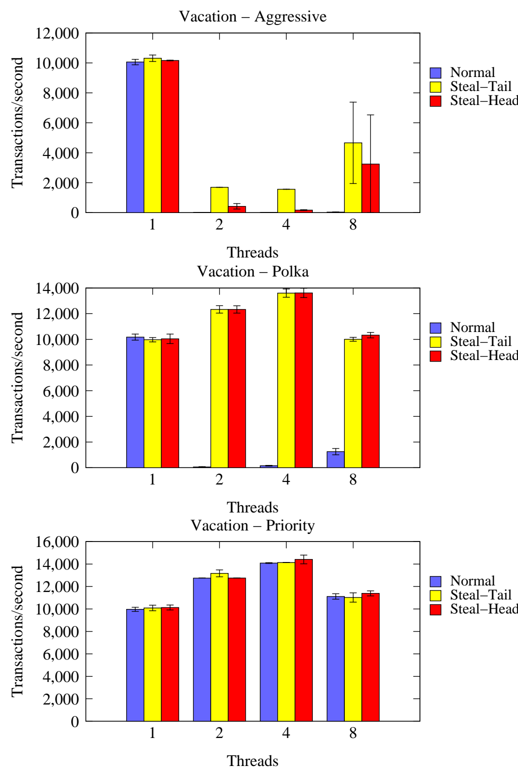

Figure 6: Vacation throughput. Higher is better.

Figure 6 shows the throughput results for Vacation. Aggressive achieves 192 times improvement with 4 threads, and 133 times improvement at 8 threads when using Steal-Tail. With Steal-Head, this falls to 21 times and 93 times respectively. Polka improves by 88 times and 8 times with both steal-on-abort strategies, respectively, and Priority does not benefit.

In this benchmark the improvements seen with Aggressive are less salient than seen in List and RBTree, and as the number of threads increases Aggressive also exhibits lower improvements than in RBTree. Conversely, Polka improves by larger margins than seen previously, and steal-on-abort again improves Polka's performance to bring it close to that of Priority, as seen in List.

Performance results have shown that Steal-Tail performs better with Aggressive, but Steal-Head performs better with Polka, and neither improve Priority's performance. The results also showed that the steal-on-abort strategies gave varying degrees of improvements depending on the application used. Steal-Head suffered from inconsistent performance improvements that lead to large standard deviations, and thus raises opportunities for further refinement.

3.5 Wasted Work

The transactional metric wasted work is the proportion of execution time spent in executing aborted transactions, and is useful in measuring the cost of aborted transactions in terms of computing resources, and it is used here to see if steal-on-abort reduces this cost. Wasted work execution times are normalized to Normal execution's wasted work time for each number of threads in each benchmark (hence

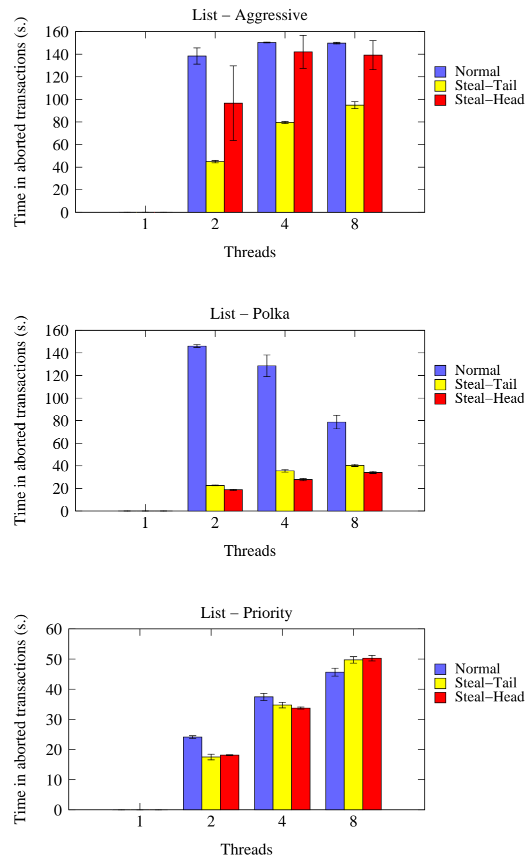

Figure 7: Wasted work in List. Lower is better.

Normal wasted work is always 1.0). No transactional aborts occur in single thread execution since there is no other concurrent transaction to conflict with, and thus single thread execution has zero wasted work.

Figures 7-9 show wasted work results. With Aggressive, in all benchmarks Steal-Tail reduces wasted work by larger margins than Steal-Head, which corresponds to the improved performance seen earlier. Steal-Tail reduces wasted work by 30% to 70% in List, 30% to 50% in RBTree, and 15% to 80% in Vacation.

With Polka, Steal-Head improves by a slightly larger margin in all benchmarks. Steal-Head reduces wasted work by 57% to 87% in List, 4% to 56% in RBTree, and 95% to 99% in Vacation. Although the average wasted work increases slightly in RBTree at 8 threads, it is within the standard deviation error, and thus a non-result.

Finally, steal-on-abort gives mixed wasted work results for Priority. In List the wasted work reduces by 25% at 2 threads, but increases by 22% in RBTree with Steal-Tail, and at 8 threads wasted work increases in List by 10% for both steal-on-abort strategies. At 4 threads the results for both List and RBTree show steal-on-abort to reduce wasted work. However, in Vacation wasted work falls by 15-30%, gradually as the number of threads increases.

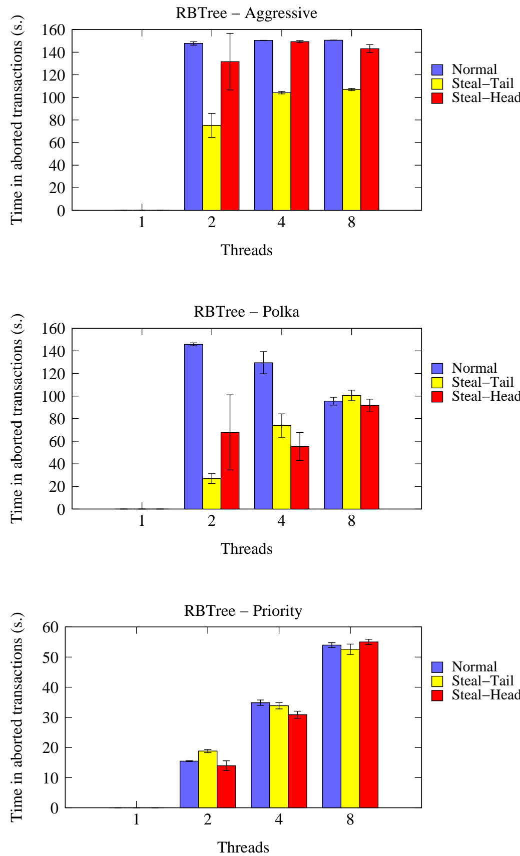

Figure 8: Wasted work in RBTree. Lower is better.

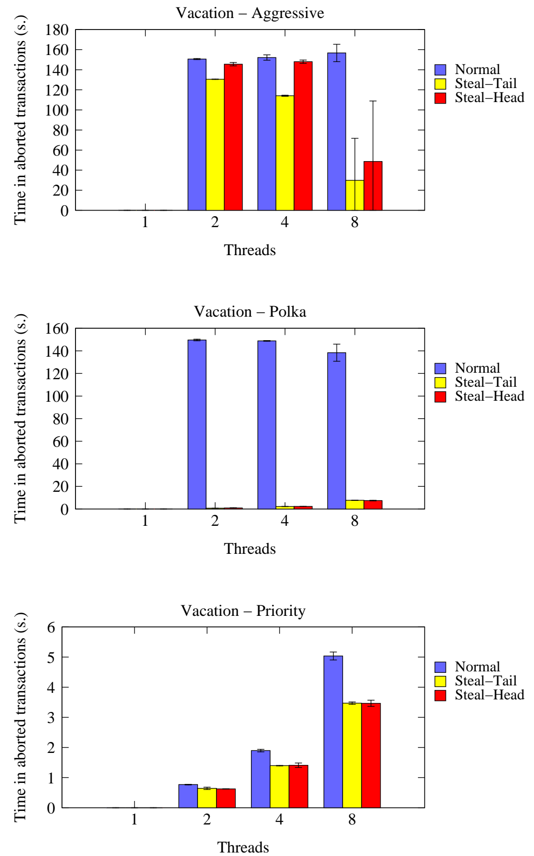

Figure 9: Wasted work in Vacation. Lower is better.

3.6 Aborts per Commit (APC)

The aborts per commit ratio (APC) is the number of aborts divided by the number of commits. It is another indicator of the efficiency with which computing resources are utilized, but it is less significant than wasted work because it ignores the execution durations of the aborted and committed transactions. We investigate it here because steal-on-abort should reduce APC in benchmarks that exhibit repeat conflicts.

Figures 10-12 present APC results. Using Aggressive, for all benchmarks at 2 threads there is little difference in APC between steal-on-abort and Normal execution. At 4 and 8 threads, however, steal-on-abort results in a negligible APC compared to Normal execution. Given the performance improvements of steal-on-abort for Aggressive, it is plausible that the high APC value is due to a large number of repeat conflicts. Also, again Steal-Tail is the better performer for Aggressive.

Execution with Polka for all benchmarks exhibits a high APC value when the number of threads is low that decreases as the number of threads increases. This may seem counter-intuitive: if increasing the number of threads leads to more conflict (due to the high contention parameters used), then APC should increase with the number of threads, not decrease. However, Polka's CM policy is to yield for a period of time before aborting a conflicting transaction. The yield period is dynamic, and related to the number of reads performed by the opposing transaction. For Normal execution in high contention situations, increasing numbers of transactions are likely to be yielding (due to conflict) when executing with increasing numbers of threads. This will lead to an increase in wasted work time, but a reduction in APC, and a reduction in performance, both of which have already been observed earlier.

Finally, steal-on-abort reduces APC for List and Vacation using Priority. For RBTree Steal-Tail always has a higher APC ratio than Normal execution, and Steal-Head has a similar or slightly lower APC ratio. Although the APC ratios are higher, they do not consistently correlate to a higher wasted work, or performance degradation.

4. SUMMARY AND FUTURE WORK

This paper has presented an evaluation of a new runtime approach, called steal-on-abort, that dynamically re-orders transactions with the aim of reducing the number of aborted transactions. Steal-on-abort requires no application specific information or offline pre-processing. Two different steal-on-abort strategies were introduced that differed in either executing stolen transactions immediately, or executing them last.

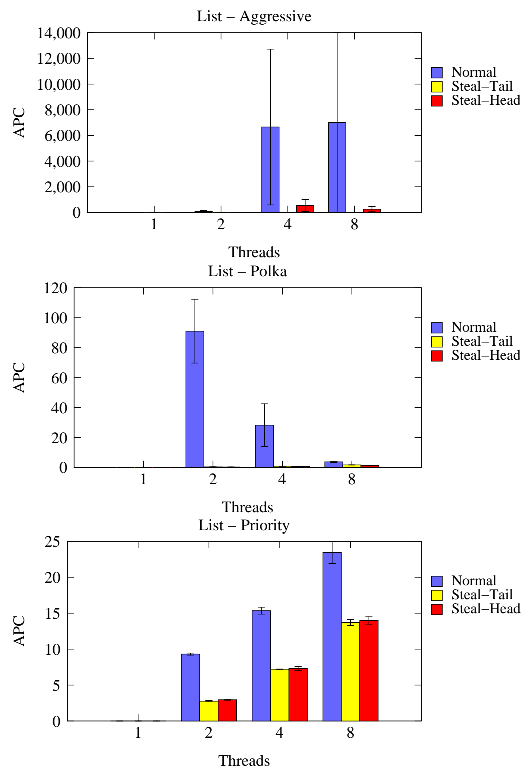

Figure 10: APC in List. Lower is better.

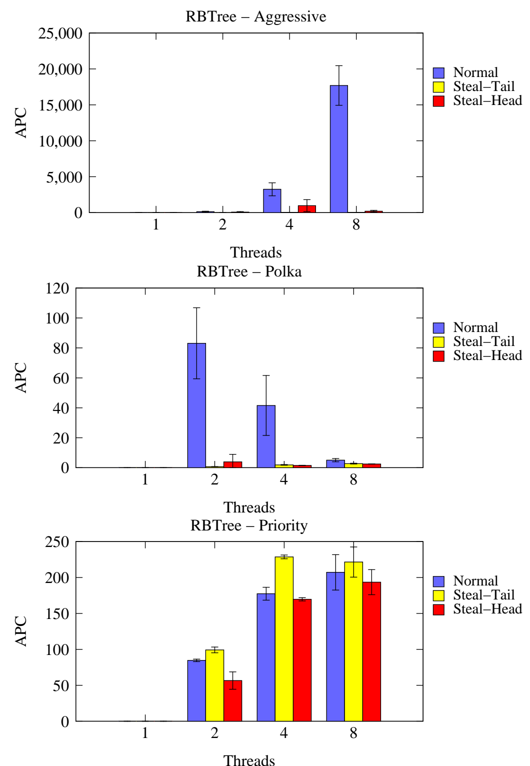

Figure 11: APC in RBTree. Lower is better.

Steal-on-abort was evaluated against two widely used benchmarks in TM, sorted linked list and red-black tree, and one non-trivial benchmark, STAMP-vacation. Using the Aggressive CM, performance results were typically improving in the range of 100 times, and significant improvements in transactional metrics was also observed. With Polka, consistent performance improvements were observed, and in two benchmarks these lead Polka to be competitive with the best CM: Priority. Finally, Priority did not benefit from steal-on-abort in terms of performance, but in several cases its transactional metrics results improved.

The steal-on-abort evaluation presented early findings, and the design space has not been fully explored, however, we have found the observed improvements encouraging, and plan to continue our investigation. For future work, we hope to implement and evaluate other more steal-on-abort strategies, as well as refine those presented. Examples of new strategies under consideration include: not moving stolen jobs in the `stolenDeque` to the `mainDeque` when the current transaction completes executing, marking certain transactions as not stealable, and stealing blocks of transactions.

5. 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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Virendra Marathe, Michael Spear, Christopher Herio, Athul Acharya, David Eisenstat, William Scherer III, and Michael 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.

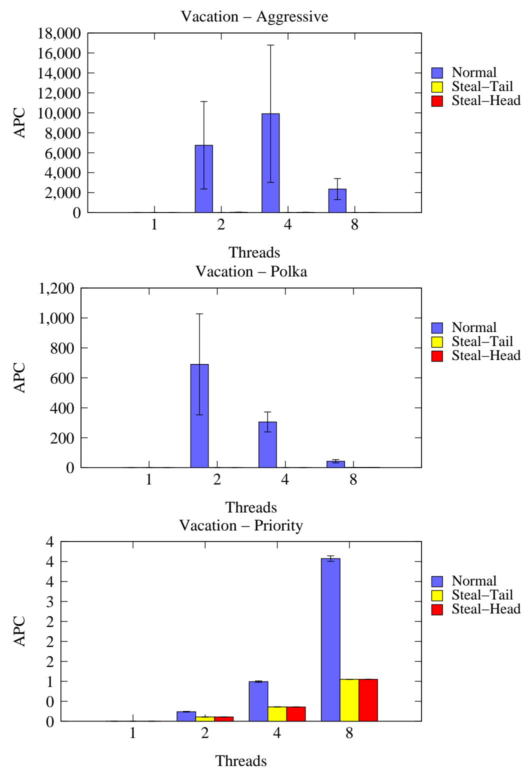

Figure 12: APC in Vacation. Lower is better.

[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] 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.

[10] William Scherer III and Michael 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.