# TRANSACTIONAL COHERENCE AND CONSISTENCY: SIMPLIFYING PARALLEL HARDWARE AND SOFTWARE

TCC SIMPLIFIES PARALLEL HARDWARE AND SOFTWARE DESIGN BY ELIMINATING THE NEED FOR CONVENTIONAL CACHE COHERENCE AND CONSISTENCY MODELS AND LETTING PROGRAMMERS PARALLELIZE A WIDE RANGE OF APPLICATIONS WITH A SIMPLE, LOCK-FREE TRANSACTIONAL MODEL.

Lance Hammond
Brian D. Carlstrom
Vicky Wong
Michael Chen
Christos Kozyrakis
Kunle Olukotun
Stanford University

With uniprocessor systems running into instruction-level parallelism (ILP) limits and fundamental VLSI constraints, parallel architectures provide a realistic path toward scalable performance by letting programmers exploit thread-level parallelism (TLP) in more explicitly distributed architectures. As a result, single-board and single-chip multiprocessors are becoming the norm for server and embedded computing, and are even starting to appear on desktop platforms. Nevertheless, the complexity of parallel application development and the continued difficulty of implementing efficient and correct parallel architectures have limited these architectures' potential.

Most existing parallel systems add hardware to give programmers an illusion of a single shared memory common to all processors. Programmers must divide the computation into parallel tasks, but all tasks work on a single data set resident in the shared memory. Unfortunately, the hardware required to support this model can be complex. To provide a *coherent* view of memory, the hardware must determine the location of the latest version of any particular memory address, recover the latest version of a cache line from anywhere within the system when a load from it occurs, and efficiently support interprocessor communication of numerous small, cache-line-sized data packets. It must do all this with minimal latency, too, because individual load and store instructions depend on each communication event. Further complicating matters is the problem of sequencing the various communication events constantly passing through the system at the fine-grained level of individual load and store instructions. For software synchronization routines to work, hardware designers must devise and correctly implement sets of rules known as *memory consistency models*. Over the years, these models have progressed from easy-to-understand but sometimes performance-limiting sequential consistency schemes to more modern schemes such as relaxed consistency.

The complex interaction of coherence, synchronization, and consistency makes the job



Published by the IEEE Computer Society

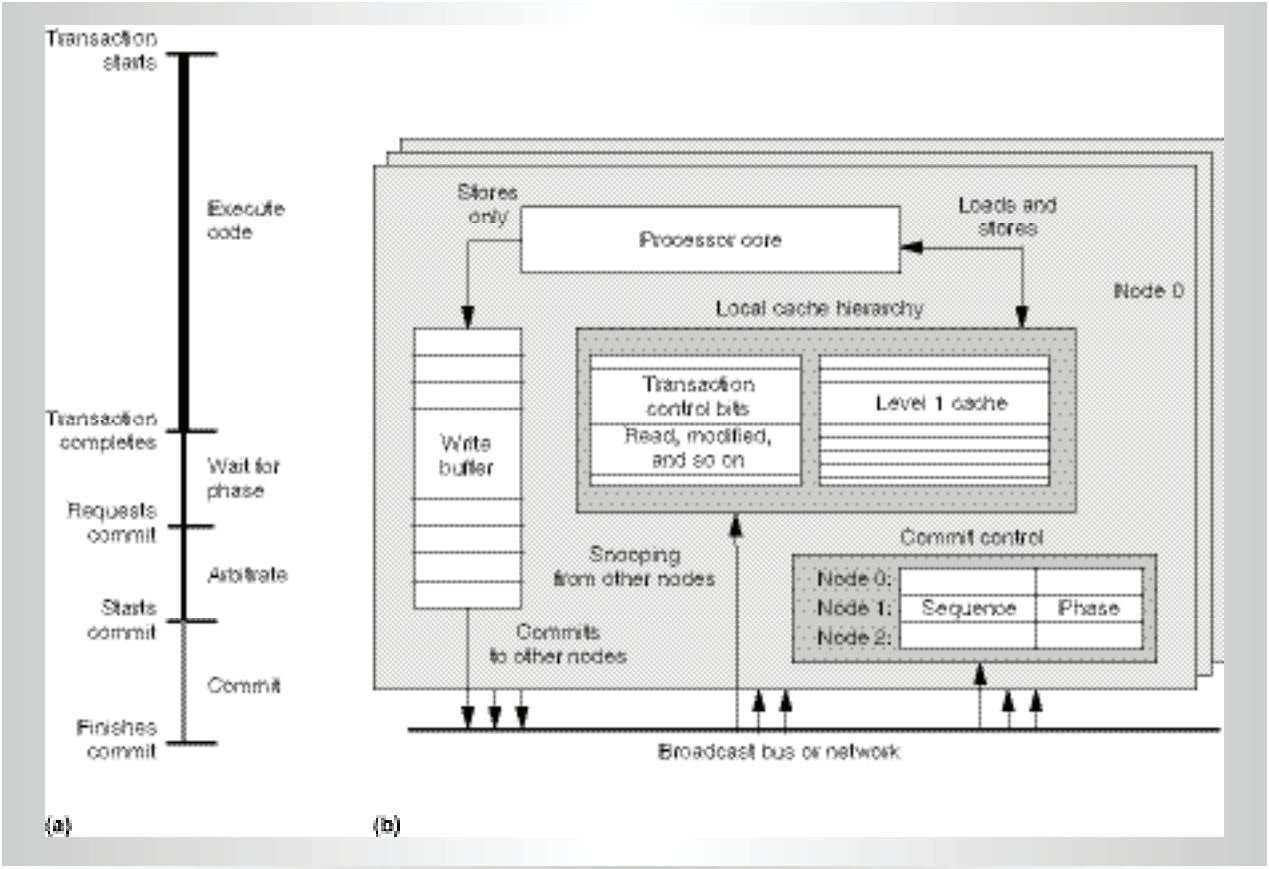


Figure 1. Transactional coherence and consistency (TCC) in hardware: the transaction cycle (a) and a diagram of sample TCC-enabled hardware (b).

of parallel programming difficult. Existing approaches require programmers to manage parallel concurrency directly by creating and explicitly synchronizing threads. The difficulty stems from the need to achieve the often conflicting goals of functional correctness and high performance. In particular, using a few coarse-grained locks can make it simpler to correctly sequence accesses to variables shared among parallel threads. On the other hand, having numerous fine-grained locks often allows higher performance by reducing the amount of time wasted by threads as they compete for access to the same variables, although the larger number of locks usually incurs more locking overhead.

Transactional coherence and consistency (TCC) simultaneously eases both parallel programming and parallel architecture design by relying on programmer-defined *transactions* as the basic unit of parallel work, communication, memory coherence, and memory consistency. Although several researchers have proposed using hardware transactions instead of locks or other parallel programming constructs individually,[4-6] TCC unifies these ideas into an "all transactions, all the time" model that allows significant simplifications of both parallel hardware[3] and software.[2] As parallel architectures become increasingly prevalent and a wider variety of hardware designers and programmers must deal with their intricacies, this advantage will increase TCC's importance.

## TCC hardware

Processors operating in a TCC-based multiprocessor execute speculative transactions in a continuous cycle (illustrated in Figure 1a) on multiprocessor hardware similar to that depicted in Figure 1b. A *transaction* is a sequence of instructions marked by software that is guaranteed to execute and complete only as an atomic unit. Each transaction pro-

duces a block of writes, which the processor buffers locally while the transaction executes, committing them to shared memory only as an atomic unit after the transaction completes. Once the transaction is complete, hardware must arbitrate system-wide for permission to commit the transaction's writes. After winning the arbitration, the processor can exploit high-bandwidth system interconnect to broadcast all of the transaction's writes to the rest of the system, as one packet. This broadcast can make scaling TCC to numerous processors a challenge. Meanwhile, other processors' local caches snoop on the write packets to maintain system coherence. Snooping also lets the processors detect when they have used data that another processor has subsequently modified—a *dependence violation*. Combining all of the transaction's writes imparts latency tolerance because it reduces the number of interprocessor messages and arbitrations and because flushing the writes is a one-way operation. The commit operation can also provide inherent synchronization for programmers and a greatly simplified consistency protocol for hardware engineers. Most significantly, this continual cycle of speculative buffering, broadcast, and (potential) violations lets us replace both conventional coherence and consistence protocols.

Consistency protocols can be simplified because communication can occur only at occasional, programmer-defined commit points, instead of at each of the numerous loads and stores used by conventional models. This has significant implications for both hardware and software. For hardware designers, TCC simplifies the design by drastically reducing the number of latency-sensitive arbitration and synchronization events that must be sequenced by consistency-management logic in a typical multiprocessor system. To programmers or parallelizing compilers, explicit commit operations mean that software can orchestrate communication much more precisely than with conventional consistency models. Simplifying matters further for most programmers, imposing an order on the transaction commits and backing up uncommitted transactions if they have speculatively read data modified by other transactions effectively lets the TCC system provide an illusion of uniprocessor execution. As far as global memory and software is concerned, all memory references from a transaction that commits earlier effectively occurred before all of the memory references of a transaction that committed later. This is true even if their actual execution was interleaved in time, because all writes from a transaction become visible to other processors only at commit time.

This simple, pseudo-sequential consistence model allows for a simpler coherence model, too. During each transaction, stores are buffered and kept within the transaction's processor node to maintain transaction atomicity. Processor caches do not use conventional modified/exclusive/shared/invalid (MESI)-style protocols to maintain lines in shared or exclusive states at any point, so many processor nodes can legally hold the same line simultaneously in either an unmodified or speculatively modified form. At the end of each transaction, the processor's commit broadcast notifies all other processors about its changed state. During this process, the other processors perform conventional invalidation (if the commit packet contains only addresses) or update (if it contains addresses and data) to keep their cache state coherent. Simultaneously, they must determine whether they have read from any of the committed addresses. If they have, they must restart and reexecute their current transactions with the updated data. This protects against true data dependencies. At the same time, because later processors' transactions do not flush out any data to memory until their own turn to commit, data antidependencies are not an issue. Until a transaction commits, transactions that commit earlier do not see its effectively later results (avoiding write-after-read dependencies), and processors can freely overwrite previously modified data in a clearly sequenced manner (handling write-after-write dependencies legally).

TCC will work in a wide variety of multiprocessor hardware environments, including various chip multiprocessor configurations and small-scale multichip multiprocessors. Within these systems, individual processor cores and their local cache hierarchies need features that provide speculative buffering of memory references and commit arbitration control. Most significantly, a mechanism for gathering all modified cache lines from each

transaction into a commit packet is required. This mechanism can be a write buffer completely separate from the caches or an address buffer that maintains a list of tags for lines containing data to be committed. The buffer must hold approximately 4 to 16 Kbytes worth of cache lines. If it fills during the execution of a long transaction, the processor must declare an overflow and stall until it obtains commit permission, when it can continue executing while writing its results directly to shared memory. Of course, this write-through behavior means that no other processors can commit while the transaction completes, to maintain atomicity. This effect can cause serious serialization if it occurs frequently, but is acceptable on occasion.

In the caches, all of the included lines must maintain the following information in some way:

- *Read bits.* Bits set on loads to indicate that a cache line (or portion thereof) has been read speculatively during a transaction. A processor's coherence logic snoops the read bits while other processor nodes commit to determine when the processor has speculatively read any data too early. If it snoops and sees a write committed by another processor modifying an address cached locally with its read bit set, it must violate its current transaction and restart.
- *Modified bits.* Every cache line has at least one modified bit. Stores set the modified bit to indicate when any part of the line has been written speculatively. The processor uses these bits to simultaneously invalidate all speculatively written lines when it detects a violation.

Additional bits can improve performance, but are not essential for correct execution. A processor cannot flush cache lines with set read bits from its local cache hierarchy in mid-transaction; if it does, the processor must stall for an overflow. Set modified bits will cause similar overflow conditions if the write buffer holds only addresses.

## TCC software

TCC parallelization requires only a few new programming constructs. It is simpler than parallelization with conventional threaded models because it needs fewer code transformations for typical parallelization efforts. In particular, it lets programmers make informed trade-offs between programmer effort and performance. In simplified form, programming with TCC is a three-step process:

1. *Divide the program into transactions.* To create a parallel program using TCC, a programmer coarsely divides the program into transactions that can run concurrently on different processors. In this respect, parallelizing for TCC is like conventional parallelization, which also requires finding and marking parallel code regions. However, with TCC the programmer does not need to *guarantee* that parallel regions are independent, as hardware will catch all dependence violations dynamically. Our interface lets programmers divide their program into parallel transactions using loop iterations and/or a forking mechanism.
2. *Specify transaction order.* The default transaction ordering is to have transactions commit results in the same order as they would in the original sequential program, because this guarantees that the program will execute correctly. However, if a programmer can verify that this commit order constraint is unnecessary, he or she can relax it completely or partially to improve performance. The interface also provides ways to specify the application's ordering of constraints in useful ways.
3. *Tune performance.* After selecting and ordering transactions, the programmer can run the program in parallel. The TCC system can automatically provide feedback about where violations occur in the program, which can direct the programmer to perform further optimizations.

We describe the interface in C, but it can be readily adapted to any programming language (for example, we also used Java).

### Loop-based parallelization

We introduce loop parallelization in the context of a simple sequential code segment that calculates a histogram of 1,000 integer

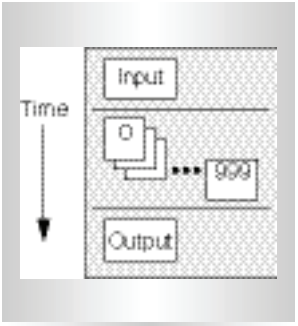


Figure 2. Sample schedule for transactional loops.

percentages using an array of corresponding buckets:

```
/* input */
int *in = load_data();
int i, buckets[101];

for (i = 0; i < 1000; i++) {
  buckets[data[i]]++;
}

/* output */
print_buckets(buckets);
```

The compiler interprets this program as one large transaction, exposing no parallelism to the TCC hardware. We can parallelize the **for** loop, however, with the **t_for** keyword:

```
...
t_for (i = 0; i < 1000; i++) {
...
```

With this small change, we now have a parallel loop that is guaranteed to execute identically to the original sequential loop. A similar **t_while** keyword is also available for **while** loops. Each loop body iteration becomes a separate transaction that can execute in parallel but must commit in the original sequential order, such as the pattern in Figure 2. When two parallel iterations try to update the same histogram bucket simultaneously, the TCC hardware causes the later iteration to violate when the earlier iteration commits, forcing the later one to reexecute using updated data and preserving the original sequential semantics.

In contrast, a conventionally parallelized system would require an array of locks to protect the histogram bins, resulting in much more extensive changes:

```
int* in = load_data();
int i, buckets[101];

/* Define & initialize locks */
LOCK_TYPE bucketLock[101];
for (i = 0; i < 101; i++) {
  LOCK_INIT(bucketLock[i]);
}
for (i = 0; i < 1000; i++) {
  LOCK(bucketLock[data[i]]);
  buckets[data[i]] ++;
  UNLOCK(bucketLock[data[i]]);
}
print_buckets(buckets);
```

If any of this locking code is omitted or buggy, the program *might* fail—and not necessarily in the same place every time—significantly complicating debugging. Debugging is especially difficult if the errors occur only during infrequent memory access patterns. The situation is potentially even trickier if an application must hold multiple locks simultaneously within a critical region, because programmers can easily write code with locking sequences that might deadlock in these cases.

Although sequential ordering is generally useful because it guarantees correct execution, in some cases—such as this histogram example—it is not actually required for correctness. In this example, the only dependencies among the loop transactions are through the histogram bin updates, which are performable in any order. If programmers can verify that no dependencies are order-critical, or if there are simply no loop-carried dependencies, they can use the **t_for_unordered** and **t_while_unordered** keywords to allow the loop's transactions to commit in any order. Allowing unordered commits is most useful in more complex programs with dynamically varying transaction lengths, because it eliminates much of the time that processors spend waiting for commit permission between unbalanced transactions.

## Fork-based parallelization

Although the simple parallel loop API will work for many programs, some less-structured programs might need to generate transactions more flexibly. For these situations **t_fork**, a transactional fork similar to conventional thread-creation APIs, is useful:

```
void t_fork(
  void(*child_function_ptr)
        (void*),
  void *input_data,
  int child_sequence_num,
  int parent_phase_increment,
  int child_phase_increment);

/* Which forks this off: */
```

```
void child_function
     (void *input_data);
```

This call forces the parent transaction to commit, and then creates two completely new—and parallel—transactions in its place. One (the new parent) continues executing the code immediately following the **t_fork**, while the other (the child) starts executing the function at **child_function_ptr** with **input_data**. Other input parameters control ordering of forked transactions in relation to other transactions, as we discuss in the following section. We demonstrate this function with a parallelized form of a simple two-stage processor pipeline, which we simulate using the functions **i_fetch** for instruction fetch, **increment_PC** to select the next instruction, and **execute** to execute instructions. The child transaction executes each instruction while the new parent transaction fetches another:

```
/* Initial setup */
int PC = INITIAL_PC;
int opcode = i_fetch(PC);

/* Main loop */
while (opcode ! = END_CODE)
{
  t_fork(execute, &opcode,
         1, 1, 1);
  increment_PC(opcode, &PC);
  opcode = i_fetch(PC);
}
```

This example creates a sequence of overlapping transactions similar to those in Figure 3. The **t_fork** call gives enough flexibility to divide a program into transactions in virtually any way. It can even be used to build the **t_for** and **t_while** constructs, if necessary.

### Explicit transaction commit ordering

The simple ordered and unordered modes might not always suffice. For example, a programmer might desire partial ordering—executing unordered most of the time, but occasionally imposing some ordering. This is rare with transactional loops, but quite common with forked transactions. To handle these cases, we have developed a simple model for explicitly controlling transaction sequencing when necessary. However, because sequencing requirements often vary in small but critical ways from program to program, this is still an active area of development as we evaluate more applications with TCC.

We control transaction ordering by assigning two parameters to each transaction: *sequence* and *phase*. These two numbers control the ordering of transaction commits. Transactions with the same sequence number might need to commit in a programmer-defined order, whereas transactions in different sequences are always independent. We can use the **child_sequence_num** parameter within a **t_fork** call to produce a child transaction with a new sequence number. Within each sequence, the phase number indicates each transaction's relative age. TCC hardware will only commit transactions in the oldest active phase (lowest value) from within each active sequence. Using this notation, an ordered loop is simply a sequence of transactions with the phase number incremented by one each time, whereas an unordered loop uses transactions with the same phase number.

We can impose more arbitrary transaction phase ordering with the following calls:

```
void t_commit(
  int phase_increment);
void t_wait_for_sequence(
  int phase_increment,
  int wait_for_sequence_num);
```

The **t_commit** routine implicitly commits the current transaction and then immediately starts another on the same processor with its phase number incremented by the **phase_increment** parameter. The most common **phase_increment** parameter is 0, which simply breaks a large transaction into two. However, using a **phase_increment** of 1 or more forces an explicit transaction commit ordering. This can be used in many ways, but the most common is to emulate a conventional barrier among all transactions in a sequence using transactional semantics. The similar **t_wait_for_sequence** call performs a **t_commit** and waits for all transactions in another sequence to complete. Programmers typically use this call to let a parent transaction sequence wait for a child sequence to complete, similar to a conventional thread join operation.

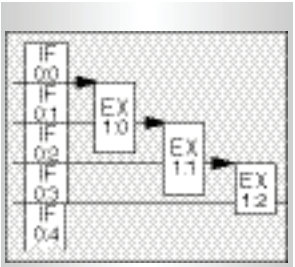


Figure 3. Simple fork example, where the parameters are sequence:phase.

**Table 1. Key parameters in our simulations (all cycle values are in CPU cycles).**

| System | Description | InterCPU bandwidth (bytes per cycle) | Commit overhead (cycles) | Violation delay (cycles) |
|---|---|---|---|---|
| Ideal | "Perfect" TCC multiprocessor | | 0 | 0 |
| Chip multiprocessor (CMP) | Realistic multiprocessor on a single chip | 16 | 5 | 0 |
| Symmetric multiprocessor (SMP) | Realistic multiprocessor on a board | 4 | 25 | 20 |

**Table 2. Characteristics of applications used for our analysis.**

| Source language | Benchmark | Application description | Source | Input | Lines of code | Lines changed (percent) | Primary TCC parallelization |
|---|---|---|---|---|---|---|---|
| Java | Assignment | Resource allocation solver | jBYTEmark | 51 × 51 array | 556 | 5.8 | Loop: two ordered, nine unordered |
| | MolDyn | N-body code-modeling particles | Java Grande | 2,048 particles | 615 | 3.3 | Loop: nine unordered |
| | LUFactor | Matrix factorization and triangular solver | jBYTEmark | 101 × 101 matrix | 516 | 1.9 | Loop: two ordered, four unordered |
| | RayTrace | 3D ray tracer | Java Grande | 150 × 150 pixel image | 1,233 | 4.9 | Loop: nine unordered |
| | SPECjbb | Transaction processing server | SPECjbb | 230 iterations without randomization | 27,249 | 1.3 | Fork: five calls (one per task type) |
| C | art | Image recognition/ neural network | SPEC2000 FP | Reference.1 | 1,270 | 8.9 | Loop: 11 chunked and unordered |
| | equake | Seismic wave propagation simulation | SPEC2000 FP | Reference | 1,513 | 0.8 | Loop: three unordered |
| | tomcatv | Vectorized mesh generation | SPEC95 FP | 256 × 256 array | 346 | 2.0 | Loop: seven unordered |
| | MPEGdecode | Video bitstream decoding | Mediabench | mei16v2.m2v | 9,834 | 4.6 | Fork: one call |

## Performance evaluation

We used our TCC programming constructs to parallelize applications from various domains. We then used simulation to evaluate and tune their performance for large- and small-scale TCC systems.

Our execution-driven simulator models a processor executing at a fixed rate of one instruction per cycle while producing traces of all transactions in the program. We analyzed the traces on a parameterized TCC system simulator that includes 4 to 32 processors connected by a broadcast network. The TLP-oriented benefits of TCC parallelization are fairly orthogonal to speedup from superscalar ILP extraction techniques within individual processors, so our results can scale for systems with processors faster (or slower) than 1.0 instructions per cycle, until the commit broadcast bandwidth is saturated.

Table 1 lists the values of three key system parameters that describe three potential TCC configurations: *ideal* (infinite bandwidth, zero overhead), *single-chip/CMP* (high bandwidth, low overhead), and *single-board/SMP* (medium bandwidth, higher overhead).

Table 2 presents the applications we used in the study. These applications exhibit a diverse set of concurrency patterns, including dense loops (LUFactor), sparse loops (equake), and task parallelism (SPECjbb). We modified the C applications to use transactions manually. In

contrast, we parallelized most Java applications automatically using the Jrpm dynamic compilation system,[4] and then ran them on top of a version of the Kaffe Java virtual machine (http://kaffe.org) customized to use transactions instead of locks. The only exception was SPECjbb, which we manually parallelized by forking transactions for each warehouse task (such as orders and payments). Although most programmers assign separate warehouses to each processor to parallelize this benchmark, TCC let us parallelize *within* a single warehouse, a virtually impossible task with conventional techniques. In all cases, we only needed to modify a modest percentage of the original code to make it parallel.

### Parallel performance tuning

After a program is divided into transactions, most problems will tend to be with performance, not correctness. To obtain good performance, programmers must optimize their transactions to balance a few competing goals:

- *Maximize parallelism.* Programmers must break as much of the program as possible into parallel transactions, which should be of roughly equal size to maintain good load balancing across processors.
- *Minimize violations.* To avoid the costly discarding of work after violations, programmers should avoid parallel transactions that communicate frequently. Keeping transactions reasonably small to minimize the amount of work lost when violations occur can also help.
- *Minimize transaction overhead.* On the other hand, programmers should generally avoid very small transactions because of the overhead associated with starting, ending, and committing transactions.
- *Avoid buffer overflows.* Buffer overflows can cause serialization by requiring processors to hold commit permission for a long time. Hence, programmers should stay away from transactions that read or write large amounts of state.

Figure 4a shows application speedups as we applied successive optimizations for CMP configurations with 4, 8, 16, and 32 processors, and Figure 4b breaks down execution time for the eight-processor case into "useful work," "waiting for commit," "losses to violations," and "idling" caused by too few parallel transactions for the available processors—usually caused by sequential code regions. Although the baseline results showed significant speedup, they often needed improvement. Where possible, we first optimized with unordered loops. However, because load balancing was rarely a problem in the applications we selected, as our low "waiting to commit" times show, this did not significantly improve performance.

We used results from initial application runs to guide our optimization efforts. These runs produced violation reports that summarized the load-store pairs and data addresses causing violations, prioritized by the amount of time lost. We then used `gdb` to determine *exactly* which variables and variable accesses caused the violations. This information is significantly more useful than feedback available on current shared-memory systems, which tends to be mostly in terms of coherence protocol statistics. This feature can greatly increase programmer productivity by guiding them *directly* to the most critical communication. Violation reports can lead to a wide variety of optimizations, many adapted from traditional parallelization techniques.

One of the most common optimizations is the privatization of shared variables that cause unnecessary violations. For example, we improved SPECjbb by privatizing some shared temporary buffers. The most common type of privatization, however, was to privatize the loop-carried "sum" variable used as a reduction target within loops that are otherwise good targets for parallelization. In our statistics, the "sum" variables caused frequent violations. However, many of the reduction operations are associative, such as addition and multiplication, and can therefore be reordered safely. Programmers can expose parallelism by privatizing a "sum" variable within each processor and then combining the private copies only at loop termination.

Although transactions should usually be monolithic, it is sometimes more helpful to break large transactions into a transactional group of two or more smaller transactions which must execute sequentially on one processor. A programmer can use a `t_commit(0)` call to mark each breakpoint between the individual transactions. Because

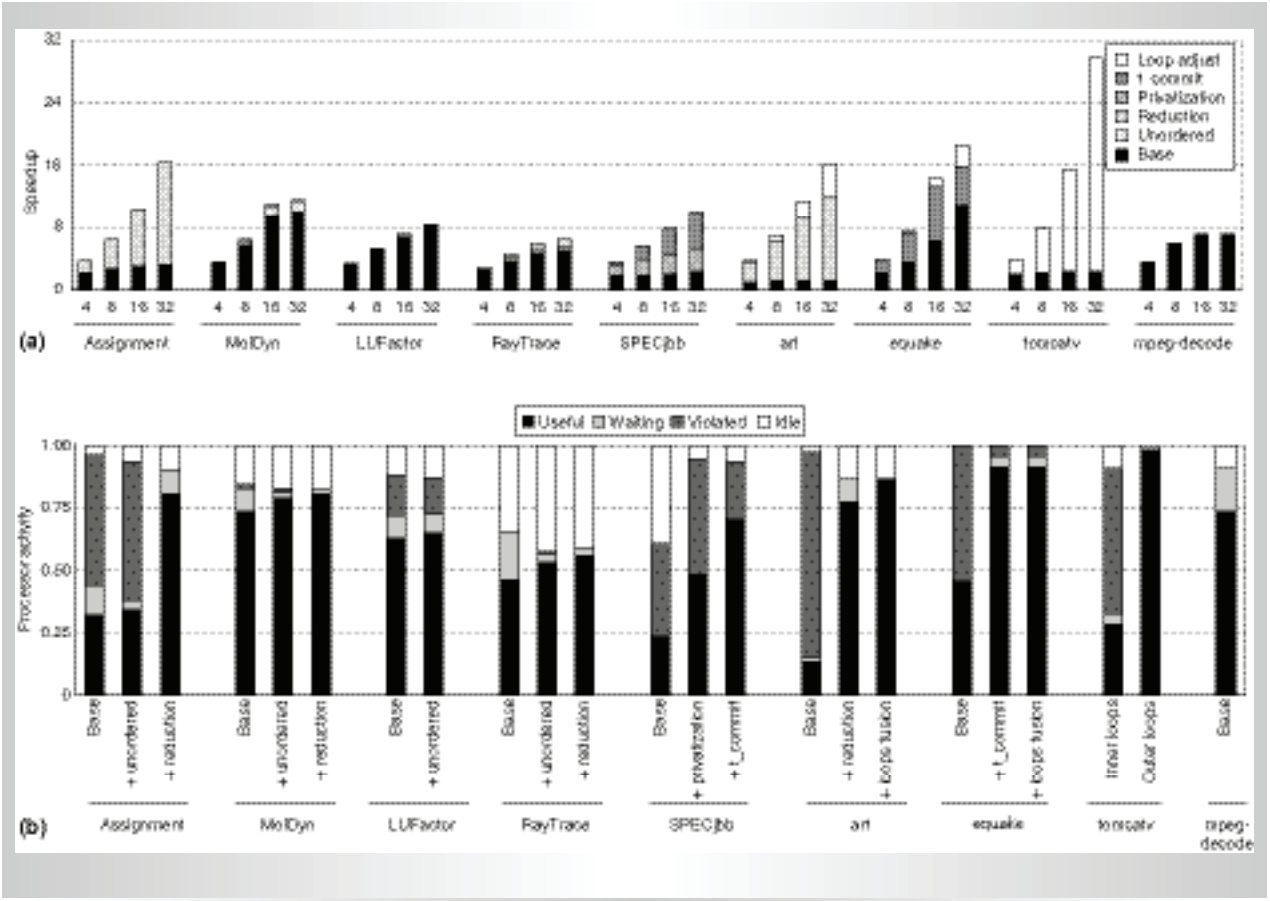


Figure 4. Evaluation results: improvements in CMP configuration speedup provided by the various optimizations for four to 32 processors (a), and processor utilization breakdowns for the eight-processor case (b).

this operation splits the original transaction into two separate atomic regions, the programmer must not place `t_commits` in the middle of critical regions that *must* execute atomically. Despite this limitation, the technique can help solve three problems:

- Inserting a `t_commit` takes a new rollback checkpoint, limiting the amount of work lost to a violation. This is helpful with SPECjbb, which has large transactions and frequent, unavoidable violations.
- It lets the processor commit and broadcast changes made by a transaction to other parallel processors early, when the original transaction would only have been partially completed.
- Because each commit flushes the processor's TCC write buffer, judicious `t_commits` can prevent or reduce buffer overflows.

Loop adjustment techniques can also be critical when optimizing code for TCC. *Loop unrolling*, *fusion*, *fission*, and *renesting* are common parallelizing compiler tricks that can all prove helpful. Although the techniques are the same, the patterns used typically differ somewhat, with optimal transaction sizes being the usual goal. In addition, with loop nests a programmer must carefully choose the proper loop nesting level to use as the source of transactions, because our model does not support nested transactions. Outer loop bodies provide large, coarse-grained transactions, but these loop bodies can easily be too large, causing frequent buffer overflows. On the other hand, using inner loop bodies can make the transactions too small, incurring excessive startup and commit overheads. Meanwhile, critical loop-carried dependencies can occur at any level. For example, tomcatv's inner loops had many dependencies and were small, forcing us to use outer loops.

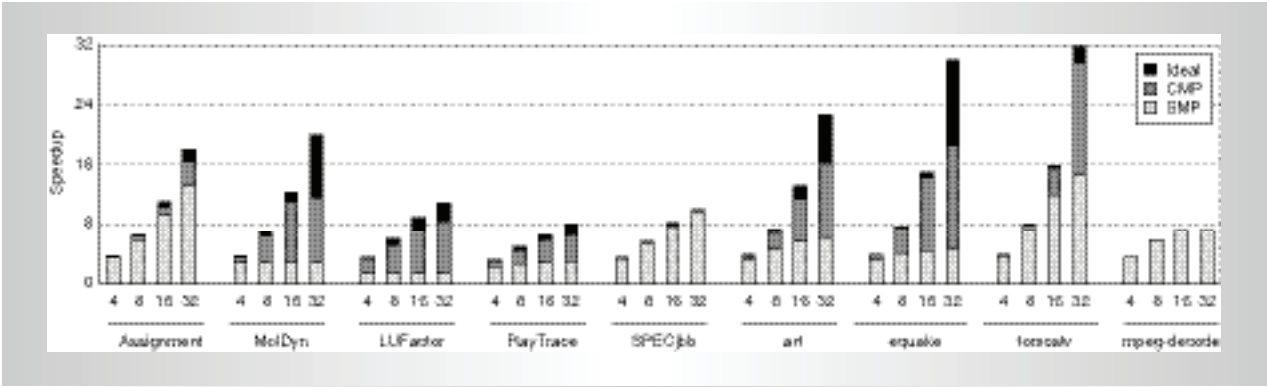


Figure 5. Overall speedup obtained using different hardware configurations.

### Overall results

Figure 5 presents the best achieved speedups for three TCC system configurations (ideal, CMP, and SMP) using 4 to 32 processors. While results are generally quite good, combinations of unavoidable violations and regions with insufficient parallel transactions, causing extra processors to idle, limit some applications. In addition, large sequential code regions limit Assignment and RayTrace.

For most benchmarks, CMP performance closely tracks ideal performance for all processor counts. The CMP configuration is worse than the ideal only when insufficient commit bandwidth is available, which generally occurs only with 32 or more processors. This causes processors to stall while waiting for access to the commit network. Similarly, the SMP TCC configuration achieves very little benefit beyond four- to eight-processor configurations because of its significantly reduced global bandwidth. Of our applications, only Assignment, SPECjbb, and tomcatv used more processors to significant advantage. This result is still promising, however, because online server applications such as SPECjbb would most likely be the primary applications running on TCC systems larger than CMPs.

The most significant TCC hardware is the addition of speculative buffer support. The amount of state read and written by an average transaction must be small enough for local buffering. Figure 6 shows the buffer size needed to hold the state read (Figure 6a) or written (Figure 6b) by 10, 50, and 90 percent of each application's transactions, sorted by the 90-percent limit's size, generally from 6 to 12 Kbytes for read state and 4 to 8 Kbytes for write state. All applications have a few very large transactions that will overflow, but TCC hardware should be able to achieve good performance as long as this does not occur often.

In the future, we plan to explore software and hardware techniques for scaling TCC to large-scale parallel systems, so that programmers can use a single programming paradigm on any parallel machines, from small CMPs to large SMPs. We will complement this work by investigating dynamic optimization techniques that will help automate the otherwise time-consuming parallel performance tuning of larger and more complex TCC programs. Ultimately, we see the combination of TCC hardware and highly automated TCC programming language support as forming a new programming paradigm that will provide a smooth transition from today's sequential programs to tomorrow's parallel software. This paradigm will be an important catalyst in transforming parallel processing from a niche activity to one that is accessible to the average software developer. MICRO

## Acknowledgments

This work was supported by US National Science Foundation grant CCR-0220138 and DARPA PCA program grants F29601-01-2-0085 and F29601-03-2-0117.

## References

1. R. Rajwar and J. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*

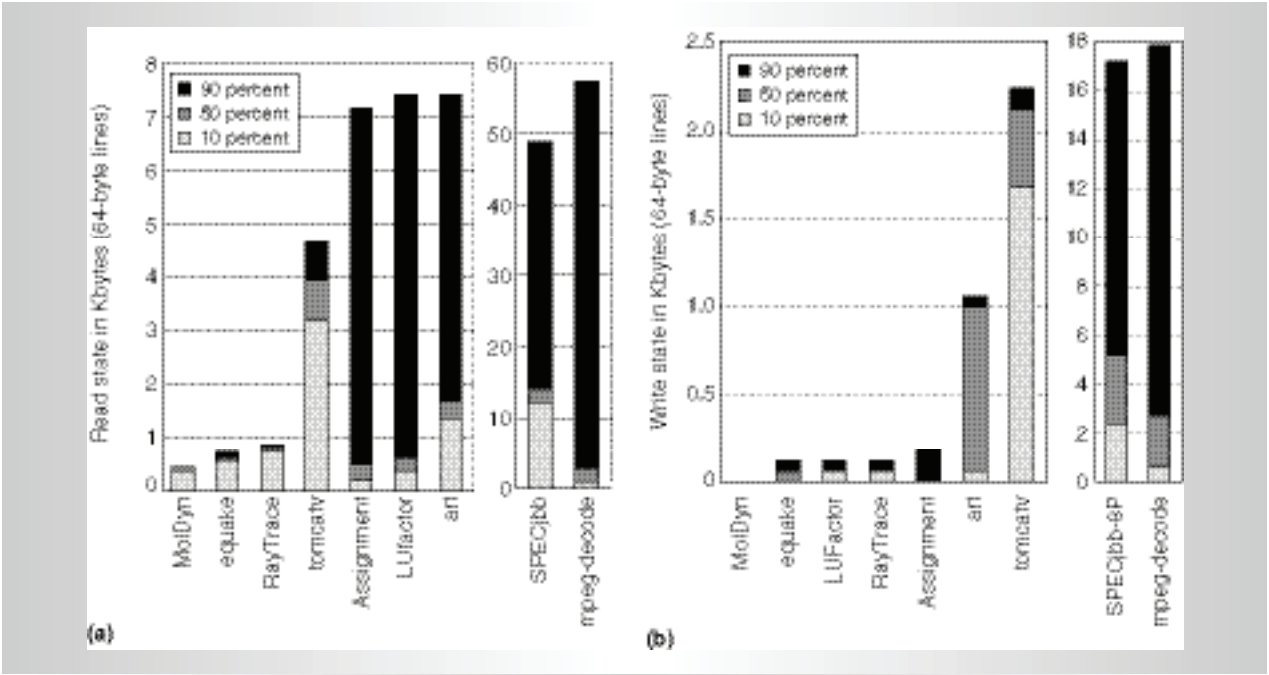


Figure 6. State read (a) or written (b) by individual transactions, with a cache/buffer granularity of 64-byte lines, for 10, 50, and 90 percent of the iterations.

(ASPLOS 02), ACM Press, 2002, pp. 5-17.

2. M. Herlihy and J. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. 20th Int'l Symp. Computer Architecture* (ISCA 93), ACM Press, 1993, pp. 289-300.
3. J. Martinez and J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Parallel Applications," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 18-29.
4. M.K. Chen and K. Olukotun, "The Jrpm System for Dynamically Parallelizing Java Programs," *Proc. 30th Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 434-445.
5. L. Hammond et al., "Programming with Transactional Coherence and Consistency (TCC)," *Proc. 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 04), ACM Press, 2004, pp. 1-13.
6. L. Hammond et al., "Transactional Memory Coherence and Consistency," *Proc. 31st Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 102-113.

**Brian D. Carlstrom** is a PhD candidate in computer science at Stanford University. His research interests include transactional programming models and other aspects of modern programming language design and implementation. Carlstrom has a master's degree in electrical engineering and computer science from the Massachusetts Institute of Technology. He is a member of the ACM.

**Vicky Wong** is a PhD candidate in computer science at Stanford University. Her research interests include parallel computer architectures and probabilistic reasoning algorithms. Wong has an MS in computer science from Stanford University.

**Michael Chen** performed this work while completing his PhD in electrical engineering at Stanford University. He is now a staff researcher in the Programming System Lab at Intel's Microprocessor Research Labs. His research interests include optimization techniques for compiling high-level languages to the Intel IXP network processors.

**Christos Kozyrakis** is an assistant professor of electrical engineering and computer science

at Stanford University. His research interests include architectures, compilers, and programming models for parallel systems ranging from embedded devices to supercomputer. Kozyrakis has a PhD in computer science from the University of California, Berkeley.

**Kunle Olukotun** is an associate professor of electrical engineering and computer science at Stanford University. His research interests include computer architecture, parallel programming environments, and formal hardware verification. Olukotun has a PhD in computer science and engineering from the University of Michigan.

Direct questions and comments about this article to Kunle Olukotun, Stanford University, Gates Bldg. 3A-302, Stanford, CA, 94305-9030; kunle@stanford.edu.