Commit Algorithms for Scalable Hardware Transactional Memory

Seth H. Pugsley, Rajeev Balasubramonian

UUCS-07-016

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

August 9, 2007

Abstract

In a hardware transactional memory system with lazy versioning and lazy conflict detection, the process of transaction commit can emerge as a bottleneck. For a large-scale distributed memory system, we propose novel algorithms to implement commit that are deadlock- and livelock-free and do not employ any centralized resource. These algorithms improve upon the state-of-the-art by yielding up to 59% improvement in average delay and up to 97% reduction in network traffic.

1 Introduction

Many research groups have identified *transactional memory (TM)* [3] as a promising approach to simplify the task of parallel programming. In a TM system, critical sections are encapsulated within transactions and it is the responsibility of either software or hardware to provide the illusion that the transaction executes atomically and in isolation. Many recent papers [1, 2, 4, 5] have argued that the implementation of transactional semantics in hardware is feasible. Most of these studies have considered small-scale multiprocessor systems (fewer than 16 processors) and have shown that *hardware transactional memory (HTM)* imposes tolerable overheads in terms of performance, power, and area. However, it is expected that the number of cores on a chip will scale with Moore's Law. Further, transactional parallel programs will also be executed on multi-processors composed of many multi-core chips. If HTM is to be widely adopted for parallel programming, it is necessary that the implementation scale beyond hundreds of cores. The HTM community is just starting to explore such scalable implementations.

An HTM system is typically classified by its choice of versioning and conflict detection mechanisms. For example, the Wisconsin LogTM implementation [4] employs eager versioning and eager conflict detection. The implementation is expected to have the same scalability properties as a directory-based cache coherence protocol. A salient dis-advantage of this approach is that it can lead to deadlocks/livelocks and requires a contention manager. A second approach, employed by the Stanford TCC project [2], adopts lazy versioning and lazy conflict detection. While this approach is deadlock-free, it is inherently less scalable. A recent paper attempts to extend the TCC implementation to improve its scalability [2], but leaves much room for improvement (explained in Section 2). Given the above advantages and disadvantages of each implementation, there is no consensus within the community on the most promising approach for HTM. Since the Stanford TCC approach is among the front-runners and since scalability is the bottleneck for that system, this paper focuses on improving the scalability for that design. We propose novel algorithms to commit a transaction in a scalable manner. These algorithms are free of deadlocks/livelocks, do not employ a centralized agent, provide high performance in the common case, and significantly reduce the number of network messages, relative to the Stanford TCC implementation.

In Section 2, we provide details on the state-of-the-art Stanford TCC implementation and identify inefficiencies in its commit algorithm. Section 3 describes our proposed algorithms and these are evaluated in Section 4. Conclusions are drawn in Section 5.

2 Background

In an HTM system, the hardware provides the illusion that each transaction executes atomically and in isolation. In reality, each thread of the application can start executing a transaction in parallel. The hardware keeps track of the cache lines that are read and written by the transaction (referred to as the *read-set* and *write-set*). In a lazy versioning system such as Stanford-TCC, writes are not propagated beyond the private cache. If the transaction reaches the end without being aborted, it commits by making all of its writes visible to the rest of the system. The cache coherence protocol ensures that other shared copies of these cache lines are invalidated. At this time, other in-progress transactions that may have read these cache lines abort and re-start. Without the above step, the illusion of atomicity for each transaction cannot be provided. In this lazy versioning system, a number of steps are taken during the commit process, possibly making it a bottleneck in a large-scale system. The algorithm for commit can be made quite simple if only one transaction is allowed to commit at a time. However, this is clearly not acceptable for a system with more than a hundred processors. In a recent paper, Chafi et al. [2] attempt to provide scalable parallel commits in a large-scale multiprocessor system.

The following baseline large-scale multiprocessor platform is assumed in that work. Numerous processors (possibly many multi-cores) are connected with a scalable grid network that allows message re-ordering. Distributed shared-memory is employed along with a directory-based cache coherence protocol. Since memory is distributed, the directory associated with each memory block is also distributed. The problem with allowing multiple parallel transaction commits is that a subset of these transactions may conflict with each other. The discovery of these conflicts mid-way through the commit process must be handled elegantly. As a solution, Chafi et al. propose the following algorithm that is invoked by a transaction when it is ready to commit:

1. Obtain TID: A centralized agent is contacted to obtain a transaction ID (TID). The TIDs enforce an ordering on transaction commits. The hardware goes on to ensure that the program behaves as if transactions execute atomically in the order of their TIDs.

2. Probe write-set directories: For every directory in the transaction's write-set, a *probe* message is sent to check if earlier transactions (those with smaller TIDs) have already sent their writes to that directory. If this condition is not true, probes are sent periodically until the condition is true. For every directory that is not part of the transaction's write-set, a *skip* message is sent so that directory knows not to expect any writes from this transaction.

3. Send *mark* messages: For all the cache lines in the transaction's write-set, mark messages are sent to the corresponding directories. This lets the directories know that these

cache lines will soon transition to an Owned state as soon as the final commit message is received from the transaction.

4. Probe read-set directories: For every directory in the transaction's read-set, another probe message is sent to check if those directories have already seen writes from earlier transactions. If this check succeeds, the transaction can be sure that it will not be forced to abort because of an earlier transaction's write. Probes are sent periodically until the check succeeds.

5. Send commit messages: A commit message is sent to every directory in the transaction's write-set. The corresponding cache lines transition to Owned state (with the corresponding transaction's core as owner) and send out invalidates to other caches that may share those cache lines. These invalidates may cause a younger transaction to abort if the lines are part of the younger transaction's read-set.

To summarize, the above algorithm first employs a centralized agent to impart an ordering on the transactions. Transactions can proceed with some of the steps of the commit algorithm in parallel as long as their read-set and write-set directories are distinct. If two transactions have to access the same directory, the process is serialized based on the TIDs of the two transactions. In other words, each directory allows only a single transaction to commit at a time, but assuming that transactions access different directories, there is a high degree of commit parallelism. The algorithm is deadlock- and livelock-free because transactions are assigned increasing TIDs when they make their first attempt to commit and a transaction is never forced to wait upon a transaction with a higher TID.

While it was shown that this algorithm has good performance [2], there are many inefficiencies in it. Firstly, a centralized agent hands out TIDs, a feature that is inherently non-scalable (although, the argument can be made that the data bandwidth requirements in and out of this centralized agent are modest). Secondly, all of the directories must be contacted in Step 2 above, an operation that clearly scales poorly as the number of cores is increased. Thirdly, if the initial probes in steps 2 and 4 fail, the probes must be periodically re-tried. In this paper, we attempt to address all of the above inefficiencies: our algorithms employ no centralized agent and significantly reduce the number of required messages (by avoiding re-tries and communication with every directory).

In the above algorithm, the total number of messages required per commit (not including the invalidates sent by the cache coherence protocol) equals

$$2 + 2w + (N - w) + W + 2r + 2w + PR$$

where N represents the number of directories, w represents the number of directories in the write-set, W represents the number of cache lines in the write-set, r represents the number

of directories in the read-set, and PR equals the number of probe re-tries. It must be pointed out that w, W, and r are typically small [2] and may not scale up with N if the application has good locality. The best-case network delay for the above algorithm (not including coherence operations and not including processing delays at the controllers) equals

$$(2 + 2 + 2 + 2) \times avg_delay,$$

where avg_delay is the average one-way delay on the network to send a message and we assume that messages in Step 2, Steps 3 and 4, and Step 5 can be all handled in parallel.

The number of required messages and the complexity of the algorithm can be reduced by alternative means. For example, in Step 1, when the centralized agent is contacted to obtain the TID, the transaction can also communicate its read and write sets to this agent. The agent keeps track of the read and write sets for all transactions that have not finished their commit process. If the agent can confirm that the new transaction has no conflicts with these outstanding transactions, it allows the new transaction to proceed with its commit. The new transaction can now freely propagate its writes and finally inform the centralized agent when it's done. This notification may allow a previously blocked transaction to receive commit permissions from the centralized agent. This approach increases the bandwidth requirements in and out of the centralized agent. It may be possible to reduce this requirement with the use of partial addresses or signatures that approximate the read and write sets. However, we believe that a centralized resource is an inelegant choice for a scalable algorithm and we do not further consider this option. It is also worth noting that with such a solution, even a single thread with high locality may have to contact a remote node to obtain its TID on every transaction commit.

3 Scalable Commit Algorithms

We next propose commit algorithms that avoid a centralized resource and have message requirements that do not scale up with the number of nodes. We begin with a conceptually simple algorithm and then add a modest amount of complexity to accelerate its execution time. Similar to the work by Chafi et al. [2], we assume a distributed shared memory system with a directory-based cache coherence protocol. To keep the discussion simple, we assume that the number of processors equals the number of directories, but typically, a single directory/memory module will be shared by a collection of processors. The above platform is also representative of a single multi-core processor where the L2 cache is shared, the L2 maintains directory state to maintain coherence among the L1s, and the L2 banks are physically distributed among the cores.

3.1 Algorithm 1: Sequential Commit

Each directory has an *"Occupied"* bit that indicates that a transaction dealing with this directory is in the middle of its commit phase. In this first algorithm, a transaction sequentially proceeds to "occupy" every directory in its read- and write-set in ascending numerical order (Step 1) (the transaction must wait for an acknowledgement from the earlier directory before proceeding to occupy the next directory). A directory is not allowed to be occupied by multiple transactions, so another transaction that wishes to access one of the above directories will have to wait for the first transaction to commit. After Step 1, the first transaction knows it will no longer be forced to abort by another transaction and it proceeds with sending information about its write-set to the corresponding directories (Step 2); these cache lines will be marked as Owned in the directory and invalidations are sent to other sharers of these lines. After receiving acknowledgements back from all directories, the transaction contacts all directories again to re-set the Occupied bit (Step 3).

If a transaction attempts to occupy a directory that is already occupied, the request is buffered at the directory. If the buffer is full, a NACK is sent back and the transaction is forced to re-try its request. In our experiments, we observe that re-tries are uncommon for reasonable buffer sizes. The buffered request will eventually be handled when the earlier transaction commits. There is no possibility of a deadlock because transactions occupy directories in numerically ascending order and there can be no cycle of dependences. Assume transaction A is waiting for transaction B at directory i. Since transaction B has already occupied directory i, it can only stall when attempting to occupy directory j, where $j > i$. Thus, a stalled transaction can only be waiting for a transaction that is stalled at a higher numbered directory, eliminating the possibility for a cycle of resource dependences. This algorithm imposes an ordering on conflicting transactions without the use of a centralized agent: the transaction to first occupy the smallest-numbered directory that is in the read/write-sets of both transactions, will end up committing first.

The total number of messages with this algorithm equals

$$2(w + r) + 2W + 2(w + r) + PR',$$

where PR' equals the number of re-tries because of lack of buffer space. The number of messages in Step 2 can be reduced further if the directory sends the transaction a single acknowledgement for all the cache lines for that directory. The best-case network delay for this algorithm equals

$$(2 \times (w + r) + 2 + 2) \times avg_delay.$$

This assumes that the operations in Step 1 are performed sequentially, operations in Step 2 are performed in parallel, and operations in Step 3 are also performed in parallel.

3.2 Algorithm 2: Momentum-Based Commit

While the above algorithm helps reduce the total number of required messages, the delay may be higher than that of the Stanford-TCC algorithm because each of the directories must be occupied in sequential order. In most cases, the set of directories accessed by a transaction is small and assuming locality, transactions will conflict infrequently for a directory. Yet, the directories must be occupied sequentially in case there is a conflict. To remove the dependence on this sequential process, we propose the following momentum-based commit algorithm. Transactions attempt to occupy directories in parallel and a transaction is allowed to steal a directory from another transaction if it is further along in its commit process.

In Step 1a, transaction T1 sends out parallel requests to occupy directories in its read- and write-sets. These requests carry a *"momentum"* value of 0 to indicate that the transaction has occupied 0 directories so far. If a directory is unoccupied, it sends back an acknowledgement and transaction T1 increments its momentum. If a directory is occupied, the new request (and the accompanying momentum) is forwarded to the transaction (T2) that currently occupies that directory. If T2 has already occupied all of its directories and moved on to steps 2 and 3, it sends a NACK to T1. If T2 is still trying to occupy its set of directories, it compares its momentum to that of T1 (currently 0) (in case of a tie, the core number is used as a tie-breaker). If T2 has a higher momentum, it sends a NACK to T1. On receiving a NACK, T1 will attempt to occupy the directory again, but with an updated value for its momentum. If T2 has a lower momentum, it will hand off occupancy of the directory to T1. T2 decrements its momentum, sends a message to the directory to update that T1 is the current occupier, and sends an ACK to T1 to indicate its occupancy of the directory.

Consider the following example: T1 and T2 both wish to commit to directories D1-D8. They send out parallel requests to all of these directories. T1 succeeds in occupying D1-D3 first and T2 succeeds in occupying D4-D8. When T1's request reaches D4, it is re-directed to T2 (with a momentum of 0). By now, T2 may have already updated its momentum to 5 and it NACKs this request. Similarly, T1 may NACK T2's requests for directories D1-D3. T1 and T2 will now again attempt to acquire directories D4-D8 and D1-D3, respectively, but with updated momentums of 3 and 5. When T2 receives T1's forwarded requests (with momentum 3), it again sends back NACKs. When T1 receives T2's forwarded requests (with momentum 5), it hands off control of those directories to T2. Thus, T2 will soon end up acquiring occupancy for all directories D1-D8 and will proceed with its commit. T1 will keep re-trying until it succeeds. Depending on delays on the network and how the momentum values get updated, it is possible that two transactions may go through a few exchanges and hand a directory back and forth to each other, but as we show below, forward progress is guaranteed.

If a transaction (T1) is waiting for another transaction (T2) to release a directory, it means that T2 has a higher momentum. If T2 is waiting for another transaction T3, then T3 must have an even higher momentum. Thus, there can be no cycle of dependences and no deadlock. To further show forward progress, consider the transaction with the highest momentum M in the system. When it sends a request out, this request will likely succeed, allowing the momentum to increase further. If the request fails, it means that some other transaction in the meantime has received positive acknowledgements for its requests and has built up a momentum M', that is higher than M. Thus, the highest momentum in the system is always monotonically increasing until a transaction ends up occupying all directories in its read- and write-sets and proceeds with commit. This guarantees forward progress.

While more complex, an important feature of this algorithm is that attempts to occupy the directories happen in parallel. Since this algorithm requires transactions to relinquish directory occupancies, the concept of momentum reduces the cost of re-acquiring occupancy and ensures forward progress. The algorithm continues to avoid any centralized resource.

The additional message requirements of this algorithm are difficult to capture with analytical equations – they are heavily influenced by the number of re-tries and the rate of growth of momentum.

In terms of related work, the momentum-based algorithm has similarities with contention management policies in software transactional memory systems proposed by Scherer and Scott [6].

4 Results

4.1 Methodology

For the preliminary analysis in this paper, we restrict ourselves to workloads that are synthetically generated. This also makes it easier to test scalability. The following additional simplifications/assumptions are made.

The N nodes in the system (each node has a processor and directory) are organized as a grid. We ignore processing delays at each node and are primarily concerned with the network delays imposed by each algorithm. Similarly, we do not model the cache coherence operations of sending invalidations to sharers of a cache line (these should be similar for

all the algorithms considered in this paper). The network delays are measured with a locally developed network simulator that models a grid topology with an adaptive routing mechanism and virtual channel flow control. Every uncontended hop on the network takes up two cycles of link delay and three cycles of router pipeline delay. When modeling the Stanford-TCC algorithm, we assume that the centralized TID vendor is co-located with a node that is in the middle of the grid.

The synthetic workload is generated as follows. Each of the N nodes is ready to commit a transaction after executing for a number of cycles that is randomly between $0.5 \times TL$ and $1.5 \times TL$, where TL is the average transaction length. The transaction has a read-set of fixed size R and a write-set of fixed size W memory blocks (cache lines). The directories corresponding to these memory blocks are chosen based on the following probabilities that attempt to mimic locality. A memory block is part of the local node with a probability P_L (varied between 90 and 95% in our experiments); it is part of a neighboring node with a probability P_N (varied between 4 and 9%); the memory block is part of a remote non-neighbor node with a probability P_R (1%). In each simulation, measurements are made over one million cycles and we report averages across three runs with different random number seeds.

4.2 Results

Figure 1 shows average commit latency per transaction for the two proposed algorithms and the baseline Stanford-TCC algorithm as a function of the number of nodes N. This experiment assumes that TL is 200, R is 16, W is 4, P_L is 95%, P_N is 4%, and P_R is 1%. Figure 2 models the same experiment as above, but reports the average number of required messages per transaction.

Even for the 16-node system, we observe that the sequential and momentum-based algorithms require much fewer messages than TCC. The number of messages required for the sequential algorithm remains constant as the nodes are increased, while the TCC algorithm's message requirements scale up linearly (as also demonstrated in the analytical equation derived in Section 2). The message requirements of the momentum-based algorithm grow at a much slower rate. As a result, we observe that the sequential algorithm always does better than TCC and the momentum-based algorithm always does better than sequential. For a 256-node system, relative to TCC, the momentum algorithm reduces the number of messages by 86% and the number of cycles by 58%.

These results are strongly influenced by the assumptions on locality. Figure 3 shows latency (left hand Y axis) and message requirements (right hand Y axis) as we vary the lo-

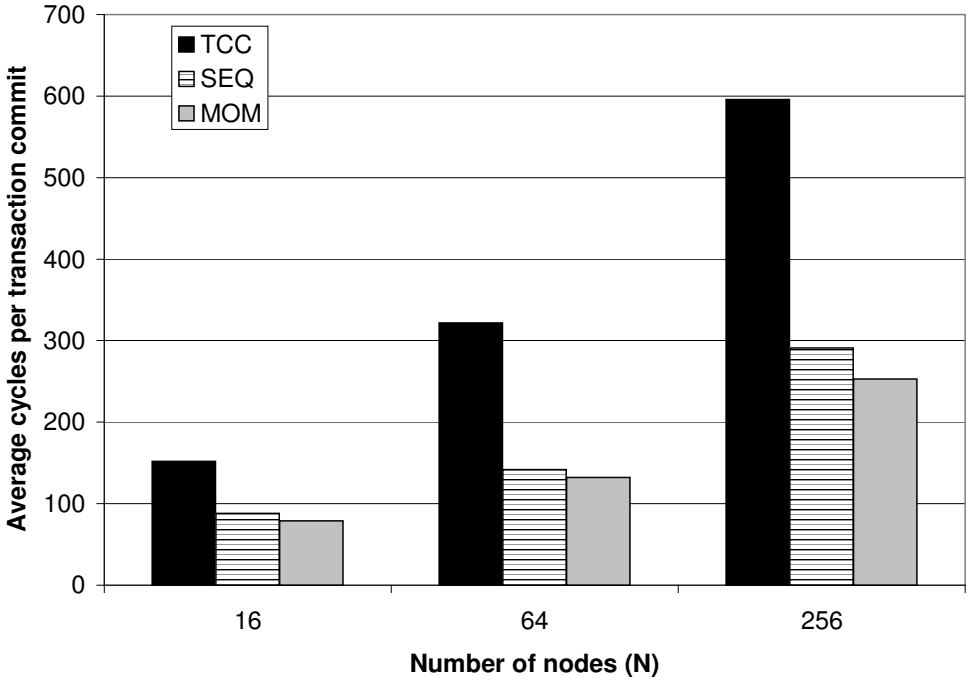

Figure 1: Performance of the three algorithms as N scales up.

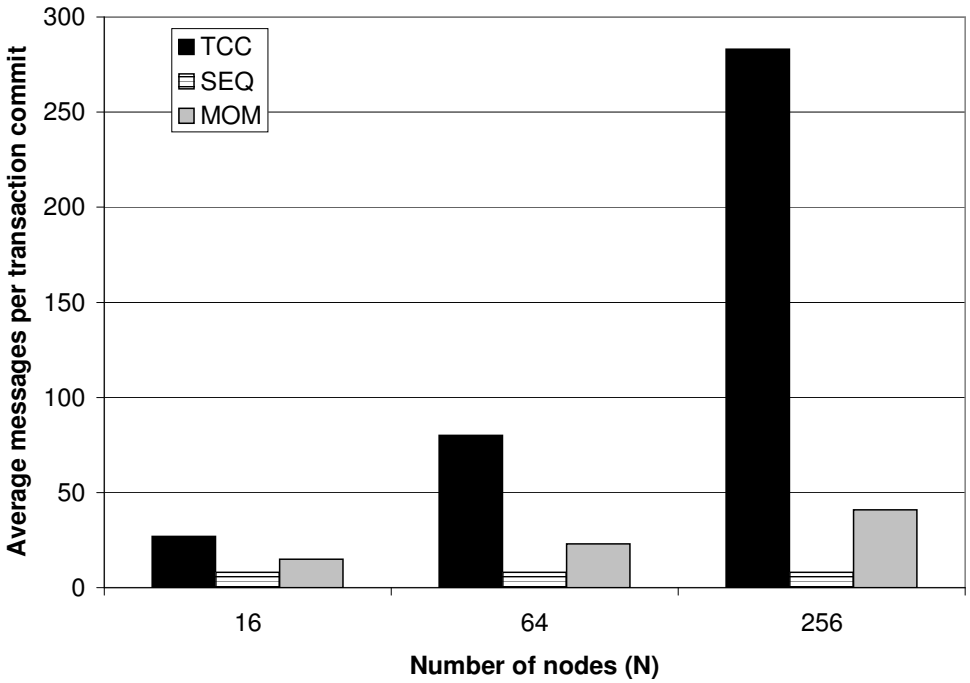

Figure 2: Message requirements.

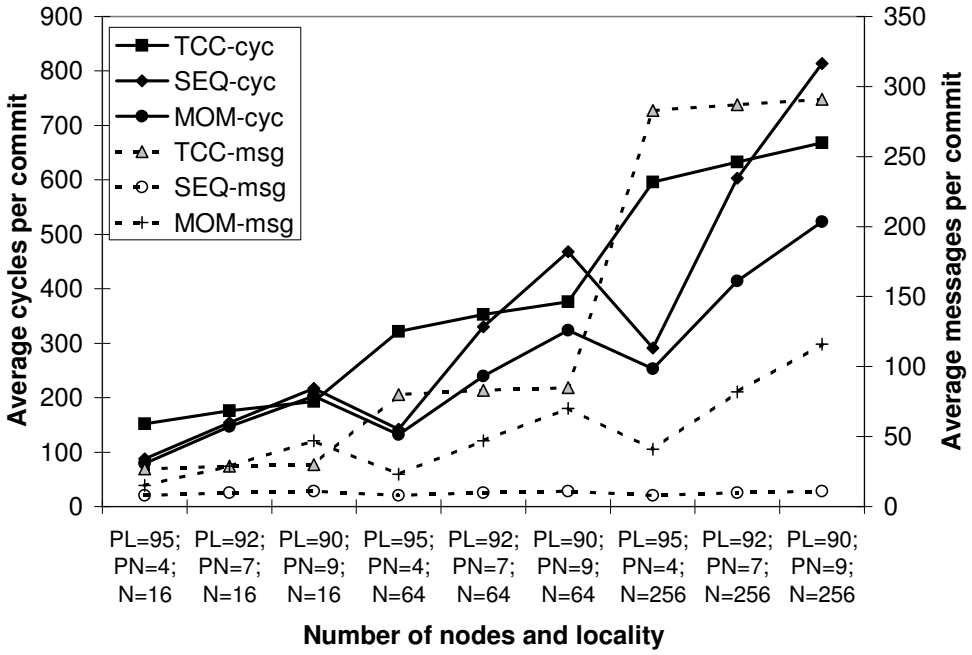

Figure 3: Cycles and messages required as a function of locality and N.

cality parameters (P_L, P_N) and the number of nodes N. If the application has less locality, the behavior of all three algorithms degrades significantly. Sequential can perform worse than TCC because a conflict at a directory prevents it from making forward progress on all higher numbered directories (unlike the behavior in TCC). Similarly, the momentum-based algorithm can also suffer from many messages and delays if conflicts cause a directory to repeatedly exchange occupancies between transactions. We are currently considering algorithm variations to deal with this problem. However, even with poor locality, a 256-node system yields best performance with the momentum algorithm. These results indicate that TCC degrades more gracefully as locality worsens, but the proposed algorithms degrade more gracefully as the number of nodes increases.

5 Conclusions

This paper introduces novel algorithms to commit transactions in a scalable manner. The proposed algorithms are deadlock- and livelock-free and do not employ any centralized resource. While Algorithm 1 requires the least number of messages, it can suffer from longer delays. The momentum-based algorithm introduces speculation and stealing in an attempt to reduce delays while increasing the number of messages. We show that relative to the Stanford-TCC baseline, the average number of messages can be reduced by as much as

97% (by Algorithm 1) and the average delay can be reduced by as much as 59% (by Algorithm 2). For future work, we plan to extend our analysis in several ways, most notably by considering real transactional workloads and by quantifying the impact on power consumption. Variations to the momentum-based algorithm will also be considered, similar to the contention management policies proposed by Scherer and Scott for software transactional memory systems [6].

References

[1] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of HPCA-11*, February 2005.

[2] H. Chafi, J. Casper, B. Carlstrom, A. McDonald, C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable Non-Blocking Approach to Transactional Memory. In *Proceedings of HPCA-13*, February 2007.

[3] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[4] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-Based Transactional Memory. In *Proceedings of HPCA-12*, February 2006.

[5] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of ISCA-32*, June 2005.

[6] W. Scherer and M. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of PODC*, 2005.