Scalable, Reliable, Power-Efficient Communication for Hardware Transactional Memory

Seth H. Pugsley, Manu Awasthi, Niti Madan, Naveen Muralimanohar, Rajeev Balasubramonian

UUCS-08-001

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

January 25, 2008

Abstract

In a hardware transactional memory system with lazy versioning and lazy conflict detection, the process of transaction commit can emerge as a bottleneck. This is especially true for a large-scale distributed memory system where multiple transactions may attempt to commit simultaneously and co-ordination is required before allowing commits to proceed in parallel. In this paper, we propose novel algorithms to implement commit that are more scalable (in terms of delay and energy) and are free of deadlocks/livelocks. We show that these algorithms have similarities with the token cache coherence concept and leverage these similarities to extend the algorithms to handle message loss and starvation scenarios. The proposed algorithms improve upon the state-of-the-art by yielding up to a 7X reduction in commit delay and up to a 48X reduction in network messages. These translate into overall performance improvements of up to 66% (for synthetic workloads with average transaction length of 200 cycles), 35% (for average transaction length of 1000 cycles), 8% (for average transaction length of 4000 cycles), and 41% (for a collection of SPLASH-2 programs).

1 Introduction

Transactional Memory (TM) [23] is viewed as a promising approach to simplify the task of parallel programming. In a TM system, critical sections are encapsulated within transactions and either the software or hardware provides the illusion that the transaction executes atomically and in isolation. Many recent papers [2, 9, 29, 35] 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 designs.

An HTM system is typically classified by its choice of versioning and conflict detection mechanisms. For example, the Wisconsin LogTM implementation [29] 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 [9], 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 [9], 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. In fact, a recent paper [3] describes how either system (and other flavors of HTM) can lead to performance pathologies for various code patterns.

Prior studies have shown that for most benchmark suites, more than half the transactions contain fewer than 200 instructions [10]. The state-of-the-art commit process in TCC [9] requires at least four (mostly serialized) round-trip messages and on a communication-bound system, this delay can represent a sizeable fraction of the total transaction execution time (20-35% for benchmarks with short transactions [9]). Since the Stanford TCC approach to HTM is among the front-runners and since commit scalability is one of the bottlenecks for that system, this paper focuses on improving the scalability of the commit process for that design.

We propose novel algorithms that significantly reduce delay, are free of deadlocks/livelocks, do not employ a centralized agent, do not produce new starvation scenarios, and significantly reduce the number of network messages (and associated power), relative to the

Scalable-TCC implementation. These algorithms are more scalable because the message requirement is not a function of the number of nodes in the system. We design a basic simple algorithm that requires very few network messages and then propose a few performance optimizations. We also show that the basic algorithm has strong similarities with the token cache coherence concept [25], allowing us to leverage existing mechanisms to handle starvation and message loss.

In Section 2, we provide details on the state-of-the-art Scalable-TCC implementation and identify inefficiencies in its commit algorithm. Sections 3 and 4 describe our proposed algorithms and their reliable versions. Evaluation results are discussed in Section 5, related work in Section 6, and conclusions in Section 7.

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. 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. [9] attempt to provide scalable parallel commits in a large-scale multiprocessor system. The following baseline 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[1]. The problem with allowing multiple

[1]A similar platform is also meaningful for a single multi-core processor. In such a multi-core, each core has a private L1, and the large shared L2 maintains a directory to ensure coherence among L1s. The large L2 may be banked and distributed across the chip, so each core has one L2 bank in close proximity and L2 requests are routed to the appropriate bank based on the index bits (just as requests are routed to the appropriate memory and directory based on the address in a distributed shared-memory multiprocessor).

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 Scalable-TCC 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. The directory can service the next TID after receiving ACKs for all the invalidates.

To summarize, the above algorithm first employs a centralized agent to impart an ordering on the transactions. Two 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 contact the centralized agent and a transaction is never forced to wait upon a transaction with a higher TID.

The total number of messages required per commit (not including the invalidates and ACKs sent by the cache coherence protocol) equals

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

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 [9] and may not scale up with N if the application has good locality.

There are many inefficiencies in this algorithm. 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. It is well-known that on-chip communication is a huge bottleneck for performance and power, especially when packet-switched grid networks with bulky routers are required to support high-bandwidth communication [13, 20]. 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).

A simple analysis shows that for the Scalable-TCC algorithm, even under the best conditions, a single transaction commit must endure four (mostly) sequential round-trip messages on the network (not counting the invalidation and acknowledgment messages originating from the written cache lines). If the application has high locality, three of these round-trips (write-probe, read-probe, and commit) need not travel far (optimistically assuming that skip messages to distant nodes are not on the critical path and there are no re-tries), while one of the round-trips (obtaining the TID) must travel half-way across the network on average. Projections show that such on-chip delays can exceed 50 cycles at future technologies [31, 38]. If the multiprocessor system is composed of multiple multi-core chips, contacting the centralized TID vendor may require off-chip access and potentially many hundreds of cycles. Thus, the commit process on average can take of the order of 100 cycles (our detailed simulation results show that this number is in the range of 57-671 cycles). This is a huge overhead, given that 50% of all transactions are less than 200 instructions long in many benchmark suites [10]. The latency and power cost of the commit process is less of a bottleneck if applications employ large transactions. As shown in [9], the Scalable-TCC commit algorithm imposes low performance overheads for most benchmarks with large transactions, but accounts for 20-35% of 64-node execution time for applications with smaller transaction sizes (*volrend, equake, Cluster GA*). It is hard to predict if future transactional workloads will be dominated by short transactions (and if the commit process will correspondingly represent a huge bottleneck), but analyses of existing multi-threaded applications indicate that most transactions are short [10]. We therefore believe that the commit problem in large-scale multiprocessors employing lazy versioning/conflict-detection is worth closer attention. To better understand the relationship with transaction size, our analyses of synthetic workloads in Section 5 show behaviors for a range of average transaction lengths. Further, with interconnect/router power emerging as a major bottleneck [13], it is important to improve upon algorithms with $O(N)$ message requirements.

3 Scalable Commit Algorithms

We next propose commit algorithms that avoid a centralized resource and have message requirements that do not directly scale up with the number of nodes. Note that we are preserving most of the Scalable-TCC architecture except the algorithm (Steps 1-5 in the previous Section) that determines when a transaction can make its writes permanent. 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. [9], 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.

3.1 Basic Algorithm: Sequential Commit (SEQ)

We introduce an *"Occupied"* bit in each directory 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 acknowledgment from a 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 the directory receives all ACKs for its invalidations, it re-sets its Occupied bit. As part of Step 2, the transaction also sends Occupancy Release messages to directories in its read-set.

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

$$2(w + r) + W + r + PR' = W + 2w + 3r + PR',$$

where PR' equals the number of re-tries because of lack of buffer space.

The SEQ algorithm can suffer from transaction starvation. If two transactions conflict, it is possible that one of them always occupies the smallest-numbered conflicting directory first (because of proximity to the directory or because it is simply a shorter-running transaction) and aborts the other. These problems can arise in any lazy HTM system, especially if frequent short transactions repeatedly conflict with a large transaction (referred to as the Starving Elder pathology in [3]). Even in Scalable-TCC, a transaction, that consistently beats another transaction to the centralized vendor, will end up starving the latter. The probability of starvation should be similar in both algorithms as starvation in both cases is determined by the consistent outcome of a single race: who reaches the lowest-numbered common directory (SEQ), or who reaches the centralized vendor (Scalable-TCC). Additional mechanisms are required to handle such situations. Typically, an aborted transaction must release the directories that it has occupied and it must take itself out of the buffer of the directory it is currently waiting upon (just as an aborted transaction in Scalable-TCC relinquishes its TID). If a transaction realizes that it has aborted S successive times, it signals starvation and does not relinquish its position in the directories and buffers. It will eventually occupy all of its directories and succeed. This has the same effect as the starvation mechanism in the Scalable-TCC algorithm, where a starved transaction does not relinquish its TID on an abort and eventually becomes the oldest transaction.

It is worth reiterating that we are making no other changes to the Scalable-TCC architecture apart from the algorithm for commit. Checks for data conflicts between transactions are not being negatively impacted: in fact, if transactions can proceed sooner with propagating their writes, conflicts will be detected sooner and the aborted transaction wastes less power and cycles.

3.2 Optimizations to the Basic Algorithm

There are a few inefficiencies in the basic SEQ algorithm that we attempt to address in this sub-section.

3.2.1 Parallel Reader Optimization – SEQ-PRO

The simple SEQ algorithm does not make a distinction between directories in the read and write sets. Each directory must be occupied sequentially, with no commit parallelism at a given directory. However, if two transactions only have reads to a given directory, there is no possibility of a conflict at this directory. These two transactions can simultaneously occupy the directory and safely proceed with the rest of their commit process. In other words, we can allow multiple transactions to simultaneously occupy a directory as long as none of these transactions write to this directory. Note that this optimization is also deadlock-free as all transactions proceed to occupy directories in a sequential order.

This optimization (SEQ-PRO) entails minor overheads. It does not entail any additional messages, but we now need separate *Read Occupied* and *Write Occupied* bits with their own buffers. We must make sure there is only one write bit occupier or any number of read bit occupiers (and each of the occupiers must be tracked). We also need policies to handle new requests. New read-occupancy requests are always granted if the Write Occupied bit is not set and there are no waiting writers, else buffered (this prevents writers from being starved). New write-occupancy requests must be buffered if either of the bits is set, else granted. When a Read or Write Occupied bit is released and there are waiting readers and writers, we give priority to the writers unless the number of waiting readers exceeds a threshold. We observed that our results are not very sensitive to the choice of policy/thresholds.

3.2.2 Occupancy Stealing with Timestamps – SEQ-TS

While the SEQ algorithm helps reduce the total number of required messages, the delay may be higher than that of the scalable-TCC algorithm because each of the directories must be occupied in sequential order. To remove the dependence on this sequential process, we propose the following timestamp-based commit algorithm (SEQ-TS). Transactions attempt to occupy directories in parallel and a transaction is allowed to steal a directory from another transaction if it is deemed to have higher priority. Priority is determined by age: every occupancy request is accompanied by the timestamp of when the transaction began its commit process. For such an algorithm to work, we need every core to participate in a distributed clock algorithm [22] so they have an approximate notion of relative time[2]. Low-overhead forms of such distributed clocks are also employed for conflict resolution in other flavors of HTM [29, 34] and for fault-detection [28].

In this algorithm, transaction T1 sends out parallel requests to occupy directories in its read- and write-sets. If a directory is occupied by a transaction (T2) with a younger timestamp (the core number is used as a tie-breaker), the new request is forwarded to the transaction (T2) that currently occupies that directory (else, the new request is buffered). If T2 has already occupied all of its directories and moved on to Step 2 (of the SEQ algorithm), it sends a NACK to T1 and T1 tries again later. If T2 is still trying to occupy its set of directories, it will hand off occupancy of the directory to T1. T2 sends a message to the directory to update that T1 is the current occupier and to place T2 at the head of the buffer, and sends an ACK to T1 to indicate its occupancy of the directory.

Such a mechanism always guarantees forward progress for the oldest transaction. Even though directories are being occupied out of order, deadlocks cannot happen. A transaction can only be waiting on an older transaction or on a transaction that has already moved to Step 2, eliminating the possibility for a cycle of dependences. Starvation may still occur: the oldest transaction may be unable to steal directory occupancy from a short transaction that quickly occupies its directories and moves on to Step 2. If this happens frequently enough, T1 can signal starvation and issue a request that places it at the head of the buffer.

While more complex, an important feature of this algorithm is that attempts to occupy the directories happen in parallel. As we show in the next section, the correspondence with the token coherence concept implies that complex algorithms can be correctly built without worrying about various corner cases, as long as specific invariants are preserved.

3.2.3 Other Optimizations

We also experimented with two other variants of the SEQ algorithm. In the first variant, a transaction optimistically sends out occupancy requests in parallel. If any of these requests fail to occupy a directory, the transaction falls back upon the sequential process (while relinquishing any directories that it may have occupied out-of-order). In the second variant,

[2]In short, every message send/receive increments the local clock, the clock value is piggy-backed on outgoing messages, and the local clock is set to the clock value received in a message if it is higher than the local clock value [22].

we use the notion of “momentum” instead of the timestamp – priorities are given to transactions that have occupied more directories (a transaction’s momentum). It is also possible to construct other algorithm variants similar to the analysis by Scherer and Scott for software TM contention managers [37]. Both of the above variants did not yield good results because they either frequently fall back upon the sequential process (variant 1) or because they result in directory occupancies frequently changing hands (variant 2). We also do not attempt optimizations similar to the MARK messages in the scalable-TCC algorithm. In our basic SEQ algorithm, write addresses are sent to each directory only after all directories are occupied. Instead, we can send write addresses to a directory right after it has been occupied so that the final commit message is short. In our initial experiments, we observed that this results in minor savings, while increasing buffering complexity, especially for the optimized algorithms. Hence, we do not discuss the above variants any further in this paper.

There are also ways to start with the Scalable-TCC algorithm and alternatively reduce the number of required messages and complexity. For example, in Step 1 of Scalable-TCC, 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 a 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 approach increases the bandwidth requirements in and out of the centralized agent. Similar to the approach outlined for the BulkSC system [8], 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 the single biggest source of long delays in the commit process – therefore, we do not further consider this option. As suggested in [9], a multicast network will help reduce the overheads of having to send skip messages to every node in Scalable-TCC. Multicast can cause the energy requirement[3] to reduce from $O(N^2)$ to $O(N)$ (still a function of N), but the latency requirement remains $O(N)$.

4 Designing Reliable Protocols

Due to shrinking transistor sizes and lower supply voltages, soft error rates in computer systems are rising [30, 39]. Detection of such an error within the processing logic in a router may result in a packet being dropped. The error rates in wire transmission are also increasing because wire dimensions and wire spacing are shrinking as well, making them

[3]These analyses apply to a ring. For a grid, the multicast complexity is $O(N)$ and $O(\sqrt{N})$ for energy and delay, respectively.

Message lost	Effect	Solution
Directory Occupy Request	Stalls the requesting transaction	Transaction timeout and retry
Directory Grant Response	Stalls every transaction dealing with that directory	Transaction timeout and retry: create a new occupancy bit with incremented counter
Directory Release	Stalls every transaction dealing with that directory	Directory timeout and probe: create a new occupancy bit with incremented counter
NACK	Stalls the requesting transaction	Transaction timeout and retry

Table 1: Problems caused by message loss in the Sequential commit algorithm.

increasingly susceptible to high energy particles, noise, and interference from neighboring channels [1, 14, 17, 18, 42]. Traditionally, communication buses have employed some form of redundancy such as ECC to detect and recover from errors. However, due to the proximity of wires within a communication channel, multiple bits can be simultaneously corrupted and even ECC cannot aid in recovery from many such multiple-bit errors. When such an error is detected, the received packet must be dropped[4]. Examples of recent efforts to address these problems include [15, 19, 32].

Within a cache coherence protocol, message losses can be catastrophic, resulting in deadlocks and even permanent data loss (for example, when the packet contains a dirty cache line that is undergoing writeback). Therefore, it is important that on-chip communication protocols also be designed to handle message loss. This is a problem that is starting to gain prominence [6, 16, 28, 40].

HTM protocols such as the ones described in previous sections, have two layers of messages: the first layer that determines when a transaction can proceed with making its writes visible and the second layer that relies on the cache coherence protocol to actually make the writes visible. In this work, we do not worry about error resilience within the cache coherence protocol – that issue is orthogonal to the problem of error resilience within the HTM commit algorithms. Techniques to make the cache coherence protocol error-resilient can be based on the work in [16, 28].

4.1 Augmenting the Protocol to Handle Message Loss

Message losses are equally problematic in the Scalable-TCC and Sequential commit algorithms. In this section, we simply focus on problems created in the Sequential algorithm and how they can be addressed. We later show how variants of the Sequential algorithm are especially amenable to error resiliency.

[4]Alternatively, the sender can buffer every outgoing packet and re-send the packet if the receiver detects an error and sends a NACK. This would require an ACK for every valid packet transfer so that the buffers at the sender can be freed. We expect this approach (buffering and ACKing every packet) to have inordinate power overheads.

Table 1 lists the problems caused by message loss as well as the proposed solutions to these problems. We start by addressing the basic SEQ algorithm with no optimizations. We also assume for now that only a single message loss happens at a time (addressing multiple simultaneous message losses is left as future work).

The first message in the protocol (Table 1) is a request to occupy a directory. If this is lost, the requesting transaction does not make forward progress. The transaction will escape such a stall if it gets aborted. To avoid stalling indefinitely, the transaction must time-out and re-try its directory occupy request if the request is not serviced soon enough. If there was no message loss and the delays are being caused by long transactions, the directory simply drops any redundant occupancy requests issued by transactions. If the occupancy request message was not lost, but shows up much later because of traffic congestion, the directory may believe that the transaction has made two separate occupancy requests and may service both of them. If the transaction receives occupancy when it was not expecting it, it simply returns the occupancy back to the directory (with the Directory Release message). The timeout window can be periodically adjusted to be much larger than average occupancy wait times, as measured by the directories.

If a Directory Grant Response is lost, the requesting transaction does not make forward progress and prevents forward progress for every other transaction that deals with that directory. The same timeout mechanism as above can be adopted by the requesting transaction. If the directory receives a request for occupancy from a transaction that it already believes is the occupier, it can be concluded that the grant response was either lost or stuck in the network. The directory can respond again with occupancy permissions. However, this can lead to a problem if the initial response was simply stuck in the network and is eventually delivered to the requester. This may lead to a situation where two different transactions believe they have occupancy of the directory. This must not be allowed. Hence, the directory occupancy bit must have an associated counter (even a one-bit toggle counter may suffice for low error rates) to indicate how many times the directory believes the bit was lost. This counter is shipped with every grant response and all cores must be informed when the counter is incremented. This enables a transaction to drop an earlier grant response that may have been stuck in the network. This process is very similar to the token re-creation process in the reliable token cache coherence protocol described in [16] and we expand more on that analogy in the next sub-section. We do not reproduce the minor implementation details [16] of that process here for space reasons.

The other important message in the most basic sequential protocol is the Directory Release message sent by the transaction after it has occupied all directories in its commit set. If the Directory Release message is lost, it stalls every subsequent transaction dealing with that directory. This can be solved with a process similar for lost Directory Grant Responses. The directory times out and sends a probe to the transaction that it believes is the occupier. The transaction will possibly detect an inconsistency (it is either waiting to occupy that directory or does not deal with that directory at all) and respond to the directory. At this point, the directory will re-create an occupancy bit with an incremented counter and proceed. If the initial Directory Release message was simply stuck in the network and is eventually delivered, the directory drops the message because of the inconsistent counter.

The only other message introduced by the basic sequential protocol is the NACK sent by the directory if the buffer is too full to accommodate a directory occupancy request. If this message is lost, the behavior is exactly as if the directory occupancy request is lost.

Note that we have not discussed the other messages required for a transaction's commit: the messages that carry the written addresses/data and the invalidations and acknowledgements corresponding to these blocks. These messages are part of the baseline cache coherence protocol. A separate solution (such as those described in [16, 28]) is required to handle message loss in the cache coherence protocol and is orthogonal to the focus of this paper (the algorithm to determine when a transaction can proceed with propagating its updates)[5].

The above changes to the protocol do not introduce any additional traffic when there is no message loss (assuming that the timeout window is long enough and there are few false alarms). The only performance degradation introduced is the time taken to detect message loss and messages arising out of false alarms (these are both functions of the timeout window). If we incorporate an adjustable timeout window, additional messages are required to communicate the new timeout value to the cores: this is not a critical message and will not lead to correctness issues if lost.

4.2 Analogy with Token Coherence

The concept of *token coherence* was introduced by Martin et al. [25] as an abstraction to build reliable and high-performance cache coherence protocols. In [25], the authors show that a cache coherence protocol is correct if it fulfils a few basic invariants:

[5]The only caveat is that a transaction does not wait for ACKs for its writes before moving on to the next transaction: this is not a violation of the sequential consistency model because any transaction that reads a stale copy of the line will anyway abort. If the write notification messages are lost, the transaction may clear its write bits without realizing this. Thus, an error-resilient cache coherence protocol for a transactional system may impose greater overheads (an ACK from the directory to the transaction after commit), but this overhead is entailed regardless of the commit algorithm (Scalable-TCC or SEQ).

- Each block has T tokens in the system ($T \geq$ number of processors).
- A processor can write a block only if it holds all T tokens for that block.
- A processor can read a block only if it holds at least one token for that block.
- If a coherence message contains one or more tokens, it must contain data.

Complex protocols can be built upon these basic invariants to improve performance. In other words, processors can employ all kinds of complex algorithms to quickly acquire the tokens they require. These algorithms need not be provably correct (meaning, some of the messages of that algorithm can even be lost as long as they do not contain tokens), but the protocol will still behave correctly if the invariants are satisfied. To avoid starvation, the authors also describe the use of *Persistent Requests* to acquire tokens. The work in [16, 28] assumes a baseline token cache coherence protocol and then designs the most basic mechanisms required to handle loss of messages containing tokens. This abstraction allows the authors to determine the minimum protocol changes required for error resiliency in a cache coherence protocol: all other messages in a cache coherence protocol not involving tokens can be viewed as the "performance substrate" that need not always be correct.

A neat side-effect of the transactional commit algorithms designed in this paper is its correspondence with the token coherence concept. This gives us the option to leverage existing results on token coherence [4, 5, 12, 16, 26, 27, 28] in interesting ways. While the token cache coherence abstraction has these many nice properties, there are some road-blocks to its direct implementation in hardware. For example, a clean block cannot be silently evicted from a cache (as is commonly done in modern protocols) – the tokens for that block must be sent back to the directory. Luckily, these negative features are not present when this abstraction is applied to the transaction commit problem.

Consider the following description of the basic sequential commit algorithm using tokens. Each directory has a single token. If a transaction has C directories in its commit set, it can proceed with propagating its writes only after it has acquired the C tokens corresponding to these directories. These are the basic invariants required for correct operation. Upon this basic framework, we can include persistent requests so transactions do not starve. We can add other mechanisms (such as the timestamp and momentum-based approaches described in the previous section) to accelerate the collection of tokens. We can leverage the theory developed in [16] to determine the minimum mechanisms required to handle message loss. In essence, messages involving token loss need careful re-creation of tokens with incremented counters, while other message losses can be typically handled with timeouts and re-tries. This corresponds very closely with the solutions developed in the previous sub-section: only the Directory Grant Response (that transfers a token from directory to transaction) and the Directory Release message (that transfers a token from transaction to directory) need careful handling in case of message loss. That work also addresses loss of persistent requests.

In this paper, we do not discuss error resilience for the other optimized algorithms described in Section 3. But the work in [16] and the similarities with token cache coherence make us believe that similar solutions exist. The multiple-reader, single-writer optimization described in Section 3 is even more in tune with the token coherence concept. The timestamp-based stealing of occupancies has a process similar to handing off write ownership to a cache line in a cache coherence protocol. We expect this synergy with token coherence to lead to interesting future results. It is especially noteworthy that the token abstraction does not introduce additional messages for the commit process (unlike the token cache coherence case where additional messages are required when evicting clean cache blocks). The overheads of token handling are much lower as well: only one token per directory (instead of one per cache line as in the token cache coherence case). Such a correspondence with token coherence does not exist for the Scalable TCC algorithm because of its reliance on Transaction IDs.

5 Results

5.1 Synthetic Workloads

Methodology

In our evaluations in this sub-section, we focus on synthetically generated commit requests. The N nodes in the system (each node has a processor and directory) are organized as a grid. We are primarily concerned with the network delays imposed by each commit algorithm. For the evaluation in this sub-section, 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). We also do not model data conflicts and aborts: note that the commit algorithms themselves should not impact the probability of data conflicts (and this probability is small anyway). In fact, aborts are handled quicker if the commit process is shortened, so this approximation marginally penalizes the faster algorithms. 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. To reflect a modern on-chip network, we assume that every uncontended hop on the network takes two cycles of link delay and three cycles of router pipeline delay [31]. We also show results with other delay assumptions. Each of the routers employs 3 virtual channels per physical channel. When modeling the Scalable-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[6]); 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.

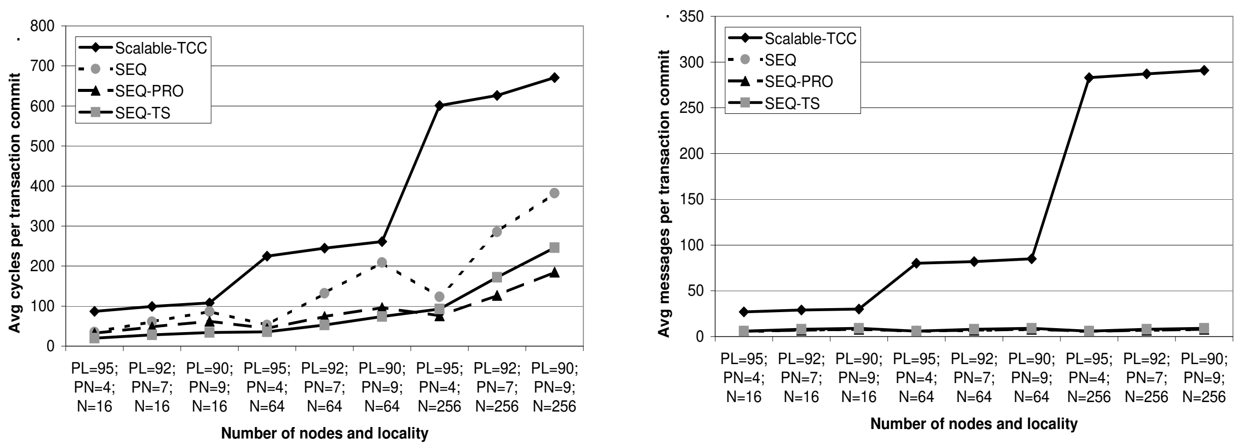
In the next sub-section, we show results for a more complete simulation with cache coherence, data conflicts, and real benchmark traces, but we believe that synthetic workloads provide more insight and a more direct comparison between the commit algorithms. The synthetic workloads allow us to easily change parameters (locality, transaction size, commit set size, number of nodes) and understand the factors influencing behavior. This is especially relevant because most available benchmark applications have not been designed or tuned to run on 64-256 nodes and the landscape of transactional workloads of the future is not clear. Also, the locality behavior of real applications will likely be a strong function of page mapping policies[7]. In spite of these problems, simulations of real workloads are also admittedly useful because they provide samples of behavior in existing programs. We have also augmented our simulator to model cache coherence, data conflicts, aborts, and the simulator reads in traces of SPLASH-2 programs generated with SIMICS. For the SPLASH-2 programs analyzed, average transaction lengths vary between 12 and 606. Our experiments with synthetic workloads therefore focus on transaction lengths in this range and the probability numbers above (P_L, P_N, P_R) generate commit sets that closely match the averages in these examined workloads.

Results

Figure 1(a) shows average commit latency per transaction for the three proposed algorithms and the baseline Scalable-TCC algorithm as a function of the number of nodes N and locality. In these experiments, locality is varied from high ($P_L = 95\%$ and $P_N = 4\%$) to low ($P_L = 90\%$ and $P_N = 9\%$) and N is varied from 16 to 256. This experiment assumes

[6]These probability ranges yield commit set sizes similar to those shown for the workloads in [9].

[7]First-touch page mapping policies often exhibit high locality for short simulation studies, but may not be representative of long-running applications where threads can migrate, working sets per thread can change, etc.



(a) Average number of cycles per transaction

(b) Average number of messages per transaction

Figure 1: **Average number of cycles and messages per transaction commit for various algorithms as a function of N and locality.**

that TL is 200, R is 16, W is 4, and P_R is 1%. Figure 1(b) models the same experiment as above, but reports the average number of required messages per transaction.

It is quickly evident from Figure 1(b) that the message requirement for the Scalable-TCC algorithm scales up linearly with N because of the need to send skip messages to every node. Even with multicast support, the energy requirements of the skip operation would increase linearly with N. All three proposed algorithms have message requirements that remain constant as N is varied, hence the claim of better energy scalability than TCC. As locality is reduced, the number of messages for each algorithm increases slightly. The average message requirement of SEQ-TS is about one more per commit than the message requirements of SEQ and SEQ-PRO.

The latency results in Figure 1(a) show the poor delay scalability of Scalable-TCC, primarily because of the longer delays to access a centralized agent and because a directory cannot advance to subsequent transactions unless it receives skip messages from (frequently) distant cores. As expected, the commit delay roughly doubles when the number of nodes is quadrupled (average distances on a grid are a function of $\sqrt{N}$). The SEQ algorithm performs much better than Scalable-TCC when locality is high. As N is increased from 16 to 64 to 256, the delay increases from 35 to 53 to 123 cycles. While the delays are much less than Scalable-TCC and even though the message requirement does not scale up with N, the delays for SEQ also appear to scale as square-root of N. Unlike Scalable-TCC, SEQ serializes transactions even when they only have read conflicts at a directory. Hence, the probability of conflicts and waiting delays increase as the number of transactions increases. The problem is exacerbated by the fact that transactions are only 200 cycles long on average. If we increase average transaction length TL to 4000 cycles (discussed subse-

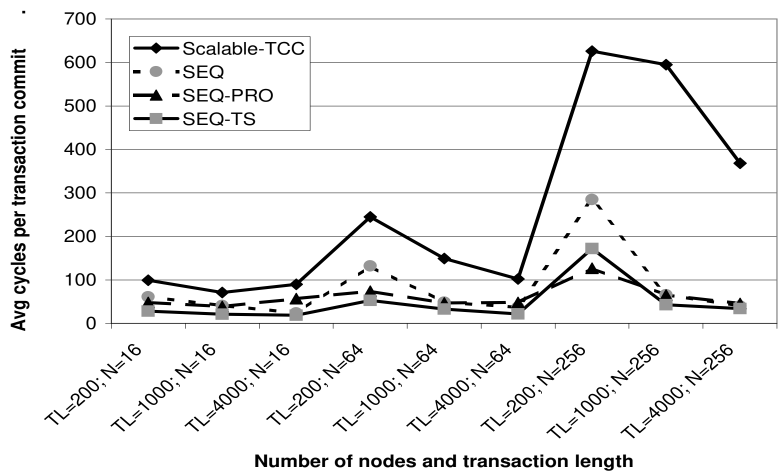


Figure 2: **Average number of cycles per transaction commit for various algorithms as a function of N and transaction length TL (P_L=92% ; P_N=7% ; P_R=1%).**

quently), the commit delay for SEQ increases from 33 cycles (16 nodes) to 42 (64 nodes) to 43 cycles (256 nodes). This is a much slower rate of increase than the corresponding numbers for Scalable-TCC (57 to 106 to 354 cycles), indicating better delay scalability for SEQ. However, SEQ degrades less gracefully when locality worsens (as can be seen by comparing the slopes of the curve segments). This is again related to SEQ's inability to handle read conflicts at a directory in parallel and the fact that multiple directories must be occupied sequentially. These problems are fixed by SEQ-PRO that allows parallel readers at a directory and by SEQ-TS that allows parallel requests for directory occupancies. Both of these algorithms result in large improvements, especially when locality is poor. These algorithms also scale better than Scalable-TCC, even for small transaction sizes (SEQ-PRO's delay increases from 32 to 44 to 76 cycles as N goes from 16 to 64 to 256).

For the mid-locality case and 64 nodes, SEQ, SEQ-PRO, and SEQ-TS reduce the commit delay by 46%, 70%, and 78%, respectively. If the transaction length is only 200 cycles, Scalable-TCC requires as many as 245 cycles for commit (a huge overhead). Hence, SEQ-TS's 78% improvement in commit delay translates to an impressive 43% improvement in overall performance. As transaction lengths are increased, the commit delays contribute less to overall performance (Amdahl's Law and fewer simultaneously competing transactions). This is reflected in Figure 2, where we show delays for commit for the mid-locality case as N and TL are increased[8]. For the mid-locality case and 64 nodes and an average transaction length of 4000 cycles, the impressive commit delay improvements for the proposed algorithms only translate into at most 2% overall performance improvements. For the 256-node case and a transaction length of 4000 cycles, the best proposed algorithm yields

[8]We also increased the commit set size in tandem with transaction length. The results were not very sensitive to commit set size.

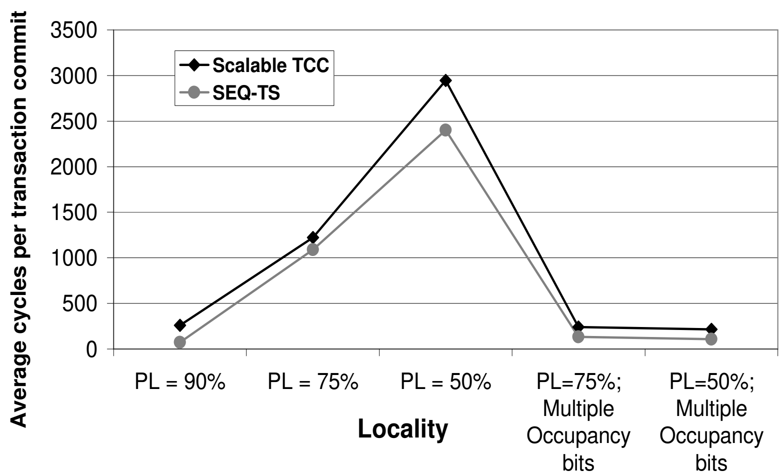


Figure 3: **Average number of cycles per transaction commit for various algorithms as a function of locality P_L for a 64-node system and transaction length TL = 200.**

a maximum overall improvement of 8%. This number is 35% for an average transaction length of 1000 cycles[9]. Note that irrespective of transaction length, the SEQ algorithm continues to reduce the message requirement by 48X for the 256-node cases.

While the number of messages is a useful comparison metric, the network energy requirements are dictated by the number of messages and the distance they travel. Since Scalable-TCC requires sending skip messages to every node, its messages travel much further than those of the SEQ algorithm. Therefore, network energy requirements of the proposed algorithms will be reduced by a factor much greater than 48X.

Sensitivity Analysis:

As a sensitivity analysis, we also evaluate the impact of network delays on overall performance. For the 64-node mid-locality case with $TL = 200$, the overall performance improvement of SEQ-TS over Scalable-TCC is 45%, 43%, and 31%, for network hop delays of three, five, and ten cycles, respectively.

Locality plays an important role in the behavior of commit algorithms and in the behavior of any multi-threaded application. Poor locality can impact our results in two ways. First, it increases the likelihood of conflicts at a directory. This issue can be alleviated by maintaining occupancy at a finer granularity (while incurring a cost in directory storage overhead). Second, directories in the commit set may be further away and that increases

[9]These numbers roughly agree with the results in [9], where benchmarks with 90% of transactions under 1100 instructions spent 20-35% of their execution time in the commit phase.

communication delays. To evaluate the effects of poor locality, Figure 3 shows average cycles per transaction commit as P_L is reduced from 90% to 75% to 50% (while also varying P_N from 9% to 13% to 25%). These results are for a 64-node system with TL of 200 cycles. To understand the contributions of the two effects listed above, we also carried out experiments with more occupancy bits per directory such that the probability of a conflict is the same as in the experiment with P_L of 90%. The SEQ-TS algorithm continues to out-perform Scalable-TCC by 19% for P_L = 50%. By having more occupancy bits per directory, the SEQ-TS algorithm degrades more gracefully as locality is worsened. However, it is clear that applications in a lazy HTM system will benefit greatly from smart page mapping policies that improve locality and this continues to be an important research area for the future.

5.2 Real Workloads

Methodology

For the experiments in this sub-section, we augment the simulator to model MSI based cache coherence, data conflicts, and aborts. We consider a subset of SPLASH-2 benchmarks that exhibit significant synchronization overhead [3]. The details of the benchmarks and their corresponding input sets are listed in Table 2. The Simics full system simulator running Linux on SPARC ISA is used to generate traces of the benchmarks [24]. Locks and unlocks in these programs are replaced with transaction begin and end statements. Other synchronization primitives are left untouched. We remove the overhead of locks and unlocks in the traces since a transactional workload need not wait for entering the critical section. A CPI of one is assumed for non-memory instructions and for memory instructions, delays within the cache hierarchy are modeled in detail. We continue to model network behavior for the commit algorithms in a cycle-accurate manner. We assume that each core has a 32 KB L1 cache and the 32 MB shared L2 cache is physically distributed among the cores. The programs were run to completion on 16 and 32 node systems.

Results

Table 2 summarizes the characteristics of the benchmark programs and their behaviors for the various algorithms. The abort percentage is less than 1% for all benchmarks. All benchmarks have average transaction lengths under 606 cycles and commit sets of average size less than 5. For a 16-core processor, SEQ performs worse than Scalable TCC by 34% whereas SEQ-PRO and SEQ-TS yield performance improvement of 26% and 40% respectively. The corresponding numbers for a 32-core processor are 39%, 26%, and 41%. While 32-node simulations are not enough to demonstrate scalability, they help validate the models chosen for the synthetically generated workloads.

Characteristic	Barnes	Ocean-Cont	Radiosity	Water-nsquared	FFT
Input size	16384 bodies	258	batch	512 mol	4096 points
Avg. Transaction length (cycles)	606	12	506	498	27
Avg. directories read	3	3	3	5	1
Avg. directories written	3	0.2	2	2	2
Avg. transaction execution time (cycles) for a 16-core processor					
Scalable-TCC	1721	425	1607	1288	419
SEQ	1862	1256	1139	2876	205
SEQ-PRO	1733	146	1081	888	205
SEQ-TS	844	681	691	894	150
Avg. commit messages per transaction for a 16-core processor					
Scalable-TCC	89	45	70	67	33
SEQ	13	9	10	15	5
SEQ-PRO	13	9	10	15	5
SEQ-TS	14	9	10	15	6
Avg. transaction execution time (cycles) for a 32-core processor					
Scalable-TCC	3052	806	1696	1761	2197
SEQ	4146	2584	1506	4276	713
SEQ-PRO	3322	290	1443	1257	713
SEQ-TS	1285	1429	795	1450	631
Avg. commit messages per transaction for a 32-core processor					
Scalable-TCC	179	55	80	91	115
SEQ	15	9	10	16	6
SEQ-PRO	15	9	10	16	6
SEQ-TS	15	9	11	16	8

Table 2: **Characteristics for SPLASH-2 programs.**

6 Related Work

While there is a vast body of recent work on hardware transactional memory, there is limited work that examines scalability aspects in a large-scale lazy versioning/conflict-detection HTM system. The prior work by Chafi et al. [9] is strongly based on the optimistic concurrency control algorithms proposed by Kung et al. for database systems [21]. The Bulk Disambiguation work by Ceze et al. [7, 8] attempts to reduce the overheads for conflict detection by compressing the read and write sets into signatures and either broadcasting them or sending them to a central arbiter. That work [8] also proposes a distributed arbiter mechanism (similar to the use of multiple directories in our work), but still relies on a central arbiter to aggregate information if a transaction (or chunk) deals with multiple arbiters. The solutions proposed here can be augmented with the signature mechanisms of Bulk to reduce the overheads of write propagation at commit time.

There are several other variants of HTM. Those that rely on eager conflict detection have better scalability because they overlap each individual check for conflicts with other computation [29, 34]. Lazy conflict detection implementations are stalled waiting for commit while these checks are collectively performed at the end of the transaction. But as is shown in [3], each HTM variant has its pitfalls. The work here addresses one of the pathologies for lazy systems (Serialized Commit) listed in that paper.

Thread-Level Speculation (TLS) has many architectural similarities with HTM, but deals with the speculative parallelization of a sequential program. As a result, there is an ordering enforced upon the threads. There have been efforts [11, 33, 41] to make these designs scalable, but the mechanisms are very different. Each thread is assigned a TaskID and constant-time commit mechanisms typically increment a last-commit shared variable [33]. As a result, conflict checks (that are not done at commit time) may incur higher overheads to identify the most recent versions of data. Renau et al. [36] have also explored microarchitectural optimizations to reduce the traffic and checking overheads in TLS.

7 Conclusions

This paper introduces novel algorithms to commit transactions in a scalable manner in a lazy versioning/conflict-detection HTM. The proposed algorithms are deadlock-free and do not employ any centralized resource. The message requirements of these algorithms do not vary with the number of nodes N. The delay for these algorithms does increase with N, but at a slower rate than the Scalable-TCC algorithm. We show that the algorithms have a flavor very similar to that of token cache coherence and previously published approaches for token coherence can be leveraged to prove correctness and handle starvation or message loss.

The proposed algorithms yield up to 48X reductions in message requirements for commit, relative to Scalable-TCC. The delay for the commit process can be reduced by up to 7X. The impact of this on overall performance is a strong function of transaction length. Overall performance can be improved by up to 66% for an average transaction length of 200 cycles, up to 35% for 1000 cycles and by an average of 41% for a collection of SPLASH-2 programs. For future work, we plan to continue improving the behavior of these algorithms (especially for many-core designs and applications with low locality) and to explore on-chip network power optimization strategies (for example, circuit switching to handle bulk transfers at the end of a transaction).

References

[1] K. Agarwal, D. Sylvester, and D. Blaauw. Modeling and Analysis of Crosstalk Noise in Coupled RLC Interconnects. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(5), May 2006.

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

[3] J. Bobba, K. Moore, H. Volos, L. Yen, M. Hill, M. Swift, and D. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of ISCA-34*, June 2007.

[4] J. Brown, R. Kumar, and D. Tullsen. Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures. In *Proceedings of SPAA-19*, June 2007.

[5] S. Burckhardt, R. Alur, and M. Martin. Verifying Safety of a Token Coherence Implementation by Parametric Compositional Refinement. In *Proceedings of VMCAI*, January 2005.

[6] J. Cantin, M. Lipasti, and J. Smith. Dynamic Verification of Cache Coherence Protocols. In *Proceedings of WMPI*, June 2001.

[7] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of ISCA-33*, June 2006.

[8] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of ISCA-34*, June 2007.

[9] 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.

[10] J. Chung, H. Chafi, A. McDonald, C. Minh, B. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of HPCA-12*, February 2006.

[11] M. Cintra, J. Martinez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of ISCA-27*, June 2000.

[12] B. Cuesta, A. Robles, and J. Duato. An Effective Starvation Avoidance Mechanism to Enhance the Token Coherence Protocol. In *Proceedings of PDP-15*, 2007.

[13] W. Dally. Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems (OCIN), 2006. Workshop program and report at `http://www.ece.ucdavis.edu/~ocin06/`.

[14] A. Deutsch. The Importance of Inductance and Inductive Coupling for On-chip Wiring. In *Proceedings of IEEE 6th Topical Meeting on Electrical Performance of Electronic Packaging*, October 1997.

[15] T. Dumitras, S. Kerner, and R. Marculescu. Towards On-Chip Fault-Tolerant Communication. In *Proceedings of ASP-DAC*, January 2003.

[16] R. Fernandez-Pascual, J. Garcia, M. Acacio, and J. Duato. A Low Overhead Fault Tolerant Coherence Protocol for CMP Architectures. In *Proceedings of HPCA-13*, February 2007.

[17] R. Ho. *On-Chip Wires: Scaling and Efficiency*. PhD thesis, Stanford University, August 2003.

[18] Y. Ismail and E. Friedman. *On-Chip Inductance in High Speed Integrated Circuits*. Kluwer Publishers, 2001.

[19] J. Kim, D. Park, C. Nicopoulos, N. Vijaykrishnan, and C. Das. Design and Analysis of an NoC Architecture from Performance, Reliability, and Energy Perspective. In *Proceedings of ANCS*, 2005.

[20] P. Kundu. On-Die Interconnects for Next Generation CMPs. In *Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems (OCIN)*, December 2006.

[21] H. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[22] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21:558–565, 1978.

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

[24] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[25] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of ISCA-30*, June 2003.

[26] M. Marty, J. Bingham, M. Hill, A. Hu, M. Martin, and D. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *Proceedings of HPCA-11*, February 2005.

[27] M. Marty and M. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *Proceedings of MICRO-39*, December 2006.

[28] A. Meixner and D. Sorin. Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures. In *Proceedings of HPCA-13*, February 2007.

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

[30] S. Mukherjee, J. Emer, and S. Reinhardt. The Soft Error Problem: An Architectural Perspective. In *Proceedings of HPCA-11 (Industrial Session)*, February 2005.

[31] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO-40)*, December 2007.

[32] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. Das. Exploring Fault-Tolerant Network-on-Chip Architectures. In *Proceedings of DSN*, 2006.

[33] M. Prvulovic, M. Garzaran, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proceedings of ISCA-28*, June 2001.

[34] R. Rajwar and J. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of ASPLOS-X*, October 2002.

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

[36] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Thread-Level Speculation on a CMP Can Be Energy Efficient. In *Proceedings of ICS*, June 2005.

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

[38] Semiconductor Industry Association. International Technology Roadmap for Semiconductors 2005. `http://www.itrs.net/Links/2005ITRS/Home2005.htm`.

[39] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinatorial Logic. In *Proceedings of DSN*, June 2002.

[40] D. Sorin, M. Hill, and D. Wood. Dynamic Verification of End-to-End Multiprocessor Invariants. In *Proceedings of DSN*, June 2003.

[41] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of ISCA-27*, June 2000.

[42] S. Vrudhula, D. Blaauw, and S. Sirichotiyakul. Estimation of the Likelihood of Capacitive Coupling Noise. In *Proceedings of DAC*, 2002.