Hugh: A Semantically Aware Universal Construction for Transactional Memory Systems

Craig Sharp and Graham Morgan

School of Computing Science, Newcastle University
{craig.sharp,graham.morgan}@ncl.ac.uk

Abstract. In this paper we describe an implementation for exploring the scheduling of aborted transactions within transactional memory systems. We consider application semantics to be just as important as guaranteeing linearizability in arriving at an appropriate execution strategy. Our approach exploits parallelism to simultaneously create different execution orderings for rescheduled aborted transactions and chooses the most beneficial for application progression. The overall solution guarantees a lock-free universal construction if there exists at least one transaction that can commit. The appropriateness of our approach is demonstrated via micro-benchmark performance figures.

Keywords: Transactional Memory, Contention Management, Shared Memory, Concurrency Control, STM.

1 Introduction

Given the current barriers to processor frequency scaling, processor manufacturers have focused developments on parallel scaling of processing, in what has become known as the parallel revolution [1]. Unfortunately, writing concurrent software and solving problems in parallel is notoriously difficult for a wide class of problems (particularly in the area of system design). The key to exploiting parallelism effectively lies in the concurrency control mechanism used to provide correctness and progress guarantees to the concurrent program.

Transactional Memory has offered programmers a technique which simplifies the implementation of concurrency control. Atomic blocks allow sections of concurrent code to be composed, in a manner which is trivial in comparison to locking-based approaches. The problem with Transactional Memory lies in the occurrence of conflicts which require transactions to abort and retry their execution. If conflicts occur regularly and persistently, the Transaction Manager requires a Contention Management Policy (CMP) to mitigate the degradation of performance caused by aborted transactions.

From the programmer's perspective, conflicts fall into two categories: concurrent conflicts and semantic conflicts. A concurrent conflict occurs when the reads and writes of a transaction encounter an inconsistent state of shared memory and many contention managers combat these types of conflicts. A transaction may execute without interference however and still need to re-execute because semantically, the application cannot progress. For example, a transaction may need to consume an item from a shared buffer but finds it empty, or a bank account may have insufficient funds to permit a withdrawal.

Typically, a 'semantic conflict' can be dealt with in the application by (i) letting the transaction commit and re-execute in the future, (ii) or by using primitives (*retry*, *orElse* etc) as provided by Harris et al [2] which essentially allow ad-hoc coordination of transaction execution. We believe that the former approach is detrimental for application progression, raising the possibility of needless future conflicts when transactions re-execute. Meanwhile, the use of primitives places a burden on the application developer that must be addressed with an ad hoc solution, (re-introducing a fundamental problem of coordination with pessimistic concurrency control, which Software Transactional Memory originally sought to address).

In this paper we present an implementation of a Universal Construction approach to Contention Management called *Hugh*, that tackles both concurrent and semantic conflicts in an atomic object based STM model. We describe a speculative technique which serializes conflicting transactions to resolve concurrent conflicts and a parallel exploration which tackles semantic conflicts. Within the scope of this paper we consider a semantic conflict as simply the intentional abortion of a transaction by its own thread, and assume such conflicts can be avoided by executing the transaction in an alternative schedule.

The remainder of the paper is organized as follows: Section 2 describes the implementation. Section 3 describes the related work and Section 4 provides an evaluation of results obtained from an implementation of our technique. Section 5 concludes the paper and describes possible avenues for future work.

2 Implementation

The concept of the Universal Construction (hereafter UC) was first proposed by Herlihy [3] and allows any sequential data structure to be transformed into a linearizable representation that can be accessed and updated by n threads. There are three phases of UC operation: (i) threads prepare and announce a proposed input to add to the UC, (ii) each announcing thread performs consensus to decide which input will be added and (iii) a log of inputs is updated by the winning thread to reflect its input. We begin with an overview of how we use the UC technique and then provide greater detail in the remainder of this section.

2.1 Overview

We use the UC technique to provide conflict resolution and therefore the threads which use our implementation consist of the threads of aborted transactions. *Hugh* accepts as input a permutation of one or more sequentially executed transactions and decides which permutation will be added to the log.

When some $thread_a$ encounters a conflict it prepares its input to the UC by first adding its aborted transaction to a global *Transaction Table*; after this

Registration Phase the parameters of $thread_a$'s permutation are set. The thread then enters a *Speculative Phase* where it re-executes aborted transactions that have been added to the *Transaction Table*. We provide $thread_a$ with a private cache to hold copies of modified atomic objects, but no transaction is committed. Transactions are executed sequentially to prevent concurrent interference, but application semantics may still cause a transaction to abort explicitly (i.e. a semantic conflict).

During the *Speculative Phase* of $thread_a$, other threads may execute their own speculative transactions in parallel with $thread_a$. Once the *Speculative Phase* ends, each participating thread then enters a *Commit Phase* to decide which single thread's cache of modified atomic objects will be committed using a consensus algorithm. Threads whose transactions are committed return to normal execution, while those that remain aborted commence another *Registration Phase*.

Figure 1 contrasts our approach with a serializing CMP (like [4] for example). Two hypothetical scenarios, both containing a *depositor* and *withdrawer* transaction access a shared object. In scenario 1, the CMP reorders transactions to avoid concurrent conflicts. Although the *withdrawer* transaction can commit, it may need to re-execute in future (if deposits must precede withdrawals for example). In scenario 2, our approach is illustrated where a semantic abort occurs and each thread re-executes a different permutation of the aborted transactions.

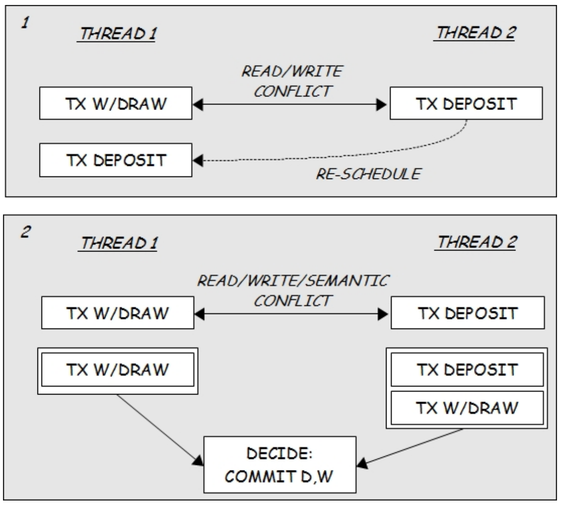

Fig. 1. In scenario 1 a read/write conflict has occurred between two transactions called withdraw (w/draw) and deposit. The depositor is aborted and rescheduled to execute after the withdrawer has committed. In scenario 2, Thread 1 aborts the depositor but then also aborts because of a semantic conflict caused by attempting to execute a withdrawal before a deposit. The conflict is resolved by the execution of an ordering which allows both to commit (deposit then withdraw).

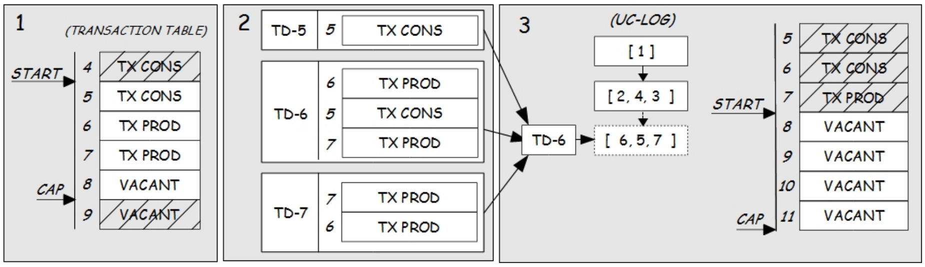

Fig. 2. In phase 1, threads add their transactions to the Transaction Table. In phase 2, a thread executes permutations of transactions within the window of the Transaction Table. In phase 3, transactions perform consensus to decide which permutation will be committed, and the result is added to the log of the Universal Construction and the Transaction Table window is advanced.

2.2 Aims and Contribution

Serializing aborted transactions to avoid concurrent conflicts has already been explored [5,6,4] given that under high contention, serialization can produce better throughput than a parallel approach. To our knowledge however, *Hugh* is the first to use additional threads to provide multiple serialized executions of aborted transactions in parallel and does not require the overhead of a thread-pool.

We combine both *direct-update* and *deferred-update* approaches in our implementation; until a transaction aborts, threads modify atomic objects directly but when transactions are retried, thread-private caches hold updates to accessed objects (as in the deferred-update model). While the use of direct-updates (sometimes called encounter time locking) is not a requirement of our approach, this technique was preferred having been shown to reduce the degree of wasted transaction execution [7]. Although the use of thread-private caching increases memory usage, it is hoped that this can reduce the occurrence of 'cache bouncing' (this approach was found to be particularly effective in Remote Core Locking [8]).

While the serial execution of aborted transactions tackles the problem of concurrent interference, we address semantic conflict by requiring that each thread execute different permutations of aborted transactions:

1. The possibility of having to re-execute a transaction because of a semantic conflict can be minimized given that there is a greater chance some permutation will avoid the semantic conflict;
2. If more transactions are aborted and conflicts are high, then this results in a greater number of threads available to explore more permutations of transaction execution. This in turn increases the possibility of finding an optimal transaction ordering so that more aborted transactions can be committed;
3. If parallel processing resources are increased, then a greater number of permutations can be explored in parallel, theoretically increasing the exploratory capacity of our approach within a shorter time-frame. Concurrency control is essentially transformed into a parallel state-space exploration problem.

List point 3 suggests our approach would benefit from a policy controlling the operating system scheduling of threads to processors and a platform with a plentiful supply of cores. This is beyond the scope of this paper and we confine our discussion to a purely 'user-level' implementation. We now describe each of the three phases of conflict resolution in detail.

2.3 Registration Phase

In order to re-execute its transaction, a thread is first required to register its transaction within a global table which we call the *Transaction Table*. When the transaction is added to this table it then belongs to the set of transactions from which permutations may be generated during the *Speculative Phase*. To avoid concurrency errors, threads use the synchronization primitive *compare-and-swap* to atomically increment an integer variable (called *current*) and thus gain a unique entry into the table.

Figure 2(1) shows the layout of the *Transaction Table*. In addition to *current*, the *Transaction Table* also maintains two integer variables called *start* and *cap*, which provide a 'window' (of size *cap* minus *start*) within which a thread's transactions must lie before it may execute its *Speculative Phase*. The window is the maximum length of the permutation the thread submits to the consensus algorithm. The first index of the permutation is equal to the threads index in the *Transaction Table* and subsequent entries are indices in the *Transaction Table* within the range of the window. For example, suppose some $thread_6$ registers and takes the $6th$ entry into the *Transaction Table* and $start = 5$ while $cap = 9$, then a valid permutation for $thread_5$ is $\{6, 5, 7, 8\}$ (see Figure 2(2)).

During the commit phase, the window is 'advanced' to allow new threads to begin their *Speculative Phase* (Figure 2(3)). Note that increasing the size of the window increases the maximum number of transactions that may commit in a single commit phase (throughput) but also incurs extra computational overhead, including the computation of consensus (the maximum number of participants in the consensus algorithm is equal to the window size). While our own experiments found that a maximum window size of 16 produced the best performance, an attractive avenue for future work would be to expand and contract the size of the window at runtime, based on the level of contention.

2.4 Speculative Phase

Once a thread has registered its aborted transaction, it commences its *Speculative Phase* (for brevity, we shall hereafter refer to these threads as *speculators*). The speculator executes transactions held in the *Transaction Table* with the aim of executing as many transactions to completion as possible. While the speculator is executing, new speculators may register (causing newly aborted transactions to appear in the *Transaction Table*) and begin their own speculative execution. All speculators must ensure two conditions are met: (i) exclusivity of atomic objects to ensure any speculative execution is sequentially *consistent* and (ii) the *Speculative Phase* must *terminate*.

Consistency. While speculators modify private copies of atomic objects, they must ensure that no active (non-aborted) thread modifies the original, otherwise their execution would be inconsistent and could not commit. Speculators must therefore have exclusive access to any atomic object they update, (as they do not update the objects directly, they do not require exclusivity from other speculators). We require that each atomic object has an owner field and that active transactions have to install themselves as owner of any atomic object they wish to modify. To support exclusivity, each atomic object also possesses an integer field denoting its version (*version*) and a reference to a global clock (*clock*). In addition, we provide a global transaction (*spec*) to denote that an object is currently owned by a speculator. The procedures for accessing atomic objects are:

– The first time an atomic object is accessed by a speculator, it checks whether the object is owned by another speculator ($owner = spec$). If true, the thread caches a copy of the object and continues its transaction (subsequent accesses modify the copy);
– If the object is not owned by a speculator and ($version <= clock$), it sets ($version = clock+1$), aborts the current owner of the object, and installs the *spec* transaction as the new owner. Setting the value of *version* eliminates the possibility that another thread can repeatedly prevent the speculator from changing the owner of an atomic object;
– Once consensus has been reached and the winning transactions have been committed, *clock* is atomically incremented so that ($version \leq clock$) is true, and any thread may once again own the object.

Before a thread executing an active transaction tries to install itself as the owner of any atomic object, it first checks whether $version \leq clock$. If this evaluates to false, then the thread knows it must abort because the object is currently being modified by a speculator. The thread will now register and become a speculator itself.

Terminating Speculation. Transactions are executed according to the indices of the speculator's permutation until either: (i) a transaction aborts, (ii) all transaction at the indices in the permutation have been executed, (iii) or the next index in the *Transaction Table* does not contain a transaction. The maximum number of transactions any speculator may execute during a single *Speculative Phase* is equal to the the size of the window. As each speculator executes a unique permutation, then for n speculators this means that a maximum of n permutations may be executed during a single session. Each speculator records the indices of the transactions it has executed successfully. Once a speculator has finished its *Speculative Phase*, it moves onto the *Commit Phase*.

2.5 Commit Phase

Once each speculator has completed its *Speculative Phase*, the log of the UC must be updated with the permutation of transactions the speculator has executed. To accomplish this:

1. Each speculator submits its permutation of executed transactions to the *decide* method of a consensus protocol;
2. The winner is decided by the permutation with the greatest number of executed transactions. The winning speculator commits the changes to the atomic objects in its cache and appends the permutation to the log (a linked list) provided by the UC.

The log provides each speculator with the necessary information to determine whether its own transaction has been successfully committed. Each speculator searches for its allocated index into the *Transaction Table* within the winning permutation appended to the log. If the permutation contains a speculator's index, that speculator's transaction has been committed. Once the winning speculator has committed the atomic objects in its cache, it atomically increments the global clock (such that $version \leq clock$ is true), indicating that any thread may once again own any atomic object. The window in the *Transaction Table* is then advanced by the winning speculator.

3 Related Work

Implementing an optimum Contention Management Policy (CMP) itself is a non-trivial task and numerous approaches have been developed. The first CMP techniques can be categorized by their employment of a wait-based criteria [9] (such as *Greedy*, *Karma*, *Polka* etc). These approaches were relatively trivial to integrate with existing STMs, requiring no involvement from the scheduling mechanism of the platform. Heber et al [10] observed an inefficiency with wait-based approaches given the difficulty in ascertaining the duration that an aborted transaction should wait before re-executing; too short could cause a repeat conflict and too long would be inefficient.

Serializing Contention Managers attempt to improve the inefficiency of wait-based CMPs by rescheduling aborted transactions to execute after a conflicting transaction and *Hugh* loosely follows this approach. Bai et al [5] introduced an approach which used 'keys' to predict the likelihood of conflicts between transactions; such transactions could then be scheduled to execute in sequence to avoid the possibility of concurrent conflict. Dolev et al introduced CAR-STM [6] which like Bai's work predicts the likelihood of conflicts between transactions and executes those transactions serially. Ansari et al developed an approach called Steal on Abort [4] where transactions could be 'stolen away' from threads with high workloads and work sharing between transaction executing threads was facilitated. *Hugh* differs from these serialising techniques by considering the effects of semantic conflicts and executing multiple transaction orderings in parallel.

Adaptive CMPs have also been developed, Yoo and Lee for example, introduced ATS [11]; a serializing CMP which used a threshold value to dynamically determine when aborted transactions should be serialized, based on a measure called the *contention intensity* and Heber et al provided *CBench*, a useful benchmark for evaluating serializing CMPs [10]. Heber et al identified a phenomenon they call mode oscillations, (where performance is hurt because the Contention Manager repeatedly switches between serialization and parallel execution) and implemented a stabilization algorithm to address the problem. Although adaptation has not been explored in our approach, potential future work may involve varying the window size with respect to contention levels.

Like *Hugh*, several contributions have approached transaction memory in the context of building a UC. Wamhoff [12] and Chuong [13] demonstrated how transactions could be used with a UC to handle failure. More recently Crain et al [14] developed a UC which in theory could remove the need for programmers to observe aborts. Unlike previous UC approaches, *Hugh* uses the UC for contention management only and submits multiple transactions as input to the consensus algorithm.

4 Evaluation

In this section we present results from a set of micro-benchmarks performed on an implementation of our system. The tests were executed on a Dell Alienware desktop PC featuring 4 x dual-core 3.40GHz Intel(R) processors (i7-2600) with 16GB of RAM, running Windows 7. The Transactional Memory software was executed in Visual Studio 2010 with a C Sharp implementation of the Java DSTM2 benchmark suite [15] (using the obstruction free factory with visible reads). Each experiment is carried out using an increasing number of threads (from 2 to 12) and executed 10 times with the average results provided. The Polka Contention Management Policy [16] has been cited as providing the best performance of wait-based Contention Managers, and so this was used to provide a comparison with our implementation (using the default parameters with respect to back-off time).

Two benchmarks were used to test the performance of our implementation: a linked list and a hash table. In both benchmarks, threads are divided into 'producers' and 'consumers' in equal number. Producers and consumers take a random value and attempt to *insert* this into the data structure in the case of the producer, or *remove* it in the case of the consumer. The highest frequency of read/write conflicts is expected in the linked list benchmark compared to the hash table which distributes items in an array of linked lists based on hashes generated from each item.

Performance results under increasing levels of semantic conflicts are provided. When there are no semantic conflicts (labelled S-L0), then threads only abort transactions if there is a read/write conflict. With Level 1 semantic conflicts (S-L1), consumer threads explicitly abort their transaction if they attempt

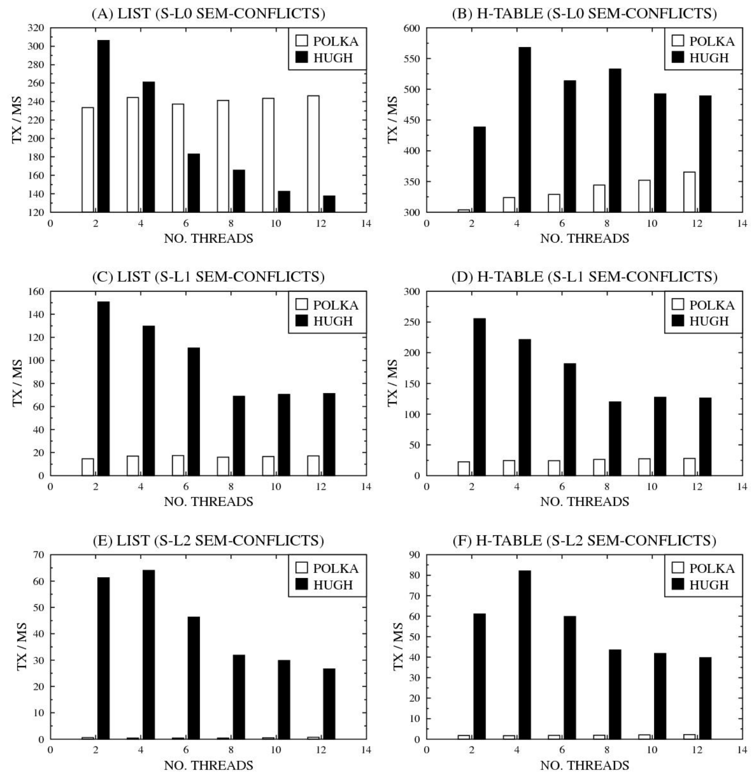

Fig. 3. Transaction Throughput

to remove an item which is not already present in the data-structure. Using Level 2 semantic conflicts (S-L2) also causes producers to abort their transactions if they attempt to add an item to a data-structure which is already present.

4.1 Transaction Throughput

Figure 3 illustrates the results for transaction throughput. The Y-axis denotes the number of transactions committed per millisecond and X-axis shows the number of threads present. In Graph A, using the list benchmark with S-L0 semantic conflicts we can see that the Polka manager performs better than *Hugh* once the number of threads increases beyond 6 due to the increase in read/write conflicts. One possible explanation is that the serialization of aborted transactions used by *Hugh* is less effective in this situation than the Polka policy and the lack of semantic conflicts means there is little benefit from the execution of permutations. In Graph B where the hash table reduces the number of read/write conflicts we see better performance under both policies, though the greatest increase in throughput is witnessed with *Hugh*.

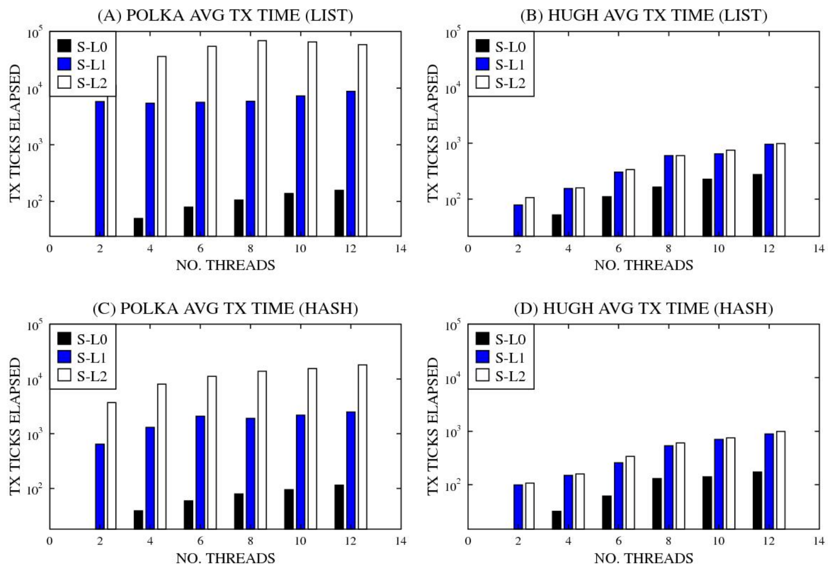

Fig. 4. Transaction Timing (Ticks)

Once semantic conflicts are introduced, *Hugh* performs markedly better than Polka under both benchmarks. With S-L1 semantic conflicts, *Hugh* shows a minimum improvement in throughput over Polka by a factor of approximately 4.3 and 4.5 for the list (Graph C) and hash table (Graph D) respectively. With S-L2 semantic conflicts, *Hugh* shows a minimum improvement by a factor of approximately 40 and 18, for the list (Graph E) and hash tables (Graph F) respectively.

Observe that with the Polka manager, as semantic conflicts are introduced the type of data structure used has less of an effect on mitigating the presence of aborts. It seems reasonable to assume that strategies for mitigating conflicts in transactional memory which rely on more 'concurrent' data-structures are of little benefit if one takes into account the kinds of semantic conflicts generated in these experiments.

4.2 Average Transaction Execution Time

In Figure 4 the average transaction execution time (ATET) is shown. In each graph, the Y-axis measures the ATET but note that the scale used is logarithmic for greater clarity and the maximum value is 10^5 ticks for all graphs. Each graph provides the results for a particular contention manager with a particular benchmark, and each bar shows the performance under a different semantic conflict level. The time is measured in elapsed ticks, (the fastest unit of time that can be measured on the platform) and denotes the average time spent executing a transaction by all threads.

One would expect that greater throughput generally corresponds to less average time spent executing a transaction (this is not guaranteed however, as unlike execution time, throughput also includes time spent outside of transaction execution). Given that *Hugh* resolves both concurrent and semantic conflicts, there should be less time required to execute a transaction when semantic conflicts are introduced, whereas with the Polka manager, transaction time should increase if repeated conflicts cause threads to back off (which involves calling the sleep function).

The performance of the Polka manager is shown in graphs A and C. One may observe that the ATET increases substantially as the level of semantic conflicts is increased. Conversely, the performance of *Hugh* (graphs B and D) does not exhibit the same degree of increase in ATET as the number of semantic conflicts is increased. This seems to suggest that the overhead of executing our policy does not increase substantially as semantic conflicts increase, unlike the Polka manager.

5 Conclusion and Future Work

This paper presents *Hugh*, a UC where threads conduct speculative execution of aborted transactions and 'commit by consensus', to mitigate both concurrent conflicts, and semantic conflicts; where some logical condition in the application ultimately prevents the progress of threads. We have described how conflicts can be resolved by a parallel exploration of transaction permutations and provided initial results which demonstrate increased throughput over a published contention manager.

The evaluation section presented some encouraging results via micro benchmarks in a custom scenario. Future work will require further testing with more sophisticated benchmarks. One issue with existing benchmarks, however, is that they evaluate performance with respect to concurrent conflicts, rather than the progress of the application (although this is not surprising given the immense scope of what can be defined as a semantic conflict).

We believe the most significant contribution made by our approach is the treatment of transaction conflict resolution as a state space exploration problem and in future we plan to conduct experiments with transactions of greater complexity, (nested transactions for instance). We anticipate that far from being a hindrance, semantic conflicts are useful as they will allow the state space of aborted transactions to be 'pruned' in favour of permutations which actually provide greater progress to the application.

References

1. Herlihy, M., Luchangco, V.: Distributed computing and the multicore revolution. ACM SIGACT News 39(1), 62–72 (2008)
2. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60. ACM (2005)

3. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(1), 124–149 (1991)
4. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 4–18. Springer, Heidelberg (2009)
5. Bai, T., Shen, X., Zhang, C., Scherer, W., Ding, C., Scott, M.: A key-based adaptive transactional memory executor. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–8. IEEE (2007)
6. Dolev, S., Hendler, D., Suissa, A.: Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, pp. 125–134. ACM (2008)
7. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 237–246. ACM (2008)
8. Lozi, J., David, F., Thomas, G., Lawall, J., Muller, G.: Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In: Work in Progress in the Symposium on Operating Systems Principles, SOSP, vol. 11 (2011)
9. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 258–264. ACM (2005)
10. Heber, T., Hendler, D., Suissa, A.: On the impact of serializing contention management on stm performance. Journal of Parallel and Distributed Computing (2012)
11. Yoo, R., Lee, H.: Adaptive transaction scheduling for transactional memory systems. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, pp. 169–178. ACM (2008)
12. Wamhoff, J., Fetzer, C.: The universal transactional memory construction. Technical report, 12 pages, University of Dresden, Germany (2010)
13. Chuong, P., Ellen, F., Ramachandran, V.: A universal construction for wait-free transaction friendly data structures. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, pp. 335–344. ACM (2010)
14. Crain, T., Imbs, D., Raynal, M.: Towards a universal construction for transaction-based multiprocess programs. In: Bononi, L., Datta, A.K., Devismes, S., Misra, A. (eds.) ICDCN 2012. LNCS, vol. 7129, pp. 61–75. Springer, Heidelberg (2012)
15. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. ACM SIGPLAN Notices 41, 253–262 (2006)
16. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 240–248. ACM (2005)