Characterization of TCC on Chip-Multiprocessors

Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom
Lance Hammond, Christos Kozyrakis, Kunle Olukotun

Computer Systems Laboratory
Stanford University
{*austenmc, jwchung, hchafi, caominh, bdc, lance, kozyraki, kunle*}*@stanford.edu*

Abstract

Transactional Coherence and Consistency (TCC) is a novel coherence scheme for shared memory multiprocessors that uses programmer-defined transactions as the fundamental unit of parallel work, synchronization, coherence, and consistency. TCC has the potential to simplify parallel program development and optimization by providing a smooth transition from sequential to parallel programs.

In this paper, we study the implementation of TCC on chip-multiprocessors (CMPs). We explore design alternatives such as the granularity of state tracking, double-buffering, and write-update and write-invalidate protocols. Furthermore, we characterize the performance of TCC in comparison to conventional snoopy cache coherence (SCC) using parallel applications optimized for each scheme. We conclude that the two coherence schemes perform similarly, with each scheme having a slight advantage for some applications. The bandwidth requirements of TCC are slightly higher but well within the capabilities of CMP systems. Also, we find that overflow of speculative state can be effectively handled by a simple victim cache. Our results suggest TCC can provide its programming advantages without compromising the performance expected from well-tuned parallel applications.

1. Introduction

Parallel processing has reached a critical juncture. On one hand, with ILP techniques running out of steam, chip multiprocessors (CMPs) are becoming the norm. Every major processor vendor has announced a CMP product [22, 23, 27]. On the other hand, writing correct and efficient multithreaded programs with conventional programming models is still an incredibly complex task limited to a few expert programmers. Unless we develop models that make parallel programming the common case, the performance potential of CMPs will be limited to multiprogramming workloads and a few server applications.

Transactional Coherence and Consistency (TCC) uses *continuous transactional execution* to simplify parallel programming [14]. Unlike previous proposals using transactional memory merely for non-blocking synchronization, TCC uses programmer-defined transactions as the basic unit of parallel work, synchronization, memory coherence, and consistency. In TCC, transactions continuously execute in a speculative manner, using the execution model illustrated in Figure 1. Each transaction is a sequence of instructions guaranteed to execute atomically. During execution, writes are buffered locally, then committed to shared memory atomically at transaction completion as well as broadcast to other transactions in the system. Other transactions snoop these broadcasts to maintain cache coherence and detect when they have used data that has subsequently been modified by the committing transaction—a dependence violation that causes transactional re-execution.

Continuous transactional execution simplifies shared-memory parallel programming in four ways [14]. First, it provides programmers with a single, high-level abstraction to reason about parallelism, communication, consistency, and failure atomicity. A single clear abstraction is key to an intuitive programming model. Second, transactions allow speculative parallelization of sequential algorithms without the need for the programmer to prove independence or manually handle rare dependencies. Third, transactions allow the user to specify the high-level atomicity and ordering requirements in parallel algorithms, but move the burden of implementing these requirements to hardware and system software. Locking conventions and the associated races and deadlocks are eliminated. Finally, as hardware continuously monitors the progress of concurrent transactions, there is an opportunity for low-overhead collection of profile data for feedback-driven or dynamic optimizations [7].

Early trace-driven experiments of TCC-based multiprocessors with idealized caches have shown good parallel performance potential for a variety of applications [16]. In this paper, we provide the first thorough, execution-driven char-

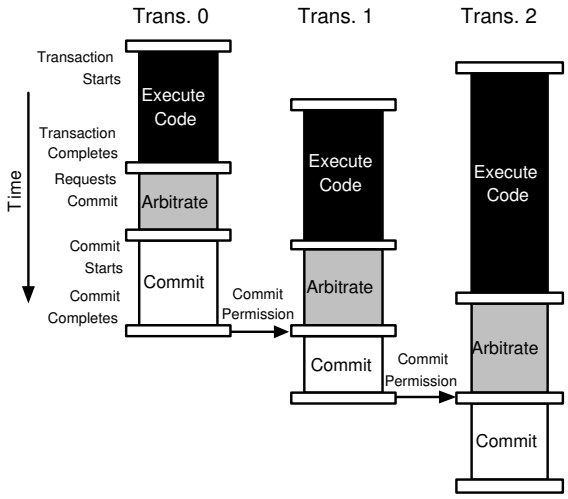

Figure 1: Execution timeline of three transactions in a TCC system.

acterization of continuous transactional execution on CMPs and make the following contributions:

- **Performance Comparison on CMPs.** A straightforward implementation of TCC performs and scales similarly to conventional cache coherence schemes. TCC has a performance advantage for difficult-to-tune applications with irregular communication patterns.
- **Bandwidth Requirements.** The commit bandwidth requirements for TCC are only slightly higher than with conventional schemes, but well within the capabilities of bus-based interconnects, even for a CMP with 16 processors.
- **Buffering Requirements.** The common buffering requirements for transactional execution are modest and can be satisfied by first-level caches. On the other hand, associativity overflows pose a frequent bottleneck, but can be virtually eliminated with a simple victim cache.
- **Design Alternatives.** We evaluate three TCC design alternatives: choice of snooping protocol (invalidate or update), single vs. double buffering, and word vs. cache line granularity for speculative state tracking. Experiments show performance is not significantly impacted by choice of protocol or buffering scheme. However, word-level granularity is clearly beneficial for applications with fine-grain sharing patterns.

Our results prove that continuous transactional execution is practical to implement for CMP systems. TCC provides for easier parallel programming without compromises in the peak performance possible for each application.

The rest of this paper is organized as follows. Section 2 describes an implementation for a TCC-based CMP system. In Section 3, we provide a qualitative comparison between TCC and conventional cache coherence. Section 4 presents our experimental methodology. In Section 5, we present the evaluation results. Section 6 discusses related work and Section 7 concludes the paper.

2. Transactional Coherence and Consistency

The conventional wisdom for supporting transactional memory is to overlay it on top of the already complex coherence and consistency protocols [26, 30, 29, 33]. Additional states are necessary in the MESI protocol and extra rules are needed for load-store ordering. In contrast, TCC directly implements transactional memory with optimistic concurrency. This mechanism is also sufficient to provide cache coherence and memory consistency at transaction boundaries without additional complications. In this section, we describe a straight-forward implementation of the TCC execution model for CMP systems. The design supports continuous execution of transactions by providing a means for buffering and committing the transactional state produced by each processor.

The CMP organization we study is similar to those in previous studies [6, 15, 23]: a number of simple processors with private L1 caches sharing a large, on-chip L2 cache. The processors and L2 are connected through split-transaction commit and refill buses. The two buses provide high bandwidth through wide data transfers and bursts. The commit bus has de-multiplexed address and data lines and is used to initiate L2 accesses (address only) and to commit their stores at the end of each transaction (address and data). The refill bus is used to transmit refill data from the L2 to the processors (data only). Both buses are logical but not physical buses, providing serialization and broadcast. To support high bandwidth, they are implemented using a star-like topology with point-to-point connections that supports pipelined operation for both arbitration and transfers [20].

2.1 Transactional Buffering

Each processor buffers the addresses and data for all stores within a transaction (the write-set) until it commits or aborts. The processor simultaneously tracks all addresses loaded within a transaction (the read-set) in order to detect dependency violations when other transactions commit their write-sets. We store both read- and write-set information in the L1 data cache because it provides high capacity with support for fast associative searches [12, 38, 24]. It also allows speculative and non-speculative data to dynamically share the storage capacity available in each processor in the most flexible way. Cached data become non-speculative when the transaction that fetched or produced them commits.

The cache stores data at the granularity of lines, but we can track the transaction read-set and write-sets at the granularity of lines or individual words. Figure 2 presents the L1 data cache organization for the case with word-level speculative state tracking. Cache lines include valid,

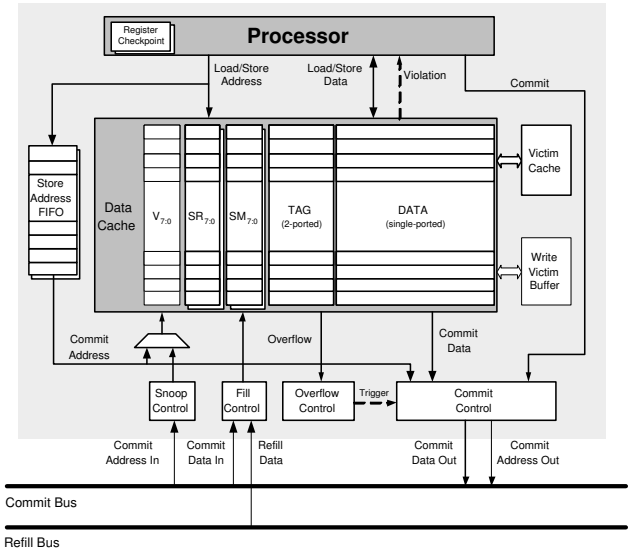

Figure 2: The data cache organization for transactional coherence. Double buffering requires additional hardware: a write victim buffer, duplicate SR/SM bits, an extra store address FIFO, and another register checkpoint. Shown is the configuration for word-level speculative state tracking.

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 by a store during the currently executing transaction. Similarly, the SR bit indicates that its word has been read by the current transaction. SR is set on loads, unless the SM bit is already set, to implement memory renaming and eliminate dependency violations due to WAW and WAR hazards across transactions[1]. Line-level speculative state tracking works in a similar manner but requires a single valid, SR, and SM bit per line. We discuss the tradeoffs of each approach in Section 2.6.

2.2 Transactional Commit

To commit a transaction, a processor first arbitrates for permission, then broadcasts its write-set to the lower levels of the memory hierarchy. Meanwhile, all other processors snoop the committed addresses and data to detect potential dependency violations and update or invalidate the contents of their caches to maintain coherence. System-wide arbitration guarantees that all processors see commits in the same order (serializability). Hence, memory consistency is maintained at transaction boundaries without rules for ordering of individual loads and stores.

[1]To handle renaming correctly in the presence of byte or halfword loads and stores, we set both the SR and SM bits on subword stores. This approach may generate a few unnecessary dependency violations in the presence of byte-level false sharing.

TCC allows programmers to define ordered, partially ordered, and unordered transactions, with ordering enforced during commit arbitration [14]. Therefore, acquiring commit permission may impose significant wait time on an application with ordered transactions, as younger transactions must always wait for older transactions to commit first. Even with unordered transactions, significant delays can occur if multiple transactions attempt to commit within a small period of time because of bus contention.

The processor identifies the transaction write-set in the data cache with help from the store address FIFO (SAF). This is a non-associative, tagless, single-ported buffer containing pointers to speculatively modified cache lines. For a 32-KB cache with 32-byte lines, the SAF requires 1024 entries with 10 bits per entry. So, the SAF area is small compared to that of the cache. During transaction execution, stores check the SM bits of the corresponding line in parallel with the tag comparison. If all SM bits are 0, this is the first speculative store to the line in this transaction and a pointer is inserted in the SAF. If an SM bit is set, the line address is already registered in the SAF. During commit, we read the SAF pointers one by one and flush out the speculatively modified words in the identified cache lines. The cache line address and a bitmask with the SM bits are transmitted along with the dirty words. The transmission itself may take multiple cycles, depending on the bus width and the number of modified words. After the write-set is committed, we reset all SM and SR bits to indicate that the corresponding lines/words are no longer speculative.

All other processors use a second port in the cache tags to snoop the committed addresses and detect any dependency violations for the transactions they currently execute. A violation occurs when the an executing transaction has speculatively read one of the words committed (i.e., there is a match for the snoop tag access and the word has the SR bit set in the cache). The processor must also check pending accesses in the MSHRs. When a violation is detected, speculatively modified lines are invalidated, SR and SM bits are cleared, and the transaction restarts.

2.3 Invalidate vs. Update

A processor must revise the state of any line it caches if it is modified by a committing transaction in order to maintain cache coherence. One can use an *update* or *invalidate* protocol to handle snoop matches: update replaces the old data value with the new committed value, while an invalidate protocol evicts any data modified by the committing transaction. An update protocol is possible for this CMP system as the committing processor is already placing the data on the commit bus for the L2 cache. The main disadvantage is that updates share the single data port in the L1 caches with the processors. Therefore, occasional stalls occur during load and store accesses from those processors.

An invalidate protocol is more complicated as we may need to invalidate a word in a line with other speculatively modified or read words. We cannot simply invalidate the whole line because we would lose part of the transaction read-set or write-set. A solution is to provide per-word valid bits, which introduces storage overhead and complicates miss handling. The performance difference between the two protocols depends on the percentage of updated or invalidated words that are later accessed by the processor. A large percentage gives a performance advantage to the update protocol.

2.4 Buffer Overflow

Buffer overflows happen when the L1 can no longer capture the write- and read-sets of the executing transaction. They occur either because the capacity or the associativity of the cache is exhausted. Overflows cause a major performance bottleneck for transactional execution regardless of how they are handled. One can switch to software transactional buffering [18, 17] or overflow to a special region in the lower levels of the memory hierarchy that has hardware support for speculative state [10, 4, 31]. In any case, the extended buffers are slower than the L1 data cache for searching, committing, or flushing speculative data. In this study, we use a simple technique to handle overflows [16]. When an overflow is detected, the hardware must commit the current write-set of the transaction, potentially waiting a long time for commit permission due to ordering constraints. No other transaction is then allowed to commit until the overflowing transaction has reached its regular commit point. Essentially, this serializes the system even if all processors execute unordered transactions.

In Section 5, we demonstrate that capacity overflows are extremely rare for processors with 32-KB caches. An overflow handling mechanism is still necessary for the uncommon case, but its performance is non-critical. On the other hand, associativity overflows are more frequent as there are often "hot" cache indices in each transaction where multiple cache lines with speculatively-read or -modified data are mapped. We address this issue with a victim cache [21] that stores the lines that would otherwise cause an associativity overflow (see Figure 2). Each line in the victim cache has the identical format as a line in the data cache. In Section 5, we demonstrate that an 8-entry, fully-associative victim cache eliminates virtually all associativity overflows in the applications we examined.

2.5 Double Buffering

So far, we assume a processor stalls while it commits a transaction before it starts the next one (see Figure 1). While simple, this approach cannot hide the latency of commit arbitration and write-set transmission. Double buffering can hide commit latency by allowing a processor to start executing the next transaction in parallel with committing its previous one. The hardware required for double buffering in the CMP implementation of TCC is shown in Figure 2: the cache maintains two sets of SR and SM bits to track the read- and write-sets for both transactions simultaneously, but only one copy of the data is kept. A second SAF is also needed. The two read-sets may intersect without any complications, but special care is necessary when both transactions write to same word. We must maintain both speculative versions of the word, as the younger transaction may violate while the older transaction commits without problems. We handle this case using a separate write victim buffer. When the younger transaction first overwrites a word modified by an older transaction, we copy the line into the write victim buffer and clear the SM bits in the cache for the older transaction. When the older transaction commits its write-set, it also commits any lines stored in the victim buffer. In our experiments, we do not limit the size of the write victim buffer in order to optimistically determine the value of double buffering.

Though resources required for double buffering are minimal in terms of area, the additional control complexity is significant as there are several corner cases (multiple versions of words, conflicts between transactions, violation and restart ordering). If commit latency is not a major bottleneck, which is often the case in a CMP environment with plentiful commit bandwidth, the occasional performance advantage of double buffering may not be worth the additional area overhead and control complexity.

2.6 Speculative State Granularity

So far, we have used word-level tracking of speculatively read and modified state. With fine-grain write-set information, we minimize commit bandwidth by sending only modified words. Additionally, word-level read-set tracking eliminates most false dependency violations—those between transactions that read and write different words in the same cache line. Alternatively, we can track speculative state at the cache line-level by keeping a single SR and SM bit per line. This saves die area, but causes spurious violations for two reasons: false sharing and the inability to resolve conflicts between a modified line and a commit to that same line. The latter case is handled smoothly by word-level tracking because we know which words are modified by the running transaction and we can skip them when updating/invalidating, thus avoiding WAW hazards.

2.7 Discussion

A few processor core modifications are needed for transactional execution. The processor must create checkpoints of architectural registers at the beginning of each transaction, which are restored in the case of a dependency violation. Two checkpoints are necessary for double buffering. Furthermore, the processor requires a few new instructions to trigger the register checkpoint and transaction commit.

The presented TCC-based CMP uses a straight-forward implementation of the TCC execution model. Several alternative implementations are possible and of interest especially for larger-scale systems: extended buffering in the L2 cache, commit in place to avoid data broadcast at transaction commit, directory-based schemes to reduce commit traffic, and two-phase schemes for parallel commit of independent transactions. We will explore these techniques in future work. Nevertheless, this implementation is sufficient to achieve good performance in a CMP environment and to analyze the virtues of transactional execution.

There are several operating system issues (virtualization, scheduling) and programming model challenges (nesting semantics, I/O) to explore with respect to continuous transactional execution. Several researchers have started exploring these issues [31]. This paper focuses exclusively on hardware feasibility and performance comparison: other challenges are not of interest if continuous transactional execution is prohibitive to implement or offers poor performance compared to conventional techniques.

3. Transactional vs. Snoopy Coherence

The TCC architecture in Section 2 provides processors with cache-coherent access to shared memory. However, there are distinctive differences between transactional cache coherence (TCC) and conventional snoopy cache coherence (SCC) using a protocol like MESI [8].

Time Granularity & Bandwidth: TCC maintains coherence at transaction boundaries. Until transactions commit, caches may temporarily contain incoherent data. Coherence events involve the whole write-set, even if only a fraction of it is actually shared across processors. On the other hand, SCC maintains coherence at a fine time granularity—on every load and store issued by any processor. Coherence events such as invalidations or ownership requests handle a single cache line at the time. However, SCC generates coherence events only when cache lines are actually shared.

Address Granularity: TCC can track coherence at a fine granularity with respect to addresses. Data are brought into the cache at the granularity of cache lines but state can be tracked at the granularity of individual words. Fine-grain state tracking is necessary to reduce commit bandwidth requirements and eliminate expensive false sharing violations during speculative execution. In contrast, SCC typically tracks data state and ownership at the granularity of whole cache lines, in order to limit the frequency and overhead of coherence events.

Speculation: TCC executes transactions in parallel using speculation. A transaction loads and stores data speculatively, even though the actual state and validity of data may not be determined until the transaction commits. Speculation allows TCC to support multiple concurrent writes without inter-processor communication at the time the writes occur. SCC, in its base form at least, is non-speculative. Only one processor may have write access to a cache line at any point in time, so multiple writes are serialized.

Synchronization: TCC inherently supports non-blocking atomic execution of operations grouped into a single transaction, regardless of the number of memory objects they process. On the other hand, SCC communication requires additional lock variables accessed with atomicity instructions (load-linked and store-conditional, test-and-set, compare-and-swap, etc.) to implement mutual exclusion. Furthermore, a consistency protocol is necessary to order ordinary and special loads and stores across processors.

The following sections discuss how the semantic differences between TCC and SCC translate to performance advantages or bottlenecks within the CMP environment. Section 5 presents quantitative data that evaluate the practical importance of each issue.

3.1 Snoopy Coherence Advantages

SCC has a performance advantage over TCC for applications with fine-grain, high-frequency communication. Data can be transferred between SCC caches as soon as the communication is detected and, apart from interconnect latency, there is no additional performance penalty. With TCC, on the other hand, fine-grain communication between transactions can lead to significant performance loss due to violations (large transactions) or excessive overhead (small transactions). However, this is not a major handicap because applications with fine-grain communication patterns tend to scale poorly on all parallel systems.

Transaction commit poses a performance challenge in TCC even for applications with coarse-grain communication. Without double buffering, the processor is idling while arbitrating for and committing a transaction. The arbitration time depends on transaction ordering, on the number of processors arbitrating concurrently, and on bus usage by other transactions. Double buffering can hide some of this latency, but its benefits are limited by two factors. First, the arbitration and commit time of the older transactions is not always perfectly balanced with the execution time of the second transaction. Second, the two transactions share a single cache and a single interface to the bus interconnect.

Commits can also expose a bandwidth bottleneck for TCC systems. With SCC, data produced by a processor are only transmitted on inter-processor communication or when they no longer fit in the cache. With the simple TCC implementation presented, the whole write-set of a transaction is transmitted, regardless of communication patterns and caching behavior. Even for a CMP environment, bandwidth may become an issue for configurations with large processor counts.

Finally, TCC performance is affected by the frequency of dependency violations and overflows. Dependency viola-

tions have a dual negative effect. First, they waste valuable time on a processor, especially if they occur toward the end of the transaction. Second, before a transaction violates, it generates cache misses that consume cycles on the commit and refill buses, delaying other transactions that are executing useful work. Overflows cause additional commits and may serialize the system.

3.2 Transactional Coherence Advantages

TCC supports speculative parallelization of irregular applications for which data independence cannot be fully proven and communication patterns are difficult to understand [32]. As long as the programmer can correctly identify transactions and their commit order, the CMP hardware executes the code correctly and achieves speedup if parallelism is available [14].

Every TCC transaction is inherently a non-blocking, atomic task on multiple memory objects. Hence, TCC can facilitate parallel access to shared data without incurring the runtime overhead of fine-grain locking. Furthermore, lock-based synchronization easily leads to races or deadlocks as programmers associate the wrong lock (or no lock) to a shared object or acquire locks in an unsafe order.

TCC can eliminate the SCC performance penalty for false sharing. By tracking the transaction read- and write-sets at word granularity, a TCC-based CMP can identify when two transactions update different words in the same cache line and avoid unnecessary dependency violations. On the other hand, SCC incurs inter-processor communication for ownership upgrades and downgrades on false sharing. Word-granularity state tracking also allows TCC to implement memory renaming and eliminate violations on output and on anti-dependencies between transactions. With SCC, output and anti-dependencies between threads must be explicitly handled with synchronization.

Despite the potentially high bandwidth requirements for commits, TCC can utilize interconnect bandwidth better than SCC. Transaction write-sets range from tens to thousands of words, hence large burst transfers on wide buses can efficiently move the data to the L2 cache and other processors, amortizing the latency of bus arbitration. In contrast, all SCC transfers involve a single cache line, making it difficult to amortize arbitration latency or use buses wider than a cache line.

4. Methodology

We evaluate the performance of CMPs with the proposed TCC implementation or the MESI snoopy cache coherence (SCC) using an execution-driven simulator modeling PowerPC processors. All instructions, except loads and stores, have a CPI of 1.0. The memory system models the timing of the L1 caches, the shared L2 cache, and the commit and refill buses. All contention and queuing for accesses to caches and buses is modeled. In particular, we model the contention for the single data port in the L1 caches, which is used for processor accesses and commits (TCC with update protocol) or cache-to-cache transfers (SCC). Table 1 presents the main parameters for the simulated CMP architecture. The default configuration for TCC uses an invalidate protocol, an 8-entry victim cache, and single buffering.

Feature	Description
CPU	1–16 single-issue PowerPC cores (8)
L1	32-KB, 32-byte cache line 4-way associative, 1 cycle latency
Victim Cache	0–32 entries fully associative (8)
Double Buffering	(Off) [TCC only]
Bus Width	16 bytes
Bus Arbitration	3 pipelined cycles
Transfer Latency	3 pipelined cycles
L2 Cache	8MB, 8-way 16 cycles hit time
Main Memory	100 cycles latency up to 8 outstanding transfers

Table 1: Parameters for the simulated CMP architecture. Unless indicated otherwise, results assume the default values in parentheses. Bus width and latency parameters apply to both commit and refill buses. L2 hit time includes arbitration and bus transfer time.

With SCC, the CMP uses the commit bus to initiate L2 refills and issue invalidate or upgrade requests. The refill bus is used for replies from the L2 cache and for cache-to-cache transfers. The victim cache is used for recently evicted data from the L1 cache. We always use an invalidation-based MESI protocol for SCC because it generates less inter-processor traffic and is more widely used than update-based protocols in bus-based multiprocessor systems [39].

We compare the two coherence schemes using nine parallel applications: `equake`, `swim`, and `tomcatv` from the SPEC CPUFP suite [36]; `barnes`, `mp3d`, `ocean`, `radix`, and `water-nsquared` (called simply `water`) from the SPLASH and SPLASH-2 parallel benchmark suites [34, 40]; and `SPECjbb2000` [37]. `SPECjbb` was only used for the CPU scaling experiments and ran on top of the Jikes RVM [3]. We chose these applications because they are representative of important workloads and their code has been heavily optimized for conventional SCC multiprocessors by the research community over a period of years. It includes optimizations to reduce synchronization, false sharing, and the impact of communication latency. Even though transactional execution with TCC allows us to speculatively parallelize applications in a manner that is beyond the capabilities of conventional parallelization techniques, we wanted to ensure we did not penalize the performance of SCC by using such applications. This study focuses on sustained performance and not on ease of programming or tuning.

We ported the applications to TCC using the following

Application	Trans. Size 90th % (Inst)	Trans. Wr. Set 90th % (KB)	Trans. Rd. Set 90th % (KB)	Ops. per Word Written 90th %
barnes [40] (2048 part.)	2,722	0.34	.80	62.1
equake [36] (ref.)	10,233	.35	2.1	129.7
mp3d [34] (3,000 mol.)	748	.34	.40	7.8
ocean [40] (258x258)	1,293	1.0	2.5	39.4
radix [40] (262,144 keys)	4,373	1.4	2.6	17.5
swim [36] (ref.)	3,876	2.9	6.1	8.1
tomcatv [36] (ref.)	751	0.66	.91	8.3
water [40] (512 mol.)	927	0.42	0.46	8.9
SPECjbb [37] (368 trans.)	50,556	2.44	1.38	143.6

Table 2: Applications used for performance evaluation. The 90th percentile transaction size, measured in instructions, the 90th percentile transaction write- and read-set sizes in KBytes, and the 90th percentile of operations per word written.

process. For SPEC and SPLASH, we converted the code between barrier calls into unordered transactions, discarding any lock or unlock statements. For `SPECjbb2000`, we converted the 5 application-level transactions into unordered transactions. Next, we used profiling information to identify and tune performance bottlenecks as described in [14]. The tuning process typically lasted a few hours per benchmark and produced code optimized for transactional execution with TCC.

5. Performance Evaluation

This section presents the quantitative evaluation of continuous transactional execution with TCC in a CMP system. First, we analyze applications to identify the ranges of runtime behavior TCC hardware must efficiently support. Next, we explore TCC implementation alternatives. Finally, we compare the performance of TCC to SCC as we scale the processor count.

5.1 Transactional Application Analysis

Table 2 shows the characteristics of our applications under TCC. These characteristics depend on the transactional programming model and the optimization level (see [14] for a detailed discussion), but not on the hardware parameters. Transaction sizes range from a half- to fifty-thousand instructions, which provides adequate work to hide the overhead of starting a new transaction (approximately 8 instructions given a preallocated stack per CPU). Table 2 also shows statistics for read- and write-sets sizes in the nine applications. The read-set size for all transactions is less than 7 KB, while the write-set never exceeds 4 KB. These maximum sizes imply a 32-KB L1 cache can capture the read- and write-sets for these applications without *any* capacity overflow. While a software-based overflow mechanism is necessary to handle rare capacity overflows, the common case for continuous transactional execution with these applications is easy to handle with fast hardware techniques. We discuss associativity overflows in the following section.

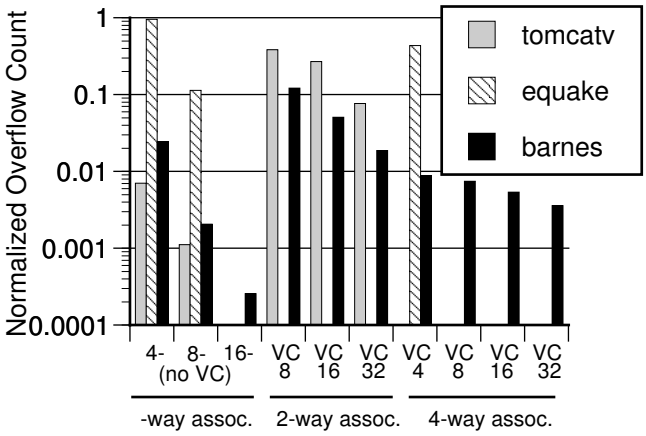

Figure 3: Associative overflows for tomcatv, equake, and barnes, varying cache configuration. Only associativity is varied on the first three plots (4-, 8-, and 16-way). The remaining plots vary both associativity and victim cache (VC) entries. The vertical axis is logarithmic and normalized to 2-way associative, no victim cache.

Furthermore, Table 2 presents the ratio of operations per word in the write-set. The ratio ranges from 8 to 143, depending on the transaction size and the store locality exhibited for each application. This ratio is important because it indicates the bandwidth requirements necessary to commit data and the commit overhead. Applications with a high ratio such as `equake` can commit quickly even when bandwidth is scarce. Applications with a low ratio like `mp3d` put pressure on commit bandwidth and experience performance loss due to bus contention. Nevertheless, the bus utilization depends both on commit (writes) and L1 miss (reads) traffic. If the latter is low, a greater need for commit bandwidth can be tolerated.

5.2 TCC Design Space Analysis

There are four major design choices to explore in the CMP implementation of TCC: associativity overflow handling, the use of an update or invalidate protocol for commit, the use of single or double buffering, and word- vs. line-level speculative state tracking.

Unlike capacity overflows, associativity overflows can

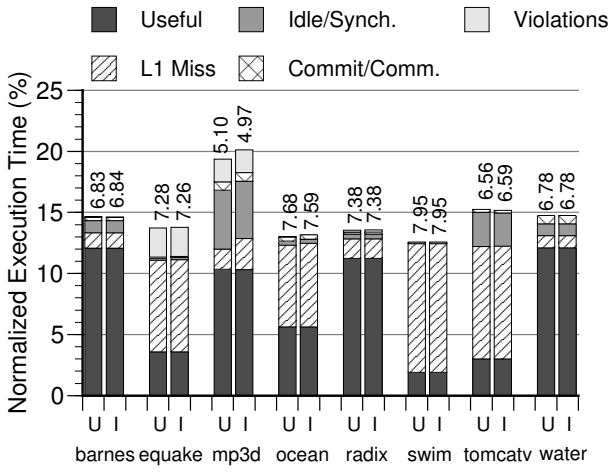

Figure 4: Normalized execution time for TCC with an invalidate (I) and an update (U) protocol for 8 processors. Execution time is normalized to sequential execution. Speedups are printed above each bar.

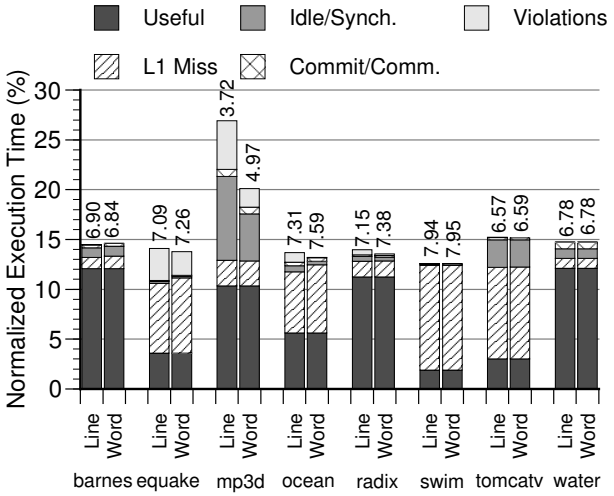

Figure 5: Normalized execution time for TCC with line- and word-level state tracking. Speedups are printed above each bar.

be common even with small read- and write-sets (e.g., `tomcatv`). Figure 3 shows the number of associative overflows in `tomcatv`, `equake`, and `barnes` when varying the L1 cache associativity and the victim cache size. The count is normalized to runs with a 2-way set-associative cache and no victim cache. Several applications did not experienced associativity overflows at all, but with a 4-way cache, the performance of the three applications in Figure 3 was significantly impacted by costly overflows. A 16-way cache virtually eliminates overflows, but such a design would have a negative impact on clock frequency. The introduction of a fully-associative victim cache can alleviate the problem without increasing the cache associativity. A 4-way cache with an 8-entry victim cache eliminates the vast majority of associativity overflows, with only 1% of `barnes`'s overflows remaining. A 2-way cache is not sufficient even with a 32-entry victim as there are a large number of sets within transactions that require 3-way to 4-way associativity and exceed the capacity of the victim cache. The remaining performance results assume a 4-way associative cache with an 8-entry victim cache.

Figure 4 compares the performance of an update and invalidate protocol for TCC commit in an 8 CPU system. Depicted is execution time normalized to the sequential versions of the applications (lower is better). Execution time has five components: *Useful* time spent executing instructions and the TCC API code, *L1 Miss* time spent stalling on loads and stores, *Commit* time spent waiting for the commit token and committing the write set to shared memory, *Idle* time spent idle due to load imbalance, and finally time spend due to *Violations*. The choice of protocol has little impact on performance. Update provides minor performance benefits in applications where updated words are subsequently accessed. An invalidate protocol would force these *re-touched* words to be re-loaded from the memory hierarchy. However, only two of our applications exhibited such behavior: `equake` and `mp3d`. For `mp3d`, update achieved a performance gain of only 2.5% over invalidate. For `equake` the difference is negligible as its high miss rate (4.28%) masks all other performance bottlenecks. The performance results use the invalidate protocol as we compare against an invalidate-based SCC.

Figure 5 shows normalized execution time with word- and line-level speculative state tracking. For most applications, performance was nearly identical. However, for applications with significant false sharing or cache-to-cache transfers, like `mp3d`, there are additional spurious violations that negatively impact performance. Since it facilitates a more robust programming environment, the remaining results use word-level granularity.

Double buffering can hide commit latency at the expense of additional complexity. We do not present extensive results due to space limitations, but double buffering is somewhat useful for applications exhibiting significant commit overhead, such as `mp3d` (6% improvement) and `water` (5% improvement) that have a low operations per word written ratio. The remaining applications averaged a 1.9% improvement because they either have high operations per word written ratios or do not otherwise stress the available bandwidth in the CMP system. The remaining results use single buffering.

5.3 TCC vs. SCC

Figure 6 compares traditional snoopy cache coherence (SCC) with transactional coherence and consistency (TCC)

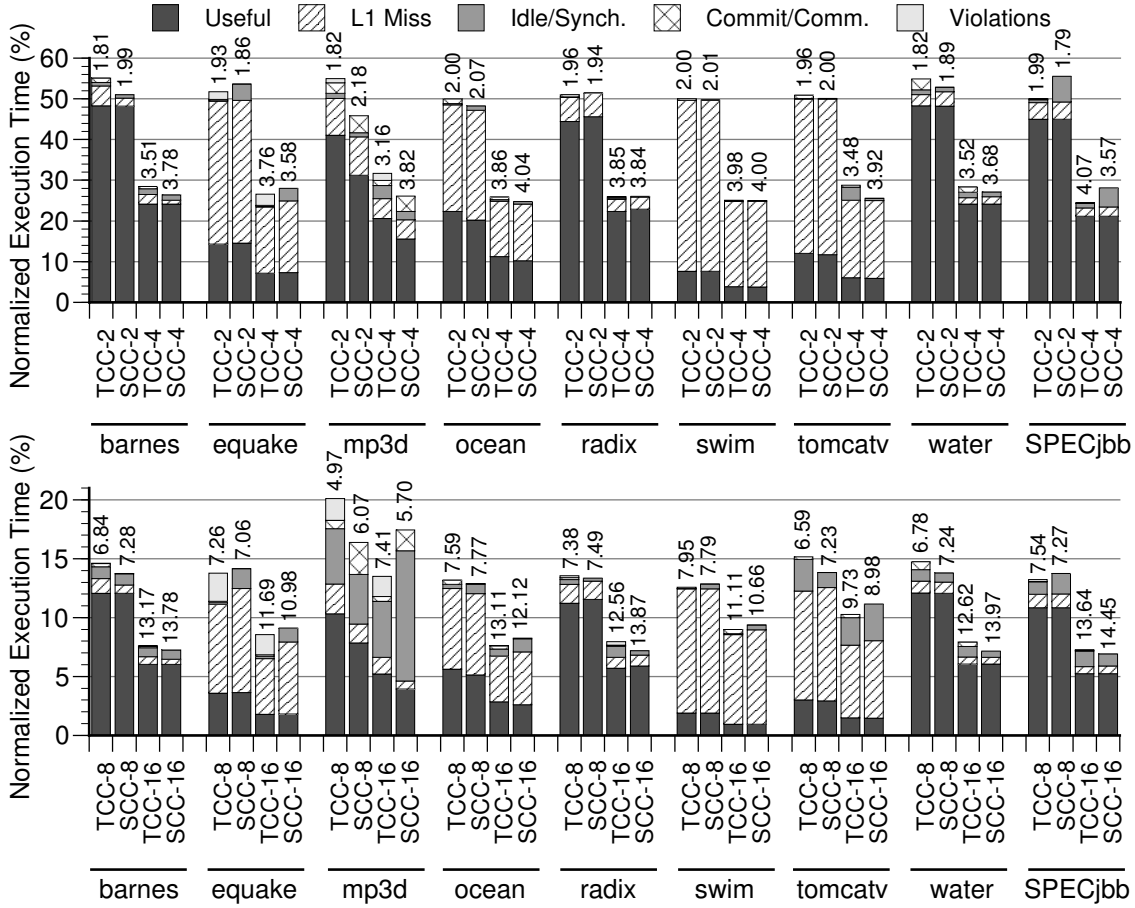

Figure 6: Normalized execution time for SCC and TCC as we scale the number of processors from 2 to 16. Parallel execution times are normalized to that of a single processor running the original sequential code. The top graph contains runs for 2 and 4 processors and the bottom graph contains the 8 and 16 processor runs; values on the vertical axis change appropriately. Speedups are printed above each bar.

as we scale the number of processors from 2 to 16. It shows execution time normalized to sequential applications (lower is better). Each TCC bar is broken into five components as described in Section 5.2. The SCC bars are slightly different: *Synchronization* is time spent in barriers and locks, and *Communication* is stall time for cache-to-cache transfers. SCC bars do not have violations. Note that the applications are optimized individually for each model.

In general, SCC and TCC perform and scale similarly on most applications for up to 16 processors. This demonstrates that continuous transactional execution does not incur a significant performance penalty compared to conventional techniques. Hence, it is worthwhile exploring the advantages it provides for parallel software development. For some applications, as the number of processors increase, time spent in locks and barriers make SCC perform poorly. TCC also loses performance on some applications, but the reasons vary. The difference between TCC and SCC is generally small, but each application exhibits interesting characteristics:

barnes: It scales well on both TCC and SCC as it only has a small amount of communication between processors. `barnes` has a high operations per word written ratio, which helps TCC amortize the time spent communicating.

equake: The SCC version of `equake` has significant synchronization overhead caused by fine-grained locking to regulate access to a sparse matrix [28]. The TCC version does not require fine-grain lock insertion, but suffers from occasional violations. `equake` is a good example of how the simple TCC programming model provides performance in the face of infrequent sharing.

mp3d: `mp3d` has a significant amount of communication and false sharing. We use it as an example of an irregular parallel program that is difficult to tune. `mp3d` scales well up to 8 processors on both architectures, but the time spent on barriers limits SCC scaling beyond 8 processors. In con-

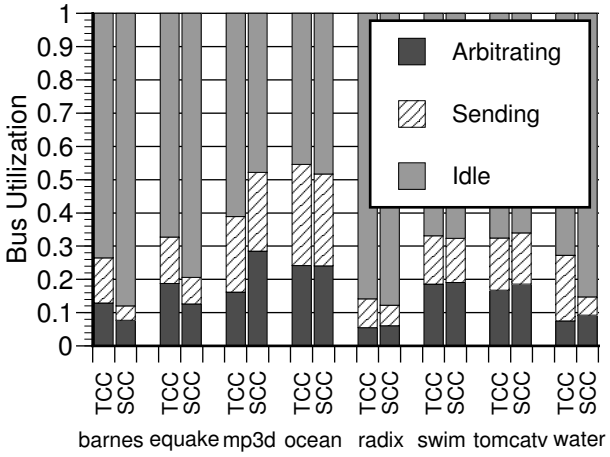

Figure 7: Bus utilization breakdown for TCC and SCC with 8 processors. The bus is 16-bytes wide.

trast, TCC essentially speculates through barriers and scales reasonably well up to 16 processors, despite the relatively low operations per word written ratio. `mp3d`'s useful time varies between TCC and SCC because of TCC API overhead.

ocean: TCC and SCC perform similarly, but at 16 processors time spent in barriers begins to somewhat hinder scalability on an SCC architecture.

radix: On SCC, `radix` scales well because of its low miss rate and lack of barrier synchronization. The TCC version suffers from load imbalance.

swim: Its execution time is dominated by the high L1 cache miss rate (above 9.15%). `swim` scales similarly with both architectures and with 16 processors is limited by data cache misses that saturate the bus to the L2 cache.

tomcatv: SCC performs better for up to 8 processors due to contention for the commit bus: `tomcatv`'s transaction sizes are small and lead to frequent commits. With 16 processors, SCC's performance begins to lag due to synchronization time spent in barriers; TCC's speculative mechanisms avoid some of this delay.

water: `water` has a tiny miss rate of 0.72%, which ensures that both SCC and TCC scale well to 16 processors. Time stalling for commit poses a small problem for TCC, because the average transaction size is small at 927 instructions and the write state is relatively large at 430 bytes.

SPECjbb: Both TCC and SCC scale acceptably. TCC shows some load imbalance, whereas SCC spends significant time in locks. TCC's scalability at 16 processors is hindered by overflows (shown as idle time)—more optimization could remove the remaining bottlenecks.

Figure 7 presents the bus utilization for TCC and SCC with 8 processors. Bars are split into three regions: one representing the time for transmission of load, store and commit data; one bar for arbitration time; and a final bar for idle time. Though TCC uses additional bandwidth to commit all data written, whether they are truly shared or not, the bandwidth utilization never exceeds 55%. Even a bus-based interconnect has sufficient bandwidth for this straight-forward implementation of transactional execution. In `water` and `barnes`, misses and L1 writebacks are rare, so committing all written data makes TCC's bandwidth usage much higher than SCC's. SCC consumes more bandwidth in `mp3d` because of its cache-to-cache transfers. `equake` has violations that cause additional memory traffic in TCC.

We also analyzed the performance of SCC and TCC as we varied the available bandwidth on the bus (8, 16, or 32 bytes per cycle) and the bus pipelined latency for arbitration and transfer (1, 2, or 6 clock cycles). Neither affected the comparison between TCC and SCC significantly. Both schemes scaled similarly in all cases.

5.4 Discussion

In Section 5.3, we compare TCC to the conventional SCC scheme with the MESI protocol [8]. Recently, researchers have proposed two extensions to SCC that target some of its performance bottlenecks.

Coherence decoupling (CD) [19] allows a processor to use invalid data in the cache while coherence messages are exchanged over the interconnect. This reduces the performance impact of false sharing and silent stores. The CD evaluation shows a 1% to 12% performance improvement for SPLASH applications on a 16 processor SMP. Speculative Synchronization (SS) [30, 26] allows a processor to speculatively proceed past locks and barriers and achieve the benefits of optimistic non-blocking synchronization. It provides a 5% to 25% performance improvement for SPLASH applications on a 16 processor SMP [30].

For the CMP environment in this study, false sharing, silent stores, and synchronization pose smaller performance challenges than with a SMP system, as the on-chip interconnect in a CMP has higher bandwidth and lower latency. Hence, the benefits from CD and SS in a CMP will be significantly lower and are unlikely to change the comparison between TCC and SCC. In addition, both CD and SS require significant additional complexity *on top* of snoopy cache control (speculative buffers and predictors). In contrast, TCC does away with all SCC hardware and implements the hardware presented in Section 2 to handle coherence, optimistic synchronization, and false sharing with a single mechanism.

The SCC model in this paper uses sequential consistency. Similarly all loads and stores from each processor in the TCC system are strictly ordered. The use of a relaxed consistency model would undoubtedly improve the performance of SCC [1]. Nevertheless, TCC would also improve as we can freely reorder loads and stores within each transaction. In fact, TCC with an out-of-order proces-

sor can be thought of as implementing release consistency, with transaction begin and end being acquire and release, respectively.

6. Related Work

There is a vast amount of prior research in the area of shared memory multiprocessor cache coherence. The research that is most relevant to this paper is concerned with transactional coherence and the performance of snoopy cache coherence protocols.

TCC uses transactions, a core concept in Database Management Systems (DBMS) [13] as a general programming construct. TCC relies on the idea of optimistic concurrency [25]: controlling access to shared data without locks by detecting conflicts and re-executing to ensure correctness. TCC extends these database ideas to memory [16], building on early transactional memory work done by Herlihy [18] as well as more recent work in Thread-Level Speculation (TLS) [35], and Stampede [38, 15, 24].

Most of the coherence schemes implemented in shared memory multiprocessors are based on snoopy cache coherence [11]. Archibald and Baer [5] provide a survey of snoopy protocols. Eggers [9] did an early study of data sharing in multiprocessors and Agarwal et al. [2] compare the performance of snoopy coherence schemes with directory based schemes. More recently, researchers have looked at combining speculation with conventional snoopy cache coherence. For example, Martinez and Torrellas [26], Rajwar and Goodman [30, 29], and Rundberg and Stenstrom [33] have independently proposed how to speculate through locks and barriers and Huh et al. have used speculation to reduce the effect of false sharing [19].

7. Conclusions

We investigated the implementation and performance of continuous transactions in TCC as compared to snoopy cache coherence for chip-multiprocessors. Using nine optimized parallel programs, we found that TCC's performance is comparable to that of conventional coherence and scales well to 16 processors. We also showed that that despite earlier evidence on the substantial bandwidth demands for TCC, bandwidth utilization on bus-based CMPs was not a hindrance to TCC scalability. We also studied implementation alternatives for TCC and showed that the choice of snooping protocol has little impact on performance, a simple victim cache can avoid most associative overflows, single buffering provides acceptable performance for most applications, and speculative state should be tracked at word level.

Overall, TCC provides excellent parallel performance for optimized applications in addition to the simpler parallel programming model.

8. Acknowledgments

This research was sponsored by 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.

Additional support was also available through NSF grant 0444470.

References

[1] S. V. Adve, V. S. Pai, and P. Ranganathan. Recent advances in memory consistency models for hardware shared memory systems. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):445–455, 1999.

[2] A. Agarwal, J. L. Hennessy, R. Simoni, and M. A. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, 1988.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[4] S. Ananian, K. Asanovic, et al. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, Feb. 2005.

[5] J. Archibald and J. L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation mode. *ACM Transactions on Computer Systems*, pages 273–298, Nov. 1986.

[6] L. Barroso, K. Gharachorloo, et al. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, June 2000.

[7] H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. TAPE: A transactional application profiling environment. In *Proceedings of the 19th ACM International Conference on Supercomputing*, June 2005.

[8] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture*. Morgan Kauffman, 1999.

[9] S. Eggers. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors*. PhD thesis, University of California, Berkeley, 1989.

[10] M. Garzaran, M. Prvulovic, et al. Tradeoffs in buffering multi-version memory state for speculative thread-level parallelization in multiprocessors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, Feb. 2003.

[11] J. R. Goodman. Using cache memory to reduce processor

memory traffic. In *Proceedings International Symposium on Computer Architecture*, pages 124–131, 1983.

[12] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[14] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency. In *Proceedings of the 11th International Conference on Architecture Support for Programming Languages and Operating Systems*, Oct. 2004.

[15] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE MICRO Magazine*, March–April 2000.

[16] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 102–113, June 2004.

[17] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

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

[19] J. Huh, J. Chang., D. Burger., and G. Sohi. Coherence decoupling: Making use of incoherence. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[20] JBus architecture overview. Technical report, Sun Microsystems, Apr. 2003.

[21] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the International Symposium on Computer Architecture*, May 1990.

[22] R. Kalla et al. Simultaneous multi-threading implementation in POWER5. In *Conference Record of Hot Chips 16*, Stanford, CA, Aug. 2003.

[23] P. Kongetira. A 32-way multithreaded Sparc processor. In *Conference Record of Hot Chips 16*, Stanford, CA, Aug. 2004.

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

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

[26] J. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[27] C. McNairy. Montecito: The next product in the Itanium Processor Family. In *Conference Record of Hot Chips 16*, Stanford, CA, Aug. 2004.

[28] D. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University, Oct. 1997.

[29] R. Rajwar and J. Goodman. Speculative Lock Elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[30] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[31] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[32] L. Rauchwerger and D. Padua. LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 1995.

[33] P. Rundberg and P. Stenstrom. Reordered speculative execution of critical sections. In *Proceedings of the 2002 International Conference on Parallel Processing*, Feb. 2002.

[34] J. P. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1).

[35] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

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

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

[38] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Las Vegas, Nevada, 1998.

[39] P. Sweazy and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *Proceedings of the 13th Symposium on Computer Architecture*, pages 1056–1072, 1986.

[40] S.Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.