A Scalable, Non-blocking Approach to Transactional Memory

Hassan Chafi Jared Casper Brian D. Carlstrom
Austen McDonald Chi Cao Minh Woongki Baek
Christos Kozyrakis Kunle Olukotun

Computer Systems Laboratory
Stanford University
{*hchafi, jaredc, bdc, austenmc, caominh, wbaek, kozyrakis, kunle*}*@stanford.edu*

Abstract

Transactional Memory (TM) provides mechanisms that promise to simplify parallel programming by eliminating the need for locks and their associated problems (deadlock, livelock, priority inversion, convoying). For TM to be adopted in the long term, not only does it need to deliver on these promises, but it needs to scale to a high number of processors. To date, proposals for scalable TM have relegated livelock issues to user-level contention managers. This paper presents the first scalable TM implementation for directory-based distributed shared memory systems that is livelock free without the need for user-level intervention. The design is a scalable implementation of optimistic concurrency control that supports parallel commits with a two-phase commit protocol, uses write-back caches, and filters coherence messages. The scalable design is based on Transactional Coherence and Consistency (TCC), which supports continuous transactions and fault isolation. A performance evaluation of the design using both scientific and enterprise benchmarks demonstrates that the directory-based TCC design scales efficiently for NUMA systems up to 64 processors.

1 Introduction

The advent of CMPs has moved parallel programming from the domain of high performance computing to the mainstream. To address the need for a simpler parallel programming model, Transactional Memory (TM) has been developed and promises good parallel performance with easy-to-write parallel code [17, 16, 14]. Unlike lock-based synchronization, transactional memory allows for non-blocking synchronization with coarse-grained code, deadlock and livelock freedom guarantees, and a higher degree of fault atomicity between concurrent threads. Several studies have already shown that TM works well for small-scale, bus-based systems with 8 to 16 processors [26, 32]. However, given the ever-increasing transistor densities, large-scale multiprocessors with more than 16 processors on a single board or even a single chip will soon be available. As more processing elements become available, programmers should be able to use the same programming model for configurations of varying scales. Hence, TM is of long-term interest only if it scales to large-scale multiprocessors.

This paper presents the first scalable, non-blocking implementation of TM that is tuned for continuous use of transactions within parallel programs. By adopting continuous transactions, we can implement a single coherence protocol and provide non-blocking synchronization, high fault isolation, and a simple to understand consistency model. The basis for this work is a directory-based implementation of the Transactional Coherence and Consistency (TCC) model that defines coherence and consistency in a shared memory system at transaction boundaries [15]. Unlike other scalable TM proposals, we detect conflicts only when a transaction is ready to commit in order to guarantee livelock-freedom without intervention from user-level contention managers. We are also unique in our use of lazy data versioning which allows transactional data into the system memory only when a transaction commits. This provides a higher degree of fault isolation between common case transactions. To make TCC scalable, we use directories to implement three techniques: a) parallel commit with a two-phase protocol for concurrent transactions that involve data from separate directories; b) write-back commit that communicates addresses, but not data, between nodes and directories; c) all address and data communication for commit and conflict detection only occurs between processors that may cache shared data. We demonstrate that these techniques allow for scalable performance.

Essentially, this work describes how to implement optimistic concurrency control [21] in scalable hardware using directories. Its major contributions are:

- We propose a scalable design for a TM system that is non-blocking, has improved fault-isolation, and is tuned for continuous transactional execution.

- We describe a directory implementation that reduces commit and conflict detection overheads using a two-phase commit scheme for parallel commit and write-back caches. The directory also acts as a conservative filter that reduces commit and conflict detection traffic across the system.
- We demonstrate that the proposed TM system scales efficiently to 64 processors in a distributed shared-memory (DSM) environment for both scientific and commercial workloads. Speedups with 32 processors range from 11 to 32 and for 64 processors, speedups range from 16 to 57. Commit overheads and interference between concurrent transactions are not significant bottlenecks (Less than 5% of execution time on 64 processors).

Overall, this paper shows how to scale TM to achieve the performance expected from large-scale parallel systems while still providing desired high-level features, such as livelock-freedom, that make it attractive to programmers.

The rest of the paper is organized as follows. Section 2 gives an overview of the scalable protocol, and a review of Optimistic Concurrency Control. Section 3 discusses the protocol's implementation. Section 4 presents our experimental methodology and evaluation results. Section 5 discusses related work and Section 6 concludes the paper.

2 Protocol Overview

2.1 Optimistic Concurrency Control

Lazy transactional memory systems achieve high performance through optimistic concurrency control (OCC), which was first proposed for database systems [21]. Using OCC, a transaction runs without acquiring locks, optimistically assuming that no other transaction operates concurrently on the same data. If that assumption is true by the end of its execution, the transaction commits its updates to shared memory. Dependency violations are checked lazily at commit time. If conflicts between transactions are detected, the non-committing transactions violate, their local updates are rolled back, and they are re-executed. OCC allows for non-blocking operation and performs very well in situations where there is ample concurrency and conflicts between transactions are rare, which is the common case transactional behavior of scalable multithreaded programs [8].

Execution with OCC consists of three phases:

- **Execution Phase**: The transaction is executed, but all of its speculative write-state is buffered locally. This write-state is not visible to the rest of the system.
- **Validation Phase**: The system ensures that the transaction executed correctly and is serially valid (consistent). If this phase does not complete successfully the transaction aborts and restarts. If this phase completes, the transaction cannot be violated by other transactions.
- **Commit Phase**: Once a transaction completes the validation phase, it makes its write-state visible to the rest of the system during the commit phase.

Kung et al.[21] outline three conditions under which these phases may overlap in time to maintain correct transactional execution. To validate a transaction, there must be a serial ordering for all transactions running in the system. Assume that each transaction is given a Transaction ID (TID) at any point before its validation phase. For each transaction with $TID = j$ (T_j) and for all T_i with $i < j$ one of the following three conditions must hold:

1. T_i completes its commit phase before T_j starts its execution phase.
2. T_j did not read anything T_i wants to validate, and T_i finished its commit phase before T_j starts its commit phase.
3. T_j did not read nor is it trying to validate anything T_i wants to commit, and T_i finished its execution phase before T_j completed its execution phase.

Under condition 1, there is no execution overlap: each transaction can start executing only after the transaction before it has finished committing, yielding no concurrency whatsoever. Under condition 2, execution can be overlapped, but only one transaction is allowed to commit at a time. The original, or "small-scale", TCC design [15], for example, operates under condition 2: each transaction arbitrates for a token and uses an ordered bus to ensure its commits finish before any other transaction starts committing. The sequential commits limit concurrency and become a serial bottleneck for the small-scale TCC system at high processor counts. Condition 3 allows the most concurrency: if there are no conflicts, transactions can completely overlap their execution and commit phases. Scalable TCC operates under condition 3 which allows parallel commits; however, this requires a more complex implementation. Furthermore, additional mechanisms are required to accommodate the distributed nature of a large scale parallel system, specifically its distributed memory and unordered interconnection network.

2.2 Protocol Operation Overview

The small-scale TCC model buffers speculative data while in its execution phase. During validation, the processor requests a commit token which can only be held by one processor at a time. Once the commit token is acquired, the processor proceeds to flush its commit data to a shared non-speculative cache via an ordered bus. The small-scale

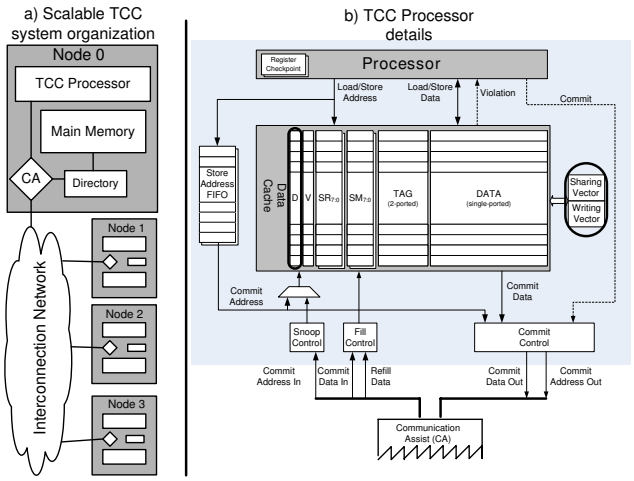

Figure 1. Scalable TCC Hardware: The circled parts of the TCC processor indicate additions to the original TCC for Scalable TCC.

TCC model works well within a chip-multiprocessor where commit bandwidth is plentiful and latencies are low [26]. However, in a large-scale system with tens of processors, it will perform poorly. Since commits are serialized, the sum of all commit times places a lower bound on execution time. Write-through commits with broadcast messages will also cause excessive network traffic that will likely exhaust the network bandwidth of a large-scale system.

Figure 1a shows the organization of the Scalable TCC hardware, which is similar to many Distributed Shared Memory (DSM) systems [23, 1, 22]. The Scalable TCC protocol leverages the directories in the DSM system to overcome the scaling limitations of the original TCC implementation while maintaining the same execution model from the programmer's point of view. The directories allow for several optimizations. First, even though each directory allows only a single transaction to commit at a time, multiple transactions can commit in parallel to different directories. Thus, increasing the number of a directories in the system provides a higher degree of concurrency. Parallel commit relies on locality of access within each transaction. Second, they allow for a write-back protocol that moves data between nodes only on true sharing or cache evictions. Finally, they allow filtering of commit traffic and eliminate the need to broadcast invalidation messages.

Scalable TCC processors have caches that are slightly modified from the caches in the original TCC proposal. Directories are used to track processors that may have speculatively read shared data. When a processor is in its validation phase, it acquires a TID and doesn't proceed to its commit phase until it is guaranteed that no other processor can violate it. It then sends its commit addresses only to directories responsible for data written by the transaction. The directories generate invalidation messages to processors that were marked as having read what is now stale data. Processors receiving invalidation messages then use their own tracking facilities to determine whether to violate or simply invalidate the line.

Message	Description
Load	Load a cache line
TID Request	Request a Transaction Identifier
Skip Message	Instructs a directory to skip a given TID
NSTID Probe	Probes for a Now Servicing TID
Mark	Marks a line intended to be committed
Commit	Instructs a directory to commit marked lines
Abort	Instructs a directory to abort a given TID
Write Back	Write back a committed cache line, removing it from cache
Flush	Write back a committed cache line, leaving it in cache
Data Request	Instructs a processor to flush a given cache line to memory

Table 1. The coherence messages used in the Scalable TCC protocol.

For our protocol to be correct under condition 3 of OCC, two rules must be enforced. First, conflicting writes to the same address by different transactions are serialized. Second, a transaction with TID i (T_i) cannot commit until all transactions with lower TIDs that could violate it have already committed. In other words, if T_i has read a word that a transaction with a lower TID may have written, T_i must first wait for those other transaction to commit. Each directory requires transactions to send their commit-address messages in the order of the transactions' TIDs. This means that if a directory allows T_5 to send commit-address messages, then T_5 can infer that transactions T_0 to T_4 have already communicated any commit information to this particular directory. Each directory tracks the TID currently allowed to commit in the *Now Serving TID (NSTID)* register. When a transaction has nothing to send to a particular directory by the time it is ready to commit, it informs the directory by sending it a *Skip* message that includes its TID so that the directory knows not to wait for that particular transaction.

Simple Commit Example Figure 2 presents a simple example of the protocol where a transaction in processor P1 successfully commits with one directory while a second transaction in processor P2 violates and restarts. The figure is divided into six parts. Changes in state are circled and events are numbered to show ordering, meaning all events numbered ❶ can occur at the same time and an event labeled ❷ can only occur after all events labeled ❶. The description makes use of Table 1 which lists the coherence messages used to implement the protocol.

In part a, processors P1 and P2 each load a cache line ❶ and are subsequently marked as sharers by Directory 0

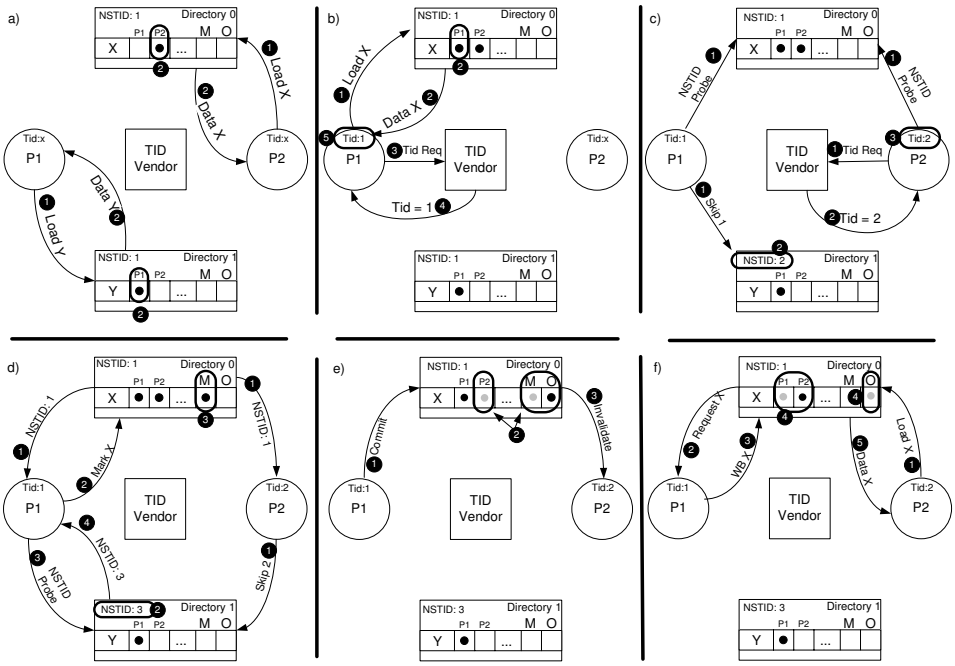

Figure 2. Execution with the Scalable TM protocol

and Directory 1 respectively. Both processors write to data tracked by Directory 0, but this information is not communicated to the directory until they commit.

In part b, processor P1 loads another cache line from Directory 0, and then starts the commit process. It firsts sends a *TID Request* message to the TID Vendor ❸, which responds with TID 1 ❹ and processor P1 records it ❺.

In part c, P1 communicates with Directory 0, the only directory it wants to write to, in order to commit its write. First, it probes this directory for its NSTID using an *NSTID Probe* message ❶. In parallel, P1 sends Directory 1 a Skip message since Directory 1 is not in its write-set, causing Directory 1 to increase its NSTID to 2 ❷. Meanwhile, P2 has also started the commit process. It requests a TID, but can also start probing for a NSTID ❶ from Directory 0—probing does not require the processor to have acquired a TID. P2 receives TID 2 ❷ and records it internally ❸.

In part d, both P1 and P2 receive answers to their NSTID probe. P2 also sends a Skip message to Directory 1 ❶ causing the directory's NSTID to change to 3. P2 cannot send any commit-address messages to Directory 0 because the NSTID answer it received is lower than its own TID. P1's TID, on the other hand, matches Directory 0's NSTID, thus it can send commit-address messages to that directory. Note that we are able to serialize the potentially conflicting writes from P1 and P2 to data from Directory 0. P1 sends a *Mark* message ❷, which causes line X to be marked as part of the committing transaction's write-set ❸. Without using Mark messages, each transaction would have to complete its validation phase before sending the addresses it wants to commit. Mark messages allows transactions to pre-commit addresses to the subset of directories that are ready to service the transaction. Before P1 can complete its commit, it needs to make sure no other transactions with a lower TID can violate it. For that, it must make sure that every directory in its read-set (0 and 1) is done with younger transactions. Since it is currently serviced by Directory 0, it can be certain that all transactions with lower TIDs have already been serviced by this directory. However, P1 needs to also probe Directory 1 ❸. P1 receives NSTID 3 as the answer ❹ hence it can be certain all transactions younger than TID 3 have been already serviced by Directory 1. Thus, P1 cannot be violated by commits to any directory.

In part e, P1 sends a *Commit* message ❶, which causes all marked (M) lines to become owned (O) ❷. Each marked line that transitions to owned generates invalidations that are sent to all sharers of that line except the committing processor which becomes the new owner ❸. P2 receives the invalidation, discards the line, and violates because its current transaction had read it.

In part f, P2 attempts to load an owned line ❶; this causes a data request to be sent to the owner ❷; the owner then writes back the cache line and invalidates the line in its cache ❸. Finally, the directory forwards the data to the requesting processor ❺ after marking it in the sharers list for this line ❹.

Each commit requires the transaction to send a single multicast skip message to the set of directories not present in either its write- or read-sets. The transaction also communicates with directories in its write-set, and probes di-

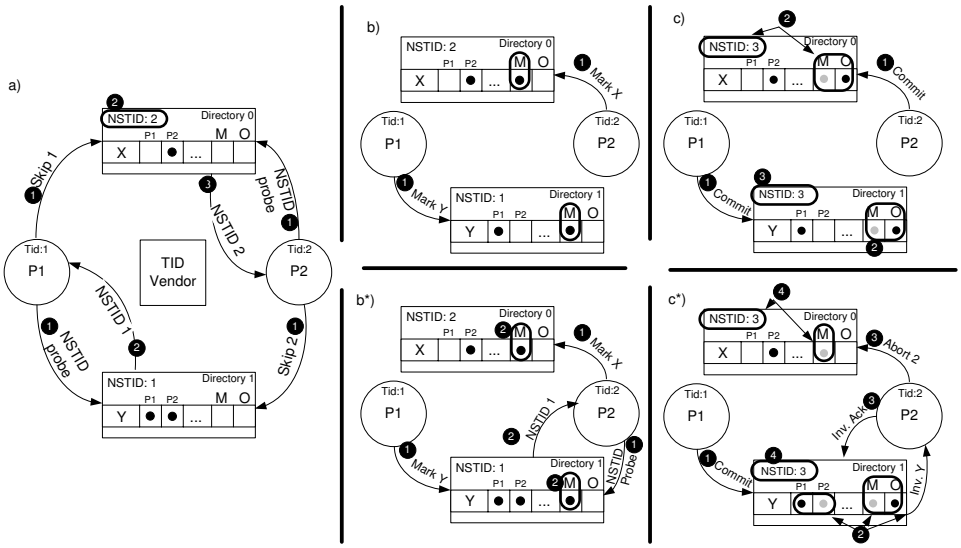

Figure 3. Two scenarios involving a pair of transactions attempting to commit in parallel. In the top scenario they are successful; in the bottom, they fail. Note that while the two scenarios start by generating the same messages in part a), in the first scenario P2 is not marked as a sharer of line Y of Directory 1.

rectories in its read-set. This communication does not limit performance scalability because limited multicast messages are cheap in a high bandwidth interconnect and the multicast is amortized over the whole transaction. Furthermore, as we will show in Section 4, the number of directories touched per commit is small in the common case. Even when this is not the case, the use of larger transactions still results in an efficient system. These observations are borne out in the performance evaluation, which shows that this protocol scales very well in practice.

Parallel Commit Example At any point in time, each directory can have a single transaction committing. But, since there are many directories in the system, and not all transactions write to all directories, there can be overlap. Figure 3 illustrates two possible scenarios for a pair of transactions attempting to commit in parallel. Assume the two processors have already asked for and obtained TIDs 1 and 2 respectively. In both scenarios, P1 has written data from Directory 1, while P2 has written data from Directory 0.

In part a, each transaction sends a NSTID Probe message to the corresponding directory and receives a satisfactory NSTID. The necessary Skip messages have already been sent by each processor.

Parts b and c present a successful parallel commit. P1 has accessed data in only Directory 1 and P2 has accessed data only in Directory 0. In part b, both processors send a Mark message to the appropriate directory ❶. There are no additional probing messages as each processor's read-set includes a single directory. In part c, both processors send their Commit messages ❶, and the two directories process the commits concurrently by updating their NSTID and the sharers and owner for any affected cache lines ❷.

Parts b* and c* present a scenario where parallel commit fails and P2 has to abort its transaction. P2 has read a word from Directory 1 that P1 will attempt to commit. In part b*, P2 has to probe Directory 1 as well because it is included in its read-set ❶. P2 receives an answer back with NSTID 1, which is smaller than its own TID of 2, so it cannot proceed. It will have to re-probe until the NSTID it receives is higher or equal to its own. In other words, the two commits have been serialized as they both involve Directory 0. However, the P2 commit will never occur as P1 commits first in part c* ❶. Since P1 commits a new value for a line speculatively read by P2, Directory 1 will send P2 an invalidation, which will cause it to violate ❷. Since P2 had already sent a Mark message to Directory 0, it needs to send an *Abort* message which causes the directory to clear all the Marked bits. Note that if P2 had a lower TID than P1, the two commits would still be serialized, but there would be no violation as P2 would commit first. When P1 commits afterward, P2 would receive the invalidation but it would detect that the invalidation was sent from a transaction that occurred logically after it; P2 will, nonetheless, invalidate the target cache line.

3 Implementation Details

3.1 Processor State

Figure 1b shows the details of a TCC processor. Speculative state is stored in multiple levels of data caches. We use on chip caches because they provide high capacity with support for fast associative searches [12, 20, 37]. It also allows speculative and non-speculative data to dynamically share the storage capacity available in each processor in a

flexible way. Figure 1b presents the data cache organization for the case with word-level speculative state tracking. Tag bits include valid, speculatively-modified (SM), and speculatively-read (SR) bits for each word. For a 32-bit processor with 32-byte cache lines, 8 bits of each type are needed per line. The SM bit indicates that the corresponding word has been modified during the currently executing transaction. Similarly, the SR bit indicates that its word has been read by the current transaction. Line-level speculative state tracking works in a similar manner but requires only a single valid, SR, and SM bit per line. This approach easily expands to multiple levels of private caches, with all levels tracking this information. Long-running transactions may overflow the limited speculative buffering hardware. This situation can be handled using mechanisms like VTM or XTM [31, 9]. However, recent studies have shown that, with large private L2 caches tracking transactional state, it is unlikely that these overflows will occur in the common case [8]. TCC is a simpler protocol; it has fewer states (regular and transient) than MESI or other TM proposals that extend conventional coherence protocols such as LogTM. The use of directories is orthogonal, as it is due to the NUMA nature of a large scale machine.

Figure 1b also uses circles to highlight the additional modifications necessary in each processor cache for the Scalable TCC protocol. We add a dirty bit (*D*) per cache line to support the write-back protocol. We check the dirty bit on the first speculative write in each transaction. If it is already set, we first write back that line to a non-speculative level of the cache hierarchy. The former non-speculative cache is required for writeback behavior. Without it, the speculative cache would only be delaying the writethrough till the next speculative access to a dirty line. The only other additions to the cache structure, compared to the small-scale TCC protocol, is the inclusion of the *Sharing Vector* and the *Writing Vector*, which include a bit per directory in the system and indicated which will be involed when the transaction commits. The Sharing Vector tracks the home directories for speculative data read by the current transaction. On each load, we infer the directory number from the address and set the corresponding bit in the vector. Similarly, the Writing Vector tracks the home directories for speculative data written by the current transaction.

3.2 Directory State

Figure 4 shows the directory organization for the scalable TCC protocol. The directory tracks information for each cache line in the physical memory housed in the local node. First, it tracks the nodes that have speculatively read the cache line (sharers list). This is the set of nodes that will be sent invalidates whenever these lines get committed. Second, the directory tracks the owner for each cache line, which is the last node to commit updates to the line until it writes it back to physical memory (eviction). The owner is indicated by setting a single bit in the sharers list and the *Owned* bit. The *Marked* bit is set for cache lines involved in an ongoing commit to this directory. Finally, we include a *TID* field which allows us to eliminate rare race conditions when an unordered interconnect is used, as discussed in the Race Elimination part of Section 3.3. Each directory also has a NSTID register and a *Skip Vector*, described below.

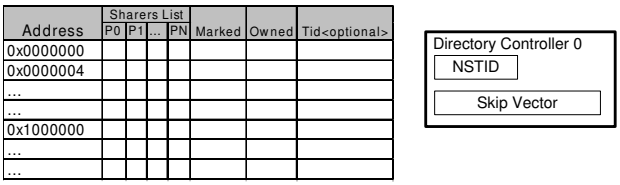

Figure 4. The directory structure for Scalable TCC.

3.3 Protocol Operation

NSTID and Skip Vector Directories control access to a contiguous region of physical memory. At any time, a single directory will service one transaction whose TID is stored in the directory's NSTID register. For example, a directory might be serving T_9; this means that only T_9 can send state-altering messages to the memory region controlled by that directory. If T_9 has nothing to commit to that directory, it will send a skip message with its TID attached. This will cause the directory to mark the TID as completed. The key point is that each directory will either service or skip every transaction in the system. If two transactions had an overlapping write-set, then the concerned directory will serialize their commits.

Each directory tracks skip messages using a *Skip Vector*. The Skip Vector allows the directory to buffer skip messages sent early by transactions with TIDs higher than the one in the NSTID. Figure 5 shows how the Skip Vector is maintained. While a directory with NSTID 0 is busy processing commit-related messages from T_0, it can receive and process skip messages from transactions T_1, T_2, and T_4. Each time a skip message with TID x is received, the directory sets the bit in location $(x - NSTID)$ of the Skip Vector. When the directory is finished serving T_0, it marks the first bit of the Skip Vector, and then proceeds to shift the Skip Vector left till the first bit is no longer set. The NSTID is increased by the number of bits shifted.

Commit Processing When a transaction is attempting to commit, it probes all directories in its Writing and Sharing Vector until it receives an NSTID equal or higher than its TID. Repeated NSTID probing can be avoided by having the directory not respond until the required TID is being serviced. For each directory that has a satisfactory NSTID, the transaction sends Mark messages for the corresponding address in its write-set. We assume write-allocate caches,

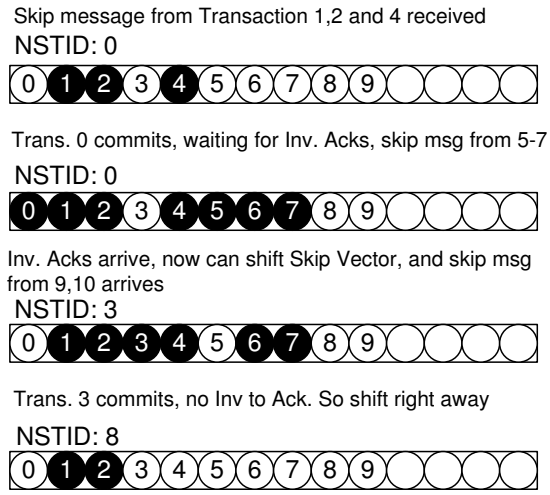

Figure 5. Skip Vector Operation. Note that bit 0 corresponds to the NSTID and black means the bit is set.

so the processor that marks a line is a sharer or the owner of the line. Note that while a transaction is sending Mark messages to one directory, it may still probe other directories. This is essential to the scalability of the design, as transactions only have to wait for directories that may be serving transactions with a lower TID. Once marking is complete for all directories in the Writing Vector and the transaction has received a NSTID higher than its TID for each directory in its Sharing Vector, the transaction can commit by sending a multicast *Commit* message to all these directories. Each directory gang-upgrades Marked lines to Owned and potentially generates invalidations if there are sharers other than the committing processor for these lines. If the transaction is violated after it has sent Mark messages to a few directories, it needs to send abort messages to all directories in the Writing Vectors to allow gang-clearing of Mark bits.

Any processor that attempts to load a marked line will be stalled by the corresponding directory. We could have allowed the processor to load the value in memory at the same time, and mark it as a sharer. We optimize for the common case and assume commit attempts will succeed. Thus it is best to stall the loading processor to avoid having to subsequently invalidate it and cause a violation.

If the processor caches track speculatively read or written data at the word level, the directories can be made to accommodate this fine granularity conflict detection. When transactions mark lines, word flags can be sent alongside the Mark messages, these flags are buffered at the directory. When invalidations are sent, these word flags are included in the invalidating messages. A processor is cleared from the sharers list of a particular line only when a commit to that line causes an invalidation to be sent. In other words, a processor does not notify the directory on evictions of non-dirty lines (replacement hints). This may generate extra invalidation messages, but will not cause any false violations as each processor uses the information tracked by its cache to determine whether to cause a violation upon receiving an invalidation (only if that word has been speculatively read by its current transaction).

Livelock-Free Operation and Forward-Progress guarantees If two transactions in the process of committing either have a write conflict or true data dependencies, the transaction with the lower TID always succeeds in committing. The design of the directory guarantees this behavior. A transaction with a higher TID will not be able to write to a directory until all transactions with lower TID have either skipped that directory or committed. Furthermore, the transaction cannot commit until it is sure that all directories it has speculatively loaded from have serviced all lower-numbered transactions that can potentially send an invalidation to it. This yields a livelock-free protocol that guarantees forward progress. Limited starvation is possible: a starved transaction keeps its TID at violation time, thus over time it will become the lowest TID in the system. While long transactions that retain their TID after aging may decrease system performance, the programmer is still guaranteed correct execution. Moreover, TCC provides a profiling environment, TAPE [7], which allows programmers to quickly detect the occurrence of this rare event.

Transactions are assigned TIDs at the end of the execution phase to maximize system throughput. This may increase the probability of starving long-running transactions, but this is mitigated by allowing those transactions to request and retain a TIDs after they violate, thus insuring their forward-progress. TIDs are assigned by a global TID vendor. Distributed time stamps such as in TLR [30] will not work for our implementation since these mechanisms do not produce a gap-free sequence of TIDs, rather only an ordered set of globally unique timestamps.

Race Elimination Certain protocol race conditions may occur and require attention. For example, whenever a transaction with a given TID commits, directories involved in the commit have to wait for all resulting invalidations to be acknowledged before incrementing their NSTID. This resolves a situation in which a transaction with TID Y is allowed to commit since it received NSTID Y as an answer to its probe before receiving an invalidation from transaction X with $X < Y$.

Systems with unordered networks may introduce additional races. For example, assume Transaction 0 commits line X and then writes back line X. Transaction 1 running on the same processor commits a new value to the same line X' and writes back line X'. X and X' have the same address but different data. It is possible in a highly congested network that the write-back X' arrives before the write-back of X. Both these flushes will have the same owner and may cause memory inconsistencies. This data race is resolved by

Feature	Description
CPU	1–64 single-issue PowerPC cores (32)
L1	32-KB, 32-byte cache line 4-way associative, 1 cycle latency
L2	512-KB, 32-byte cache line 8-way associative, 16 cycle latency
ICN	2D grid topology, 7-28 cycles link latency (14)
Main Memory	100 cycles latency
Directory	Full-bit vector sharer list; first touch allocate; Directory cache 10 cycle latency

Table 2. Parameters for the simulated architecture. Unless indicated otherwise, results assume the default values in parentheses.

attaching a TID to each owned directory entry at the time of commit. Write-backs get tagged at the processor with the most recent TID that processor acquired. In our previous example, when commit X' is processed, the directory entry will be tagged with TID 1. Flush X will have a TID 0 attached to it; thus, it will be dropped if it arrives out of order.

One final example of a data race occurs as follows. A processor sends a load request to a given directory. The directory satisfies the load, a separate transaction's commit generates an invalidate, and the invalidate arrives at the requesting processor before the load. To resolve this, processors could just drop that load when it arrives. This race condition is present in all invalidation-based protocols that use an unordered interconnect.

4 Methodology and Evaluation

In this section we evaluate Scalable TCC. We first discuss our experimental methodology. We then introduce the applications we use to evaluate our protocol, and finally we present and discuss our results.

4.1 Methodology

We evaluate Scalable TCC using an execution-driven simulator that models the PowerPC ISA. All instructions, except loads and stores, have a CPI of 1.0. Table 2 presents the main parameters of the simulated architecture. The memory system models the timing of the on-chip caches, and the interface to the communication assist and directory cache. All contention and queuing for accesses to caches and interconnects is accurately modeled. A simple first-touch policy is used to map virtual pages to physical memory on the various nodes.

Applications To evaluate Scalable TCC we use a suite of parallel applications: equake, swim, and tomcatv from the SPEC CPU2000 FP suite[34]; barnes, radix, volrend, water-spatial and water-nsquared from the SPLASH-2 parallel benchmark suites[39]; and SPECjbb2000[35]. SPECjbb executes using the Jikes RVM [2] Java Virtual Machine. We also include two applications from the CEARCH suite [5]. CEARCH applications are a set of new cognitive algorithms based on probabilistic and knowledge-based reasoning and learning. For the SPEC CPU2000, CEARCH, and SPLASH applications, we converted the code between barrier calls into TCC transactions, discarding any lock or unlock statements. For SPECjbb2000, we converted the 5 application-level transactions into unordered transactions.

Table 3 shows the key characteristics of our applications executing under TCC. Transaction sizes range from two-hundred to forty-five thousand instructions. This wide range of transaction size provides a good evaluation of the scalable TCC system. The 90%-ile read-set size for all transactions is less than 52 KB, while the 90%-ile write-set never exceeds 18 KB. Table 3 also presents the ratio of operations per word in the write-set. A high ratio implies that a transaction does a large amount of computation per address in its write-set. Hence, it may be able to better amortize commit latency. The ratio ranges from 2 to 180, depending on the transaction size and the store locality exhibited in each application. In addition to these non-architectural characteristics, Table 3 also gives key characteristics that can be used to understand the behavior of the directory-based cache coherence system, all numbers are for the 32 processor case. We show the number of directories touched per commit, the directory cache working set defined by the number of entries which have remote sharers, and finally, the directory occupancy defined by the number of cycles a directory is busy servicing a commit. These results indicate good directory performance. The working set fits comfortably in a 1MB directory cache. The directory occupancy is typically a fraction of the transaction execution time, and most applications touch only a couple of directories per commit.

4.2 Results and Discussion

In this section, we analyze the performance of each of the applications to quantify the effectiveness of the scalable TCC architecture; we also explore the impact of communication latency on the performance of the applications.

Figure 6 shows the execution time breakdown of all the applications running on a single processor. The execution time is broken down into five components that indicate how the processors spend their time. The first component is *Useful* cycles, which represents the cycles spent executing instructions that are committed. The second component, *Cache Miss*, is the time spent stalling for cache misses. The third component, *Idle*, is time spent waiting at barriers and synchronization points. The fourth component, *Commit*, is time spent waiting for a transaction to commit. Finally, the fifth component, *Violations*, refers to time wasted due to conflicts (not present in uniprocessor case, Figure 6). Commit time, the only additional overhead of a TCC processor is insignificant at around 1 percent on average, thus a TCC system with one processor is equivalent to a conventional

Application	Input	Trans. Size 90th % (Inst)	Trans. Wr. Set 90th % (KB)	Trans. Rd. Set 90th % (KB)	Ops. per Word Written 90th %	Directories per commit 90th %	Working set (Dir.) 90th % (Entries)	Directory Occupancy 90th % (Cycles)
barnes [39]	16,384 mol.	7,462	0.66	1.69	11.04	1	384	813
Cluster GA [5]	ref	238	0.01	0.14	7.25	1	266	354
equake [34]	ref	866	0.35	1.73	11.00	3	8926	485
radix [39]	1M keys	32,681	7.41	8.16	17.20	32	13725	643
SPECjbb2000 [35]	1,472 trans.	5,556	0.12	0.16	180.60	2	14422	229
SVM Classify [5]	ref	13,054	0.62	9.72	84.27	2	61	248
swim [34]	ref	45,876	18.00	52.00	9.60	1	941	765
tomcatv [34]	ref	21,060	10.50	15.90	7.83	2	1572	426
volrend [34]	ref	1,098	0.31	0.39	2.09	1	977	560
water-nsquared [39]	512 mol.	948	0.45	0.45	8.12	1	139	323
water-spatial [39]	512 mol.	7,466	1.26	1.27	23.14	2	752	312

Table 3. Applications and their scalable TM characteristic for performance. The 90th percentile transaction size in instructions, transaction write- and read-set sizes in KBytes, and operations per word written. We also show the number of directories touched per commit and the 90th percentile of both the working set cached at the directory in number of entries and the directory's occupancy in cycles per commit

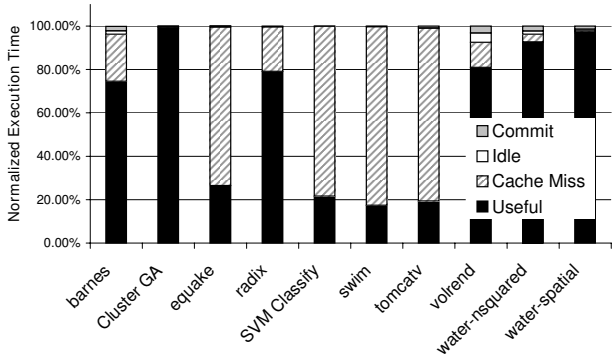

Figure 6. The normalized execution time of applications running on one processor.

uni-processor. Figure 7 shows the execution time of the applications normalized to the single processor case as we scale the number of processors from 8 to 64.

We see from Figure 7 that the overall performance of the Scalable TCC architecture is good. Speedups with 32 processors range from 11 to 32 and for 64 processors, speedups range from 16 to 57. For most applications in our suite, commit time is a small fraction of the overall execution time. Looking at the applications in more detail gives us more insight into the behavior of Scalable TCC.

barnes performs well on our system. This performance is due to the fact that all components of the execution time scale down with increasing processor counts. In particular, even at high processor counts, commit time remains a small fraction of overall execution time, indicating that the Scalable TCC commit protocol behaves well.

Cluster GA is a genetics algorithm. At low processor counts, it suffers from violations which are not evenly distributed across the processors. This leads to additional load imbalances. At high processor counts, the roughly fixed number of cycles wasted due to violations are more evenly distributed across the processors.

equake is a SPEC application with limited parallelism and lots of communication. This results in transactions that are small to avoid violations inherent to transactions that communicate excessively. Even though small transactions may reduce violation time, they lead to increased commit time at high processor counts.

radix has large transactions and the operations per word written ratio is one of the highest of the applications in our suite, which would indicate that it should perform well in our system. Even though radix has an extremely high number of directories touched per commit (all directories are touched) it still performs well on our system because its transactions are large enough to hide the extra commit time.

SPECjbb2000 scales linearly with the number of processors due to its very limited inter-warehouse communication. This benchmark has the highest operations per word written ratio of all studied applications; thus, it is ideal for Scalable TCC.

SVM Classify is the best performing application. This is due to large transactions and a large operations per word written ratio. In a similar fashion to barnes, the Scalable TCC commit protocol behaves very well. In particular, commit time is non-existent, even at high processor counts,

swim and **tomcatv** are examples of applications with very little communication. They exhibit large transactions that generate large write-sets. However, these write-sets do not require remote communication.

volrend is affected by an excessive number of commits required to communicate flag variables to other processors. This behavior yields a low operations per word written ratio that limits the scalability of this application, and increases commit time. A breakdown of this commit time (not shown) indicates that the majority of the time is spent probing di-

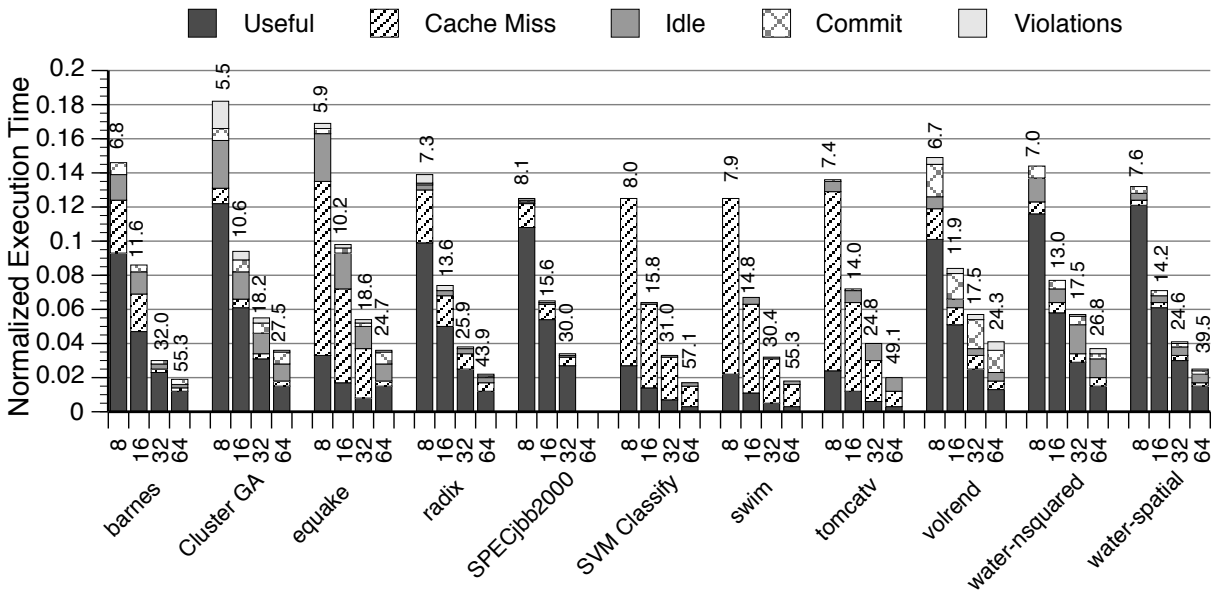

Figure 7. The performance of Scalable TCC as the processor count varies from 8 to 64 CPUs. The Execution time is normalized to a single CPU. The numbers on the top of each bar represent the speedup achieved over a single processor.

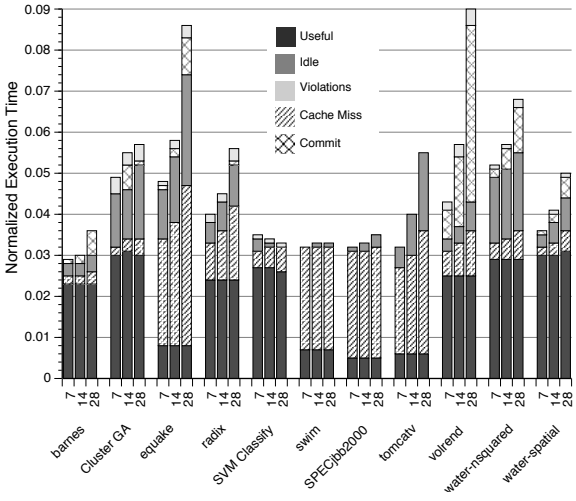

Figure 8. The impact of communication latency for 32 processors normalized to a single processor. The X-axis is cycles-per-hop.

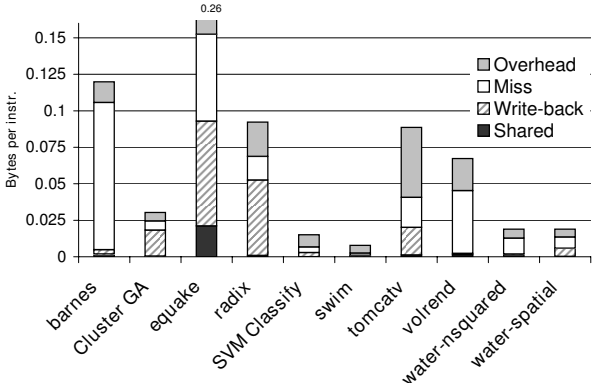

Figure 9. Remote traffic bandwidth for 32 processors.

rectories that are in a processor's Sharing Vector.

Lastly, comparing **water-nsquared** and **water-spatial** is instructive; water-spatial has larger transactions and a larger number of operations per word written. The water-spatial algorithm has inherently less communication and synchronization. Thus, water-spatial scales better: it has less commit time, less violation time, and less synchronization time. For these applications, a breakdown of the commit time shows similar behavior as volrend.

Figure 8 shows the impact of varying the communication latency for 32 processors. The degree to which an application's performance is affected is determined by the number of remote loads and commits. For example, equake has a significant amount of remote load misses, and so increasing the latency to 28 cycles per link increases the execution time by 50%; likewise, volrend, which has a significant amount of commit time, sees the same level of degradation. In contrast to these applications, SPECjbb2000 and swim, which do not have significant remote load misses or commit time, suffer almost no performance degradation with increasing communication latency.

Finally, Figure 9 shows the traffic produced and consumed on average at each directory in the system at 32 processors. We report this traffic in terms of bytes per instruction. The total traffic, including data, address, and control, in bytes per instructions ranges between 0.01 to 0.26 bytes per instruction. With processors executing at 2 GHz, the to-

tal bandwidth ranges from 2.5 MBps for swim to 160 MBps for barnes, which is within the bandwidth range of commodity cluster interconnects such as Infiniband [18]. Additionally, large transactions and a high ratio of operations per word written yields low overhead. Comparing the traffic in Figure 9 for the SPLASH-2 benchmarks (barnes, radix, and water) to the published traffic numbers suggests that our numbers are within the published range (sometimes better, sometimes worse). Minor differences can be accounted for by the differences in architectural parameters such as cache sizes, block sizes, and the use of TCC versus conventional DSM cache coherence protocols [39].

5 Related Work

There have been a number of proposals for transactional memory (TM) that expand on the early work by Knight [19], Herlihy [17] and Stone [38]. Researchers have shown that transactional execution provides good performance with simple parallel code [30, 15, 3, 31, 27]. Some early transactional memory schemes hid transactions from programmers by performing lock-elision on conventional code [29, 30, 25], while more recent schemes propose transactions as the programming model [15, 3, 27].

In general, TM proposals can be classified by the policies they use for data version management and conflict detection [27]. The data version management policy specifies how to handle both new data, which becomes visible if the transaction commits, and old data, which should be retained if the transaction aborts. TM systems with eager version management store the new value in place (and must restore the old value if the transaction aborts) while systems with lazy version management buffer the new value until the transaction commits. Conflict detection can also be handled eagerly by preemptively detecting potential overlaps between the reads and writes of different transactions as they occur or lazily by detecting these conflicts only when a transaction is ready to commit.

Transactional memory proposals that use eager version management, such as LogTM, write to memory directly. This improves the performance of commits, which are more frequent than aborts; however, it may also incur additional violations not incurred by lazy versioning [6], and provides lower fault isolation [3, 27]. In the event of a fault, this policy would leave the memory in an inconsistent state. Moreover, conflicts are detected eagerly as transactions execute loads and stores [3, 27], which could lead to livelock. The solution proposed to the livelock problem is to employ a user-level contention manager to force the application to deal with livelock issues.

In TCC, transactions run continuously so all executed code is part of some transaction. Continuous transactions provide a uniform consistency model that is easy for programmers to reason about. In contrast to other TM proposals, TCC uses lazy conflict detection and version management which guarantees forward progress without application intervention. TCC keeps speculative updates stored in a write-buffer until commit to guarantee isolation of the transactions even in the face of hardware faults (this is similar to some other proposals [30, 3]). On abort, all that is required is to invalidate this buffer, which results in very low abort overhead. TCC postpones conflict detection until a transaction commits, which guarantees livelock-free operation without application-level intervention [15]. Even though commits are more costly with lazy version management and conflict detection, we have shown that it is still possible to achieve excellent parallel performance with these policies.

TCC has been heavily influenced by work on Thread-Level Speculation (TLS) [33, 13, 37, 20, 28, 11]. The basic difference is that TLS attempts optimistic concurrency within the semantics of a sequential program and communicates speculative state, while TCC provides optimistic concurrency with parallel algorithms and only communicates committed state. Nevertheless, similar hardware mechanisms can support both models [14]. Both TLS and transactional memory proposals have explored scalable implementations by modifying existing directory-based coherence [1, 22, 4]. The TLS proposals provide scalability but are limited to sequential program semantics [10, 36]. The scalable transactional memory system in [27] provides users with no guarantee for livelock-free operation.

In a similar manner to this proposal, Token Coherence [24] makes use of limited broadcast with small impact on the overall bandwidth requirements of the system. The excellent performance of both Scalable TCC and Token Coherence demonstrates that limited use of broadcast is not incompatible with scalability.

6 Conclusions

This paper presents a scalable TM architecture for directory-based, distributed shared memory (DSM) systems based on TCC. This is the first scalable implementation of a livelock-free hardware transactional memory system for continuous transactional execution. The system does not require user-level contention managers to provide its livelock-free guarantees. The architecture is based on a directory design that provides support for parallel commit, write-back caches, and coherence traffic filtering. Through execution-driven simulation, we demonstrated that the proposed design scales efficiently through 64 processors for both scientific and enterprise applications. Speedups with 32 processors range from 11 to 32 and for 64 processors, speedups range from 16 to 57. The ability to commit multiple transactions in parallel allows the design to avoid performance losses due to commit serialization.

Overall, our performance evaluation of the Scalable TCC architecture shows that it is possible to retain the parallel

programming benefits of TCC and still provide scalable performance for a wide range of applications. We find that the commit behavior of most applications works quite well with the Scalable TCC commit protocol.

7 Acknowledgments

Effort sponsored by National Science Foundation Grant CCF-0444470 and the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant number NBCH104009. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] A. Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proc. of the 22nd Annual Intl. Symp. on Computer Architecture (ISCA'95)*, pages 2–13, 1995.

[2] B. Alpern and other. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] C. S. Ananian et al. Unbounded Transactional Memory. In *Proc. of the 11th Intl. Symp. on High-Performance Computer Architecture (HPCA'05)*, San Franscisco, California, February 2005.

[4] L. A. Barroso et al. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proc. of the 27th Annual Intl. Symp. on Computer Architecture*, Vancouver, Canada, June 2000.

[5] CEARCH Kernels. `http://cearch.east.isi.edu/`.

[6] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proc. of the 33rd Intl. Symp. on Computer Architecture*, pages 227–238, 2006.

[7] H. Chafi et al. TAPE: A Transactional Application Profiling Environment. In *ICS '05: Proc. of the 19th Annual Intl. Conf. on Supercomputing*, pages 199–208. June 2005.

[8] J. Chung et al. The Common Case Transactional Behavior of Multithreaded Programs. In *Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, February 2006.

[9] J. Chung et al. Tradeoffs in transactional memory virtualization. In *ASPLOS-XII: Proc. of the 12th Intl. Conf. on Architectural support for programming languages and operating systems*. Oct 2006.

[10] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-memory Multiprocessors. In *Proc. of the 27th Intl. Symp. on Comp. Arch.*, June 2000.

[11] M. J. Garzarán et al. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *HPCA '03: Proc. of the 9th Intl. Symp. on High-Performance Computer Architecture*, page 191, February 2003.

[12] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proc. of the Fourth Intl. Symp. on High-Performance Computer Architecture*, February 1998.

[13] L. Hammond et al. Data speculation support for a chip multiprocessor. In *Proc. of the 8th Intl. Conf. on Architecture Support for Programming Languages and Operating Systems*, October 1998.

[14] L. Hammond et al. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI: Proc. of the 11th Intl. Conf. on Architectural support for programming languages and operating systems*, pages 1–13, October 2004.

[15] L. Hammond et al. Transactional memory coherence and consistency. In *Proc. of the 31st Intl. Symp. on Computer Architecture*, pages 102–113, June 2004.

[16] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. of the 18th annual ACM SIGPLAN Conf. on Object-oriented programing, systems, languages, and applications*, pages 388–402, 2003.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, pages 289–300, 1993.

[18] Infiniband Trade Association, *InfiniBand*. `http://www.infinibandta.org/`.

[19] T. Knight. An architecture for mostly functional languages. In *LFP '86: Proc. of the 1986 ACM Conf. on LISP and functional programming*, pages 105–112, August 1986.

[20] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

[21] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. on Database Systems*, 6(2), June 1981.

[22] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *ISCA '97: Proc. of the 24th annual Intl. Symp. on Computer architecture*, pages 241–251, 1997.

[23] D. Lenoski et al. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[24] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proc. of the 30th Intl. Symp. on Computer Architecture*, pages 182–193, June 2003.

[25] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X: Proc. of the 10th Intl. Conf. on Architectural support for programming languages and operating systems*, October 2002.

[26] A. McDonald et al. Characterization of TCC on Chip-Multiprocessors. In *PACT '05: Proc. of the 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, September 2005.

[27] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *12th Intl. Conf. on High-Performance Computer Architecture*, February 2006.

[28] M. Prvulovic et al. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proc. of the 28th Intl. Symp. on Computer architecture*, 2001.

[29] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO 34: Proc. of the 34th annual ACM/IEEE Intl. Symp. on Microarchitecture*, 2001.

[30] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X: Proc. of the 10th Intl. Conf. on Architectural support for programming languages and operating systems*, pages 5–17, October 2002.

[31] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA '05: Proc. of the 32nd Annual Intl. Symp. on Computer Architecture*, pages 494–505, June 2005.

[32] B. Saha et al. A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proc. of the eleventh ACM SIGPLAN Symp. on Principles and practice of parallel programming*, March 2006.

[33] G. S. Sohi, S. E. Breach, and T. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Annual Intl. Symp. on Comp. Arch.*, June 1995.

[34] Standard Performance Evaluation Corporation, *SPEC CPU Benchmarks*. `http://www.specbench.org/`, 1995–2000.

[35] Standard Performance Evaluation Corporation, *SPECjbb2000 Benchmark*. `http://www.spec.org/jbb2000/`, 2000.

[36] J. G. Steffan et al. A Scalable Approach to Thread-level Speculation. In *Proc. of the 27th Intl. Symp. on Computer Architecture*, June 2000.

[37] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proc. of the 4th Intl. Symp. on High-Performance Comp. Arch.*, 1998.

[38] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology*, 01(4):58–71, November 1993.

[39] S. C. Woo et al. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, pages 24–36, June 1995.