Transaction Reordering to Reduce Aborts in Software Transactional Memory

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, conflicts between two concurrently executing transactions reduce performance, reduce scalability, and may lead to aborts, which waste computing resources. Ideally, concurrent execution of transactions would be ordered to minimise conflicts, but such an ordering is often complex, or unfeasible, to obtain. This paper identifies a pattern, called *repeat conflicts*, that can be a source of conflicts, and presents a novel technique, called *steal-on-abort*, to reduce the number of conflicts caused by repeat conflicts. Steal-on-abort operates at runtime, and requires no application-specific information or offline preprocessing. Evaluation using a sorted linked list, and STAMP-vacation with different contention managers show steal-on-abort to be highly effective at reducing repeat conflicts, which leads to a range of performance improvements.

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 raised a difficult challenge for software development.

The execution time of software has improved on successive generations of uniprocessors thanks to the increasing clock frequency, and complex strategies designed to take advantage of instruction-level parallelism. However, on future multi-core processors this 'free' improvement will not materialise unless the software is multi-threaded, i.e., parallelised, and thus able to take advantage of the increasing number of cores. Furthermore, the number of cores predicted in future processors suggests software will need to scale 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. The need to simplify parallel programming, as indicated earlier, has led to a surge in TM research. 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 searches for conflicts by comparing each executing transaction's data accesses against that of all other concurrently executing transactions, 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, typically 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 high scalability on multi-core architectures, it is important that the number of conflicts is kept to a minimum, as conflicts require the execution of conflict resolution code, which reduces effective parallelism by deviating execution from application code. Effective parallelism is further reduced by the grace period offered by some contention management policies to the victim transaction of a conflict, during which the non-victim transaction must wait. Finally, if the conflict leads to an abort, then not only is effective parallelism reduced by restarting the aborted transaction and repeating work performed previously, but computing resources used in executing the aborted transaction are also wasted. This is made worse in certain (update-in-place) TM implementations, which use further computing resources in rolling back any updates made by the aborted transaction.

The order in which transactions are executed concurrently can affect the number of conflicts that occur, and given complete information a priori it may be possible to determine an optimal order (or schedule) that minimises the number of conflicts. 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 because it is impractical to obtain. Additionally, even if complete information is available, the search space for computing the optimal order of transactions is likely to be unfeasibly large.

This paper presents a novel technique called *steal-on-abort*, which aims to improve transaction ordering at runtime. As mentioned earlier, 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, which may lead to another aborted transaction. Steal-on-abort targets such a scenario: a transaction that is aborted is not restarted immediately, but instead 'stolen' by the non-aborted transaction, and queued

behind it, thus preventing the two transactions from conflicting again. Two key advantages of steal-on-abort are that it requires no application-specific information or offline pre-processing, and it is only executed when an abort occurs, thus adding no overhead when transactions are largely committing. Even when aborts do occur in large numbers, the evaluation suggests that the overhead of steal-on-abort is low.

Steal-on-abort is evaluated using 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: a sorted linked list [3], and 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 organised as follows: Section 2 introduces steal-on-abort, and Section 4 its implementation in DSTM2, along with strategies for tuning it to particular workload characteristics. Section 6 evaluates steal-on-abort's performance in the benchmarks mentioned earlier. Section 7 discusses recent related work. Finally, Section 8 completes the paper with a summary.

2 Steal-on-abort

In all TM implementations, conflict resolution is invoked when a data access conflict between two concurrently executing transactions is detected. Conflict resolution may give the victim transaction a grace period before aborting it, but once a transaction is aborted then typically it is restarted immediately. However, we observed that the restarted transaction may conflict with the same opponent transaction again, which we refer to as a *repeat conflict*, and lead to another abort.

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 takes advantage of this idea, and consists of three parts. First, it does not restart the aborted transaction immediately; the opponent transaction 'steals' the aborted transaction, and hides it. This dynamic reordering avoids repeat conflicts from occurring, since the two transactions are prevented from executing concurrently. Secondly, the thread whose transaction has been stolen acquires a new transaction to execute. The new transaction has an unknown likelihood of conflict, whereas the stolen transaction has a higher likelihood of conflicting due to the chance of a repeat conflict. If the new transaction commits, then throughput may improve. Thirdly, when a transaction commits it releases all the transactions it has stolen.

Since steal-on-abort relies on removing repeat conflicts to improve performance, 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 be less effective

at improving results. It is worth noting that steal-on-abort, like most contention management policies, reorders transactions with the aim of improving performance, which has implications for fairness. The effect on fairness is dependent on application characteristics, and it is beyond the scope of this paper to provide a complete analysis of steal-on-abort's impact.

Steal-on-abort's design complements the current function of conflict resolution. As the name suggests, steal-on-abort will only reorder transactions when the call to abort a transaction is made, not when a conflict is detected. Existing and future contention management policies can be used to respond to conflicts as usual, and steal-on-abort will only come into play when a transaction is aborted. This maintains the flexibility of using different contention management policies, while still attempting to reduce repeat conflicts.

3 Effectiveness and Applicability

However, the contention management policy can influence the performance of steal-on-abort. For example, steal-on-abort's benefit could be reduced when using the Greedy contention management policy [5], which in some cases causes the victim transaction to wait indefinitely for a resumption notification from its opponent. Since the victim is not aborted, steal-on-abort is never invoked. Conversely, steal-on-abort could have a greater impact when contention management policies such as Aggressive, which immediately aborts the opponent transaction, are used.

The benefit of steal-on-abort also changes with the method of updating shared data employed. Update-in-place requires an aborted transaction to roll back original values to shared objects as part of its abort operation. Steal-on-abort is likely to provide better throughput with update-in-place, than deferred-update, since steal-on-abort may reduce the number of aborts that occur, and thus the number of times roll back is performed. Deferred-update typically has a much faster abort operation, and consequently may see a lesser benefit from steal-on-abort.

Visibility of accesses also affect the benefit of steal-on-abort. The detection of repeat conflicts requires one transaction be active while another concurrently executing (active) transaction conflicts with it, and aborts, multiple times. Such a scenario can only occur when visible accesses are used: either read, write, or both, with both giving greater likelihood of detecting repeat conflicts. Steal-on-abort is not applicable if both reads and writes are invisible, as conflicts cannot be detected between active transactions, which prevents repeat conflicts from occurring. Furthermore, the quicker the accesses are detected, the higher the chance of repeat conflicts. As a result, steal-on-abort is likely to be most effective when visible reads and visible writes are used in conjunction with eager validation (i.e., checking for conflicts upon each data access, as opposed to lazy validation, which checks for conflicts after executing the transaction, not as each access is performed). This should not come as a surprise; steal-on-abort attempts

to reduce conflicts, and a configuration with visible accesses and eager validation is most suited to the quick detection of conflicts.

4 Implementation in DSTM2

This section details the concrete implementation of steal-on-abort in DSTM2, and then goes on to explain the two design variants of steal-on-abort evaluated in this paper. DSTM2, like most other STM implementations [6–8], creates a number of threads that concurrently execute transactions, and is extended to support the three key parts of steal-on-abort. First, each thread needs to store transactions stolen by its currently executing transaction. Second, each thread needs to acquire a new transaction if its current transaction is stolen. Finally, a safe mechanism for stealing active transactions is required. We implemented a thread pool framework to support the first two parts, and a lightweight synchronisation mechanism to implement the third part.

4.1 Thread Pool Framework

DSTM2, like other STMs [6–8], creates a number of threads that concurrently execute transactions. This is extended into a thread pool model where application threads submit transactional jobs to a thread pool that executes transactions. The thread pool model simplifies the task of stealing and releasing transactions, and for acquiring new transactions when a thread's transaction is stolen.

The thread pool comprises of worker threads, and Figure 1 illustrates that each worker thread has its own work queue holding transactional jobs, in the form of a deque (double-ended queue) named `mainDeque`. A transactional job is simply an object that holds parameters needed to execute a transaction. In remainder of this paper we use the terms transaction and job, and the terms deque and queue, interchangeably. Per-thread deques are used as a single global deque may result in high synchronisation overhead. Worker threads acquire jobs from the head of their own queue, and place it in the thread-local variable `currentJob`, to execute. Worker threads acquire a job when their current job commits, or is stolen through steal-on-abort. The benchmarks used in this paper were modified to load jobs onto each thread's `mainDeque` in a round-robin manner during benchmark initialisation, which is excluded from the execution times reported.

In order to keep threads busy, randomised work stealing [2] is used for load balancing. Figure 2 illustrates work stealing in action. If a thread's deque is empty, then work stealing attempts to steal a job from the tail of another randomly selected thread's deque. If a job is not available in the other thread's deque, then yet another thread is randomly selected. In any one attempt to steal jobs from other threads, 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 obtained through work stealing, then it is stored in the thread variable `currentJob`, and not in any deque. Since each thread's deque can be accessed by multiple threads concurrently, it needs to be thread-safe. As DSTM2 is Java-based, the deque used is a `java.util.concurrent.LinkedBlockingDeque`.

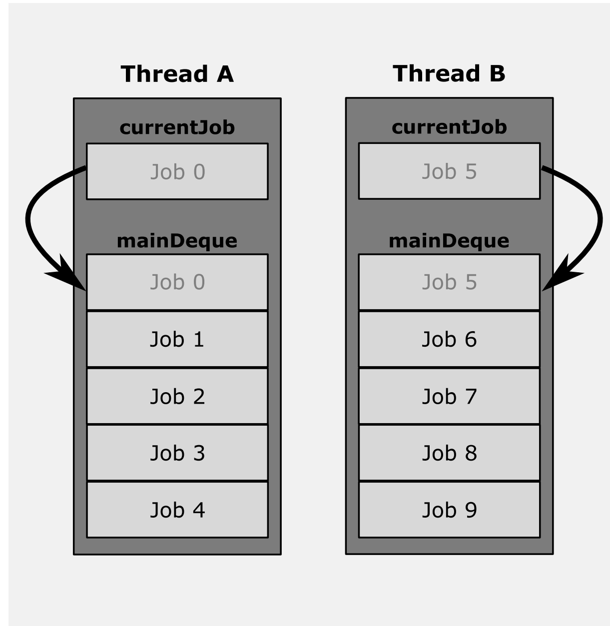

Fig. 1. DSTM2 is modified to implement per-thread deques that store transactional jobs. Threads take jobs from the head of their own deque.

4.2 Steal Operation

To implement the steal operation, a second private deque, named `stolenDeque`, is added to each worker thread to hold jobs stolen by an executing transaction. Once a transaction commits, the jobs in the `stolenDeque` are moved to the `mainDeque`. The second deque is necessary to hide stolen jobs otherwise they may be taken through randomised work stealing, and executed concurrently by other threads while the current transaction is still active, which re-introduces 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, following which it first attempts to acquire a new job, and then clears the `stolen` flag. The victim thread must wait on the `stolen` flag, otherwise access to the variable `currentJob` may 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, and places the job in the stealing thread's `stolenDeque`. After the job is taken, the victim thread's `stolen` flag is set.

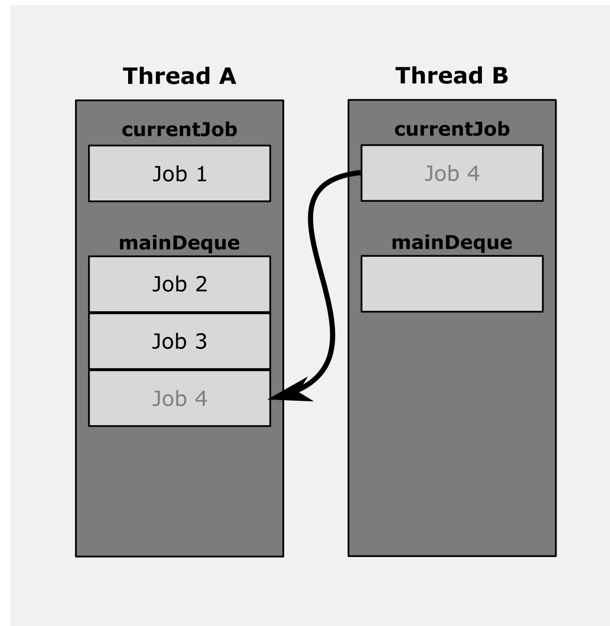

Fig. 2. If a thread's deque is empty, it steals work from the tail of another, randomly selected, thread's deque.

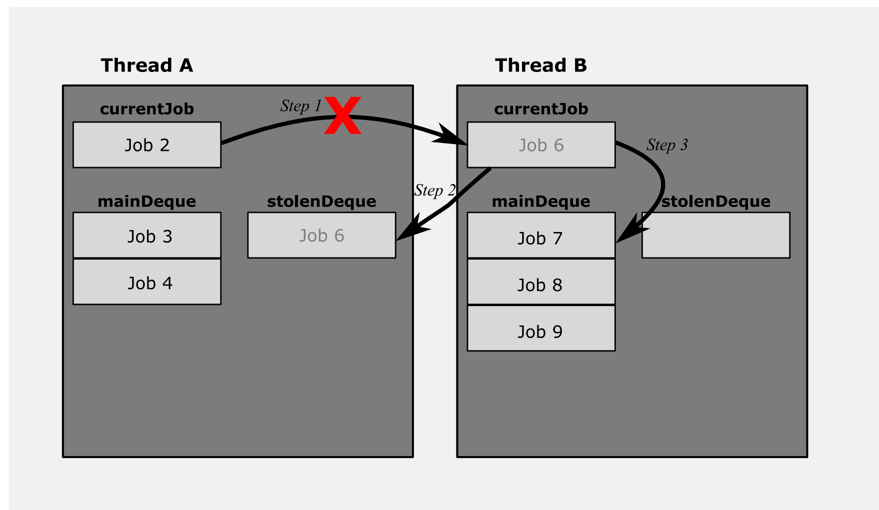

Fig. 3. Steal-on-abort in action. Thread A is executing a transaction based on Job 2, and Thread B is executing a transaction based on Job 6. In step 1, thread A's transaction conflicts with, and aborts, Thread B's transaction. In step 2, thread A steals thread B's job, and places it in its own `stolenDeque`.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

4.3 Semantic Considerations

There are two important changes to consider when using a thread pool to execute transactions. First, in the implementation described above, application threads 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 a return value from the committed transaction that may be required by the application thread's dependent code. Such synchronous job execution requires a simple modification to the implementation described already. However, the use of asynchronous job execution, where possible, is preferred as it permits greater parallelism. Solely using synchronous job execution limits the extent to which worker threads may execute simultaneously, as the maximum number of jobs would be limited to the number of application threads.

4.4 Eliminating Repeat Conflicts

The implementation described does not completely eliminate repeat conflicts. As transactions are abort-stolen by the currently executing transaction, they are placed in a steal queue associated with the thread in which the aborter is executing. When the transaction being executed by a thread commits, all the transactions in the thread's steal queue are moved to the thread's work queue. However, there is still a chance of repeat conflict. Imagine transaction T1 and T2 conflict, and T2 is abort-stolen by (and serialised behind) T1. Transaction T3 then conflicts with transaction T1, and abort-steals it. Now transactions T1 and T2 are in the steal queues of different threads (so T2 is no longer serialised behind T1), and could possibly conflict again.

A second approach is for *per-transaction steal queues*. When a transaction abort-steals another, it adds it to its own internal steal queue. Once a transaction commits, it releases its abort-stolen transactions from its internal steal queue into the work queue of the thread in which it was executing. This approach guarantees repeat conflicts do not occur, since an abort-stolen transaction will never be made visible until its stealer commits. This approach has been investigated, but is not presented in this paper, as the former approach is sufficiently efficient at reducing repeat conflicts for the evaluation performed.

5 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 of where Steal-Tail may benefit, 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.

6 Evaluation

This section presents highlights the potential performance benefit of using steal-on-abort, and analyses how such benefits are achieved, and the overhead of using steal-on-abort. As mentioned already, this paper only evaluates the per-thread steal queues implementation. In this section, 'Normal' 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 utilise the thread pool framework described earlier.

6.1 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, eager validation, and visible accesses. 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.

6.2 Benchmarks

The benchmarks used to evaluate steal-on-abort[1] are linked list [3], and STAMP-vacation [4]. Hereafter, they are referred to as List, and Vacation, respectively. List is a microbenchmark that transactionally inserts and removes random numbers from a sorted linked list. Vacation is a non-trivial transactional 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, and the method of achieving high contention for each benchmark is described. List is 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 6ms 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. The small size of the table, and the large number of modifications per transaction, results in high contention.

6.3 Contention Managers

A contention manager (CM) is invoked by a transaction that detects a conflict with another (opponent) transaction. In this evaluation, three CMs are used to provide coverage of the published CM policies: Aggressive [3], Polka [9], and Priority. Aggressive immediately aborts the opponent transaction. Polka, the published best CM, 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 [9]. Priority immediately aborts the younger of the two transactions based on their timestamps.

Steal-on-abort should be most effective with Aggressive and Priority, as they immediately make a call to abort the victim transaction. Conversely, steal-on-abort should be less effective with Polka, as it chooses to give the victim transaction a grace period before aborting it. In scenarios with high contention, Aggressive and Polka are more likely to cause transactions to temporarily livelock

[1] Reviewer's note: The SHCMP paper also used a red-black tree benchmark. This has been omitted to make space for further analysis of steal-on-abort's results with linked list and STAMP-vacation.

as they always choose to abort the opponent transaction. Priority will not livelock transactions since it is always selects the same transaction to abort (the one with the younger timestamp). Aggressive and Polka abort the opponent to provide non-blocking progress guarantees by preventing a transaction becoming indefinitely blocked behind a zombie opponent transaction. Priority does not provide such a guarantee.

6.4 Throughput

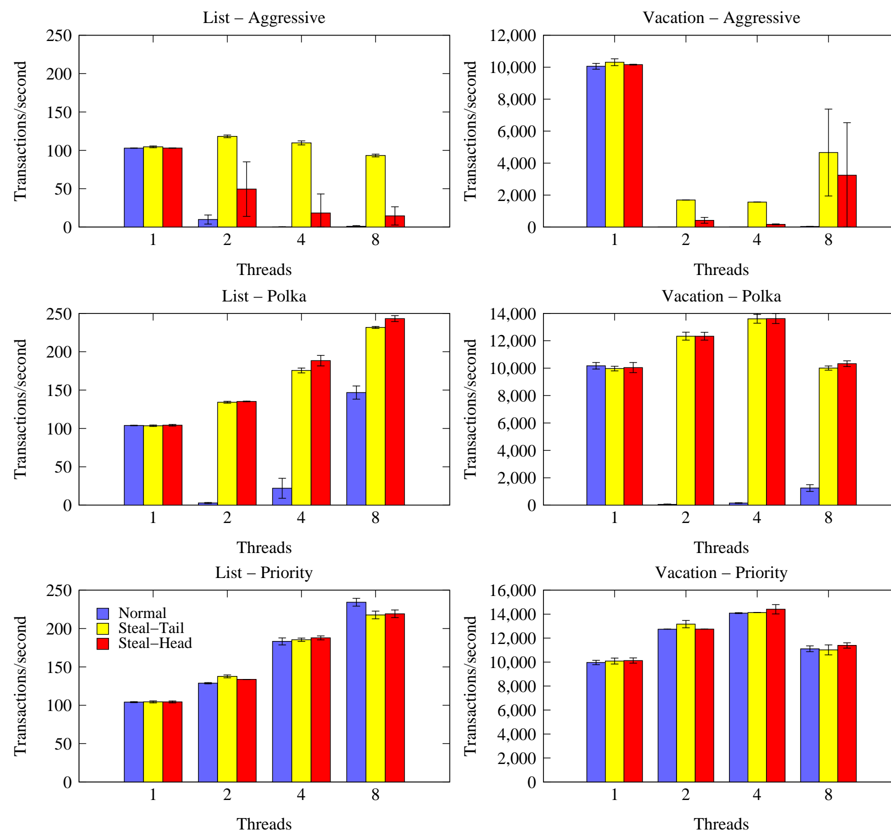

Fig. 4. Throughput results. Higher is better.

Figure 4 shows the transaction throughput results for List and Vacation. Looking at the Normal results (i.e., without steal-on-abort), the Aggressive CM gives the worst performance in both benchmarks, as the high contention scenarios cause the transactions to livelock due to Aggressive's policy to abort the opponent. The Polka CM fares better due to its grace period allowing opponents

to commit, but the policy to abort the opponent still causes transactions to livelock. Priority gives the best performance in both benchmarks, as it does not necessarily always abort the opponent, and thus avoids transactions livelocking.

Looking at the steal-on-abort results (i.e., both Steal-Tail and Steal-Head), Aggressive and Polka benefit significantly, while Priority sees no benefit of using steal-on-abort, with performance degrading slightly at eight threads in List. As explained earlier, Aggressive and Polka may livelock conflicting transactions through repeated conflict and abort. This raises the number of repeat conflicts that occur, which results in better throughput with steal-on-abort. Since Priority does not suffer from such livelock, it has fewer repeat conflicts, and thus benefits less from steal-on-abort. However, as mentioned earlier, there is a trade off in using Priority: it provides weaker non-blocking progress guarantee compared to Aggressive and Polka.

Drilling down further, Aggressive sees Steal-Tail give higher performance than Steal-Head in both List and Vacation. Steal-Head's large standard deviations in List with Aggressive, and both Steal-Head's and Steal-Tail's in Vacation at eight threads, are due to (data not shown here) execution either completing in a short duration, or approximately 5 times greater duration than the short duration. Comparing profiling data from the high and low throughput runs reveals that the low throughput runs are caused by the throughput falling almost instantaneously early in the execution, and then failing to recover. Although the reason for the drop in throughput is not known, it indicates an opportunity to improve steal-on-abort to avoid, or resolve, the condition that caused the drop in throughput, and achieve high throughput with greater consistency.

Polka sees Steal-Head give marginally higher performance than Steal-Tail in List, and at eight threads in Vacation, in contrast to the results with Aggressive. This contrast suggests that the steal-on-abort strategies are not only more suited for certain application traits, but also the CM used. It is also worth noting that Steal-Head improves Polka's performance such that it is within 3% of Priority's performance. Thus, it is possible to have both the performance of Priority, and the non-blocking progress guarantee of Polka.

6.5 Wasted Work

The transactional metric wasted work [10] 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. It is used here to see if steal-on-abort reduces this cost.

Figure 5 shows wasted work results. No transactional aborts occur in single thread execution since there are no other concurrent transactions to cause conflicts, and thus all single thread execution results have no wasted work. The results show the benchmarks' high contention parameters result in Aggressive and Polka spending nearly all of their execution time in aborted transactions (except List with Polka). This supports the suggestion that transactions are livelocked. There is negligible impact on wasted work of using steal-on-abort with Priority, which suggests a lack of repeat conflicts.

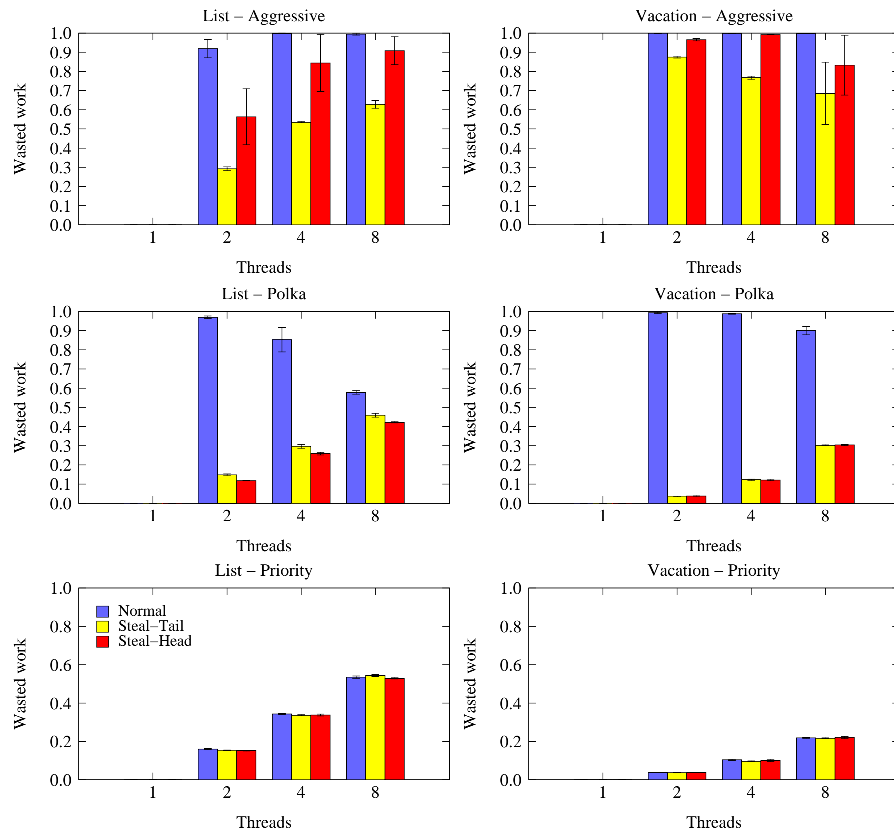

Fig. 5. Proportion of total execution time spent in aborted transactions (wasted work). Lower is better.

With Aggressive, Steal-Tail reduces wasted work by larger margins than Steal-Head in both benchmarks. Steal-Tail reduces wasted work by 30% to 70% in List, and 15% to 80% in Vacation. With Polka, Steal-Head reduces wasted by a slightly larger margin than Steal-Tail in both benchmarks. Steal-Head reduces wasted work by 57% to 87% in List, although the reduction decreases with the number of threads, and 95% to 99% in Vacation. These results reflect the trends seen in the throughput results seen earlier. Also, we mentioned earlier that steal-on-abort improves Polka's performance to within 3% of Priority's, and this is reflected in the similarity in wasted work between Polka with steal-on-abort, and Priority.

6.6 Aborts per Commit (APC)

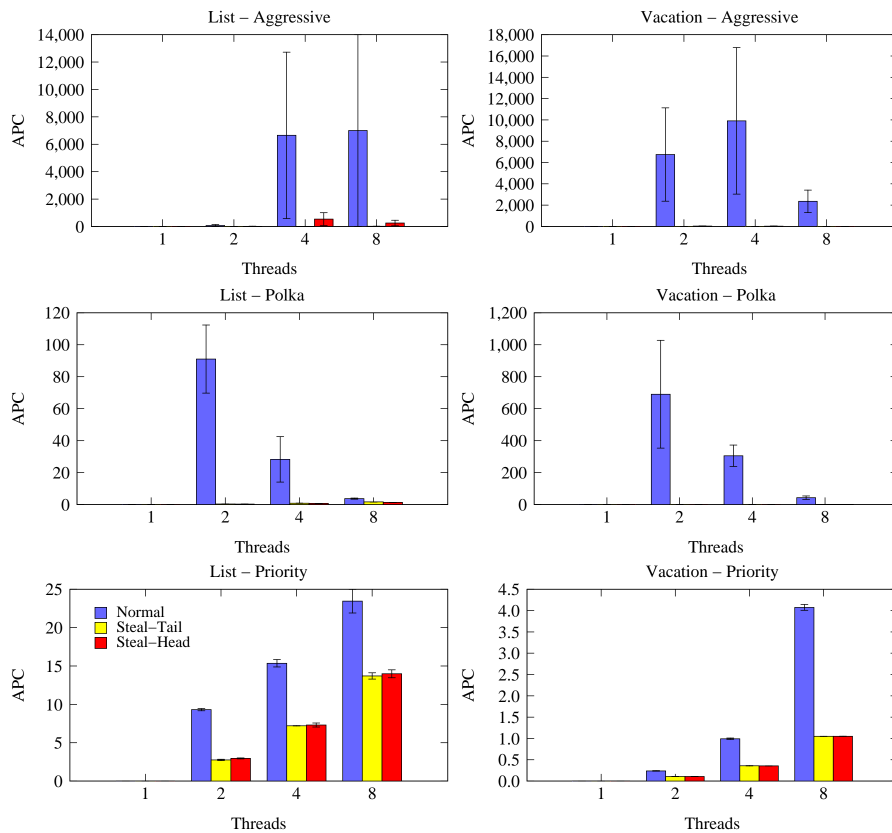

Fig. 6. APC results. Lower is better.

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 utilised, but it is not as useful as wasted work because it ignores the durations of the aborted and committed transactions. We investigate it here because steal-on-abort should reduce APC in benchmarks that exhibit repeat conflicts.

Figure 6-presents APC results. Using Aggressive, on average steal-on-abort has a low APC compared to Normal, but Normal exhibits significant variance in APC between runs of the same benchmark configuration. Polka's results paint a similar picture, except that the APC is significantly lower, and consistently falls as the number of threads increases. Polka's lower APC is due to the grace time that Polka offers an opponent transaction, which delays the call to perform an abort, and thus reduces the total number of aborts. Given the performance improvements of steal-on-abort for Aggressive and Polka, it is plausible that the high APC value with Normal is due to a large number of repeat conflicts.

Priority has a very low APC compared to Aggressive and Polka, attributable to the lack of livelock. Steal-on-abort reduces APC compared to Normal, but this does not correlate with the wasted work or throughput results shown in earlier sections. This implies that, on average, transactions make greater progress with steal-on-abort than Normal, but still get aborted eventually.

6.7 Repeat Conflicts

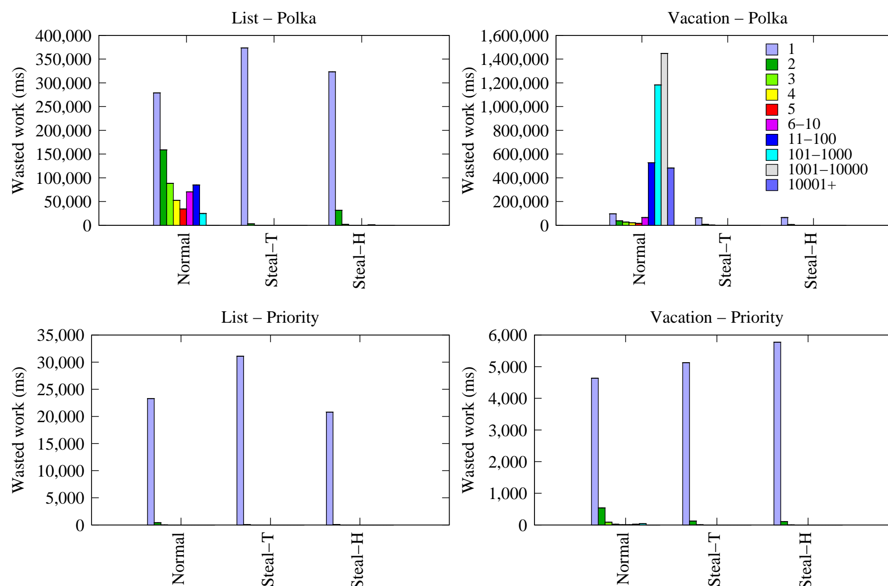

Fig. 7. Repeat conflict distribution. Lower is better.

This section examines the amount of time spent in repeat conflicts. Figure 7 shows histograms of the distribution of time spent in aborted transactions (wasted work) for a given number of conflicts with a particular transaction. As an example, consider transaction T1 aborts seven times before committing, thus it has seven lots of wasted work. Four aborts occur through conflict with transaction T2, two with T3, and one with T4. The time wasted in the executions of T1 that conflicted with, and were aborted by, T2 are added to column '4', the two lots associated with T3 are added to column '2', and the one lot associated with T4 is added to column '1'. For brevity, only the eight thread results are discussed, although better performance improvements were observed with fewer threads. Furthermore, results with Aggressive are not presented as there is significant variance in the repeat conflict results from one run to another.

Since steal-on-abort should target repeat conflicts it should reduce the amount of time in all but the first column. For Polka, this is confirmed by the results: Steal-Tail reduces time in the remaining columns (repeat conflicts) by 99% in List, and 99% in Vacation. Furthermore, the results show that repeat conflicts represent a significant proportion of the total wasted work: 65% in List, and 96% in Vacation. Thus, steal-on-abort is highly effective at reducing the number of repeat conflicts, even they occur in large proportions. In contrast, for Priority the number of repeat conflicts is quite low, which confirms previous suspicions, and explains why steal-on-abort did not improve performance with Priority as significantly as with Aggressive and Polka. Also, although there are differences in repeat conflict distribution results between Normal and steal-on-abort for Priority, recall that the difference in wasted work between them was minimal.

This raises the obvious question of whether steal-on-abort could benefit applications that use Priority since, so far, repeat conflicts have arisen due to the policies of Aggressive and Polka livelocking conflicting transactions. The answer is yes, steal-on-abort could benefit applications that use Priority. Recall Priority aborts transactions that are younger. Thus, a younger transaction can repeatedly conflict and abort against an older transaction. The longer the older transaction remains active, the more repeat conflicts and aborts possible. In such a case, steal-on-abort may improve performance with Priority when the aborted transaction is stolen, and then replaced with a new transaction from the work queue, which may or may not commit, and thus improve performance. This strategy was not useful in these benchmarks as they were configured to have generally high contention, and thus all transactions, including the new transaction from the work queue, had a high likelihood of conflicting with some other transaction, and the lower likelihood of a repeat conflict became insignificant.

It is worth noting that steal-on-abort increases single conflict (non-repeat) wasted work in most cases. In Polka 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 less than the decrease in repeat conflict wasted work because steal-on-abort prevents repeat conflicts from occurring, which leads to the reduction in wasted work shown earlier. In Priority the increase in single conflict wasted work is also due to

steal-on-abort attempting a new transaction on abort, whereas Priority may attempt the same transaction repeatedly (if it is younger than its opponent).

6.8 Steal-on-abort Overhead

Steal-on-abort consists of two short operations, and we have not tried to measure the overhead of these directly as the overhead of the measurement code is likely to distort the results. Thus, we attempt to measure the overhead of steal-on-abort indirectly. The first source of overhead is performing the steal operation. Figure 8 shows average committed transaction durations for each benchmark, which includes the overhead of the steal operation. Only results for Polka are shown, as its high number aborts suggests steal-on-abort is invoked a large number of times. Taking the standard deviations into account, generally the overhead of stealing transactions seems negligible. Furthermore, the transactions in Vacation are significantly shorter than those in List, yet stealing transactions does not add noticeable overhead.

The reductions in average committed transaction durations with steal-on-abort are due to Polka's policy. Polka causes transactions to wait for their opponents, which increases the average time it takes to execute a transaction that eventually commits if it encounters conflicts. 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.

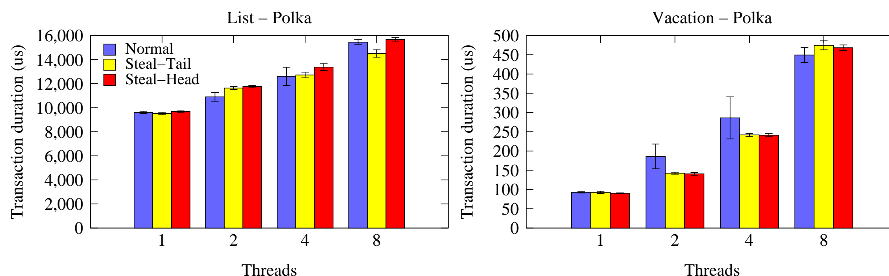

Fig. 8. Average Committed Transaction Duration (microseconds). Lower is better.

The other major source of steal-on-abort overhead is due to moving transactions in the `stolenDeque` to the `mainDeque` after the local transaction commits. The in-transaction metric (InTx), which is the proportion of execution time spent in executing transactions, is used to measure this overhead. For the benchmarks used in this evaluation there are two major sources of out-of-transaction execution: work stealing, and moving transactions between deques after a transaction commits. Since Normal execution also utilises work stealing, the difference between Normal and steal-on-abort execution should approximately represent the cost of moving jobs between the deques.

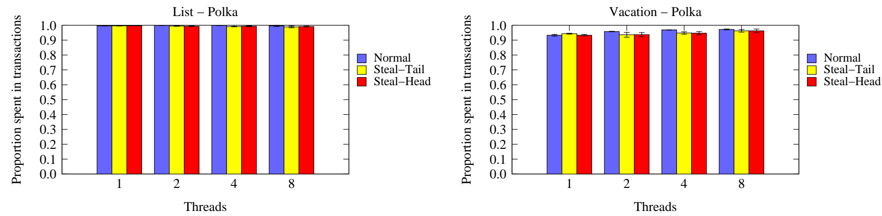

Fig. 9. Proportion of total execution time spent within transactions. Higher is better.

Figure 9 shows the InTx results, again only with Polka. It identifies that there is negligible overhead in moving jobs between deques and work stealing in List. However, in Vacation the overhead becomes visible, with most strategies observing an overhead of 3%. This equates to an average overhead of moving jobs of 2.8 microseconds per transaction. However, this cost is related to the number of jobs moved between deques, and with Steal-Tail this averages to 2.2 jobs per transaction.

7 Related Work

Limited research has been carried out in transaction re-ordering for improving TM performance. Bai *et al.* [11] 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. Furthermore, performance is based on the effectiveness of the formula, and it may be difficult to generate such formulae for some applications. In contrast, our approach does not require any application-specific information.

Dolev *et al.* recently published work similar to steal-on-abort called CAR-STM [12], which also attempts to reorder transactions to minimise repeat conflicts. CAR-STM uses a thread pool with multiple work queues to execute transactions. Application threads submit transactional jobs to the thread pool, and then suspend until the transaction is committed by one of the thread pool's worker threads (i.e., use synchronous job execution). CAR-STM consists of two parts: collision avoidance, and collision resolution.

Collision avoidance is an interface to support (optional) application-supplied conflict prediction routines. If supplied, these routines are used to queue transactions that are likely to conflict into the same work queue, rather than using round robin distribution. This concept is a simpler version of the adaptive key-based scheduler of Bai *et al.* [11] described above.

Collision resolution is similar to steal-on-abort; transactions that conflict are queued behind one another. CAR-STM also describes the per-thread and per-transaction steal queues. However they do not explore the design space as thoroughly; they do not attempt to support existing contention management policies, and they do not investigate stealing strategies such as releasing stolen

transactional jobs to the head or tail of a work queue. Furthermore, CAR-STM supports only synchronous job execution, and does not support work stealing, thus threads can be left idle, which reduces its performance potential. The thread pool implementation described in this paper recognises that not all scenarios require the application thread to wait for the transaction to complete, and allows asynchronous job execution, which can improve performance by increasing the number of jobs available for execution through work stealing, and by reducing the synchronisation overhead between application threads and thread pool worker threads.

8 Summary

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 caused by repeat conflicts. 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.

Steal-on-abort was evaluated against two widely used benchmarks in TM: a sorted linked list microbenchmark, and STAMP-vacation, a non-trivial TM benchmark. Performance improvements were observed when using the Aggressive and Polka CM policies, and Polka is the published best CM. Further analysis showed steal-on-abort improved their performance by eliminating repeat conflicts, and also showed that steal-on-abort is highly effective at removing repeat conflicts, even when they occur in significant proportions.

However, no benefit was observed with the Priority CM policy, and Priority provided the best non-steal-on-abort performance in the benchmarks used. Priority's policy provides weaker non-blocking progress guarantees than Aggressive and Polka, which allowed it to avoid livelocks they encountered. This reduced the number of repeat conflicts with Priority, and thus also the benefit of steal-on-abort. This introduces a trade off when selecting CM policies: high performance and weaker progress guarantees, or vice versa. However, steal-on-abort eliminated this trade off as it improved Polka's performance to within 3% of Priority's performance, and combined high performance, and superior progress guarantees.

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. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 258–264. ACM Press, July 2005.
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. 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.
10. 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.
11. 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*, pages 1–8. IEEE Computer Society Press, March 2007.
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*, pages 125–134. ACM Press, August 2008.