Programming with Transactional Coherence and Consistency (TCC)

Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen,
Christos Kozyrakis, and Kunle Olukotun

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

{lance, bdc, vicwong, elektrik, broccoli, kozyraki, kunle}@stanford.edu

ABSTRACT

Transactional Coherence and Consistency (TCC) offers a way to simplify parallel programming by executing all code within transactions. In TCC systems, transactions serve as the fundamental unit of parallel work, communication and coherence. As each transaction completes, it writes all of its newly produced state to shared memory atomically, while restarting other processors that have speculatively read stale data. With this mechanism, a TCC-based system automatically handles data synchronization correctly, without programmer intervention. To gain the benefits of TCC, programs must be decomposed into transactions. We describe two basic programming language constructs for decomposing programs into transactions, a loop conversion syntax and a general transaction-forking mechanism. With these constructs, writing correct parallel programs requires only small, incremental changes to correct sequential programs. The performance of these programs may then easily be optimized, based on feedback from real program execution, using a few simple techniques.

Categories and Subject Descriptors

C.5.0 **[Computer System Implementation]**: General.

D.1.3 **[Programming Techniques]**: Concurrent Programming — *parallel programming*.

General Terms

Performance, Design, Languages.

Keywords

Transactions, feedback optimization, multiprocessor architecture.

1. INTRODUCTION

With uniprocessor systems running into instruction-level parallelism (ILP) limits and fundamental VLSI constraints [2], parallel architectures provide a realistic path towards scalable performance by allowing one to take advantage of thread-level parallelism (TLP) in more explicitly distributed architectures. Single-board and single-chip multiprocessors are becoming the norm for server [18, 31] and embedded [5] computing, and are starting to appear even on desktop platforms. Multiprocessor systems provide a good match to the coarse-grain parallelism available in applications such as enterprise services, bio-computing, telecommunications, and multimedia. Nevertheless, the key factor limiting the potential of parallel architectures is the complexity of parallel application development.

Existing parallel programming approaches require the programmer to manage concurrency directly by creating and synchronizing parallel threads. The difficulty stems from the need to achieve the often conflicting goals of functional correctness and high performance. With shared memory systems [25], a small number of coarse-grain locks makes it simpler to correctly sequence accesses to variables shared among parallel threads. On the other hand, more numerous fine-grain locks often allow 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 used usually incurs more locking overhead. A similar trade-off exists with message-passing programming [10]. High performance requires early scheduling of all communication events, while correct execution requires a programmer to carefully match send and receive requests across threads, even for applications with dynamic and unpredictable communication patterns. Managing this trade-off makes parallel programming more time-consuming and error-prone than writing an equivalent sequential program.

This paper introduces parallel programming techniques for transactional coherence and consistency (TCC) systems [12]. TCC relies on programmer-defined transactions as the basic unit of parallel work, communication, memory coherence, memory consistency, and error recovery. TCC hardware speculatively executes transactions in parallel using local buffering. After a transaction completes, the hardware commits all its writes to shared memory as an atomic unit. At this point, the writes become visible to other transactions, which may rollback due to dependency violations. TCC simplifies parallel hardware design by eliminating the need for cache line ownership tracking in the cache coherence protocol. It also replaces the need for numerous small, low latency messages for cache coherence with fewer large, high-bandwidth messages for atomic commit.

TCC simplifies parallel programming by eliminating the need for manual orchestration of parallelism using locks or messages. Programmers simply need to divide computation into potentially parallel transactions and then specify any ordering dependencies that must be observed between those transactions' commits. TCC hardware guarantees correct synchronization always occurs by automatically restarting transactions on dependency violations. Therefore, decomposing code into transactions is primarily a matter of

ASPLOS'04, October 7–13, 2004, Boston, Massachusetts, USA.

performance tuning, and not a matter of correctness. The use of a single abstraction for parallelism, communication, and synchronization also simplifies performance tuning by allowing programmers to use simple transaction statistics (violations, buffer requirements, overheads) to identify and remove performance bottlenecks.

To our knowledge, this work is the first proposal for parallel programming using transactions as the central programmer abstraction for both parallelism *and* synchronization in a shared memory system, completely eliminating the need for locks and allowing for much more automated sequencing of parallel code regions. The specific contributions of this paper are:

- *Transactional programming constructs:* We identify two methods for defining transactions and specifying commit orders. The first one is appropriate for loop-based code, while the second one resembles general thread forking. The constructs can express ordered, partially ordered, and unordered commit successions. We show that the two constructs can express the parallelism in a diverse set of C and Java applications, ranging from array computations to an online server benchmark.
- *Transactional performance tuning:* We show how the programmer can tune application performance using feedback on transaction behavior (violations, buffering requirements, overheads) obtained during initial, unoptimized runs of the program. We propose a set of simple source code optimizations that can lead to additional performance improvements without affecting application correctness.
- *High application performance:* We use simulation to demonstrate that, despite its simplicity, parallel programming with TCC allows excellent speedups for chip multiprocessor (CMP) across a range of 4–32 processors and board-level symmetric multiprocessor (SMP) systems for 4–8.

The rest of the paper is organized as follows. Section 2 provides an overview of the operation of a TCC-based parallel system. Section 3 introduces TCC programming techniques, and discusses correctness and performance tuning. In Section 4, we demonstrate the use of the programming techniques in parallelizing a diverse set of applications. Section 5 discusses related work and we conclude in Section 6.

2. TCC HARDWARE OVERVIEW

Processors operating in a TCC-based multiprocessor continually execute speculative transactions, using a cycle illustrated in Figure 1a on multiprocessor hardware with additions similar to those 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 produces a block of writes which are buffered locally while the transaction executes and are then committed to shared memory only as an atomic unit, after the transaction completes. Once the transaction is complete, hardware must arbitrate system-wide for the permission to commit its writes.

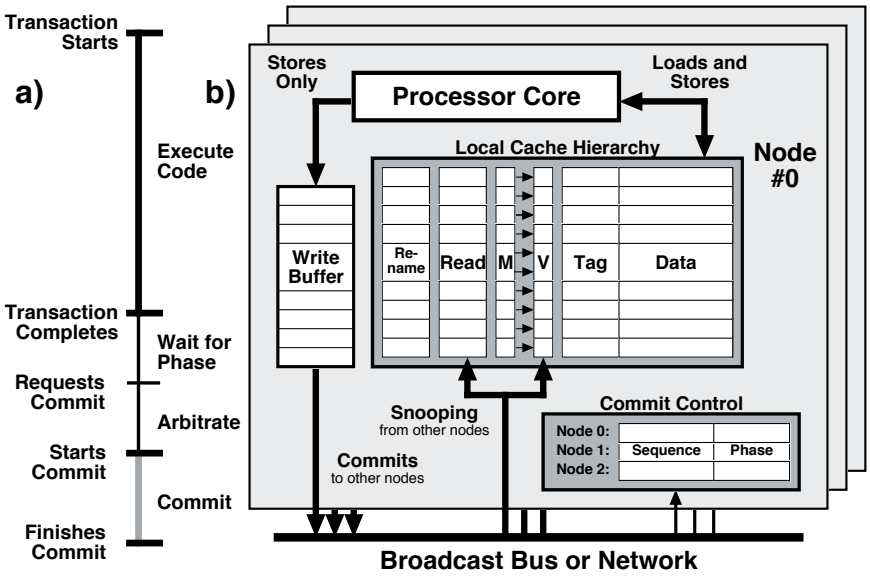

Figure 1: a) A transaction cycle (time flows downwards) and b) a diagram of sample TCC-enabled hardware.

After this permission is granted, the processor can take advantage of high-bandwidth system interconnect to broadcast all writes for the entire transaction out as one large packet to the rest of the system. Meanwhile, the local caches in other processors snoop on these store packets to maintain coherence in the system. Snooping also allows them to detect when they have used data that has subsequently been modified by another processor — a dependence violation. Combining all writes from the entire transaction together minimizes the latency sensitivity of this scheme, because fewer interprocessor messages and arbitrations are required, and because flushing out the writes is a one-way operation. At the same time, the commit operation can also be leveraged to provide inherent synchronization and a greatly simplified consistency protocol, since we have to control only the ordering between entire transactions instead of individual loads and stores.

This continual cycle of speculative buffering, broadcast, and (potential) violations, described in further detail in [12], allows us to replace both conventional coherence and consistence protocols:

- **Consistence:** Instead of using rules that control ordering between individual memory reference instructions, as with most coherence schemes, TCC just controls ordering between transaction commits. This can drastically reduce the number of latency-sensitive arbitration and synchronization events required by low-level protocols in a typical multiprocessor system. 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 to the sequence of memory references generated by software. As far as the global memory and software is concerned, *all* memory references from a transaction that commits earlier happened "before" *all* of the memory references of a transaction that commits afterwards, even if their actual execution was interleaved in time, because all writes from a transaction become visible to other processors *only* at commit time, all at once.

- **Coherence:** Stores are buffered and kept within the processor node for the duration of the transaction in order to maintain the atomicity of the transaction. No conventional MESI-style cache protocols are used to maintain lines in "shared" or "exclusive" states at any point in the system, so it is legal for many processor nodes to hold the same line simultaneously in either an unmodified or speculatively modified form. At the end of each transaction, the broadcast notifies all other processors about what state has changed during the completing transaction. 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 if they may have used shared data too early. If they have read any data modified by the committing processor during their current transaction, they are forced to restart and update their copy of the data. This protects against true data dependencies. At the same time, data antidependencies are handled simply by the fact that later processors will eventually get their own turn to flush out data to memory. Until that point, their "later" results are not seen by transactions that commit earlier (avoiding WAR dependencies) and they are able to freely overwrite previously modified data in a clearly sequenced manner (handling WAW dependencies in a legal way). Effectively, the simple, sequentialized consistence model allows the coherence model to be greatly simplified.

Transactional hardware hides many of the difficulties associated with parallel programming from typical programmers, but in order to get good performance programmers do need to keep a few basic goals in mind when dividing applications into transactions:

- **Minimize Violations:** Programmers should try to avoid parallel transactions that read and write the same variables frequently, in order to avoid costly discarding of work that occurs after violations. Keeping transactions reasonably small, to minimize the amount of work lost when violations occur, can also help.

- **Minimize Transaction Overhead:** On the other hand, *very* small transactions should generally be avoided when possible because of the overhead associated with starting, ending, and committing transactions.

- **Avoid Buffer Overflows:** TCC hardware must buffer all writes made by a processor during a transaction. Our previous results in [12] indicate that most applications naturally divide into transactions of reasonable size, but when very large transactions occur it is possible to overflow the finite buffer space. While the system is always able to handle these situations *correctly* when they occur, simply by having the processor request access to the broadcast network and then hold it while writing through directly to memory for the remainder of the transaction, this can obviously have a negative impact on performance and should be avoided if possible.

Other than these few factors, which mostly affect performance tuning, programmers can generally ignore the hardware. This contrasts well against conventional parallel systems, where they would have to carefully consider issues such as the layout of data in the machine's memory, latency and bandwidth requirements for all communication, and other machine-specific factors that can have an impact on both correctness *and* performance. This allows them to focus more on writing correct code.

3. PROGRAMMING TECHNIQUES

TCC parallelization is a series of simple steps that requires only a few new programming constructs. This process is simpler than parallelization with conventional threaded models because it reduces the number of code transformations needed for typical parallelization efforts. In particular, it allows programmers to make *informed* tradeoffs between programmer effort and performance. Basic parallelization can quickly and easily be done in a way that is guaranteed to be safe. Programmers can then use feedback obtained from violation reports produced during initial parallel program execution to insert program refinements and constraint relaxation in order to get significantly greater speedups. In a simplified form, programming with TCC can be summarized as a three-step process:

- *Divide into Transactions:* The first step in the creation of a parallel program using TCC is to coarsely divide the program into blocks of code that can run concurrently on different processors. In this respect, parallelizing for TCC is very similar to conventional parallelization, which also requires that programmers find and mark parallel regions. However, the actual process is simpler with TCC because the programmer does not need to *guarantee* that parallel regions are independent, since the TCC hardware will catch all dependence violations during execution. The interface presented in this section allows programmers to divide their program into parallel blocks on loop iterations and by forking of transactions. Currently, our interface supports only "flat" transactions, and not "nested" ones [9].

- *Specify Order:* The default ordering for transactions is to have them commit results in the same order as the original sequential program, since this guarantees that the program will execute correctly. However, if a programmer is able to verify that this commit order constraint is unnecessary, then it can be relaxed completely or partially in order to provide better performance. The interface also provides ways to specify the ordering constraints of the application in useful ways.

- *Performance Tuning:* After transactions are selected and ordered, the program can be run in parallel. The TCC system can automatically provide informative feedback about where violations occur in the program, which can direct the programmer to perform further optimizations.

While the interface is described in C, it should be noted that these constructs can be readily adapted to any programming language (for example, we have also adapted it to Java) in order to allow it to take advantage of TCC's features.

3.1. Loop-Based Parallelization

The parallelization of loops will be introduced in the context of a simple sequential code segment that calculates a histogram of 1000 integer percentages using an array of corresponding buckets:

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

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

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

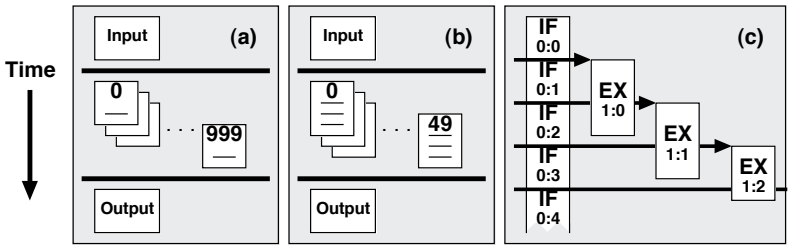

Figure 2: Illustrations of scheduling for transactional loops, chunked loops, and a simple fork example. The numbers in c's transactions are *sequence*:*phase*.

The compiler interprets this program as one transaction, so it exposes no parallelism to the underlying TCC hardware. The obvious candidate for parallelization is the `for` loop.

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

The only thing that changed is the `for` loop keyword, which has been replaced with its transactional version, `t_for`. With this small change, we now have a **parallel** loop that is *guaranteed* to execute the original **sequential** code *correctly*. Each iteration of the loop body will now become a separate transaction that will commit in the original sequential order, in a pattern like that in Figure 2a. This behavior preserves the original sequential order of the code by defining an ordering between the transaction commits. Although a later iteration may run in parallel with an earlier one, it *cannot* commit its transaction out of order. Therefore, if an earlier iteration updates a histogram bucket which is also updated by later iterations, when the earlier iteration commits, the TCC hardware will catch any dependence violations with data used by the "later" parallel iterations and restart them, forcing them to re-execute using updated data in order to preserve the original sequential semantics. For example, there may occasionally be collisions during access to the histogram bins in this program. TCC will handle these automatically, without any extra code.

The programmer ease-of-use of this paradigm compares favorably with similar schemes previously proposed. It is similar to the thread-level speculation (TLS) `pfor` construct [29], but does not allow forwarding of modified data between active speculative transactions. A conventionally parallelized system, however, would require an array of locks to protect the histogram bins, resulting in our example growing to something much uglier, with significant locking code, like this:

```
int* data = 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);
```

Unlike the TCC version, if any of this locking code is omitted or buggy, then the program *may* fail—and not necessarily in the same place every time—significantly complicating debugging. Debugging is especially hard if the errors only happen for patterns of memory accesses that occur only rarely, due to non-deterministic interprocessor timing. The situation is potentially even trickier if multiple locks need to be held simultaneously within a critical region, because one must be careful to avoid locking sequences that may deadlock [25]. It is clear that TCC can both simplify the parallelization of loops *and* reduce the runtime overhead associated with conventional locking and synchronization structures—since they largely disappear.

However, this simple application of TCC will not always result in good performance, so further modification may be desired. The first problem in this example stems from the size of the loop transactions. We now have 1002 transactions, one for the code before the loop, 1000 for the loop iterations, and 1 for the code after the loop. However, the loop body transactions are very small. To amortize the overheads associated with starting and ending transactions, `t_for` can be replaced with `t_for_n`, where the "`n`" allows the number of iterations to be included in a transaction to be specified by the programmer:

```
. . .
t_for_n (i = 0; i < 1000; i++; 20) {
. . .
```

Now we we have "chunked" 1000 transactions into 50 transactions, each executing 20 iterations of the original loop, for a total of 52 transactions, as is depicted in Figure 2b. This type of transformation is equivalent to unrolling the original loop and applying a `t_for` to the unrolled loop. A compiler or runtime environment should be able to perform this optimization automatically, since merging transactions while preserving ordering always maintains program correctness. By specifying the unrolling factor as a variable expression, the degree of "chunking" may even be set at run time, in response to characteristics of the input dataset.

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 case, there are no dependencies among the loop transactions except through additions to histogram elements, which is an associative operation that does not demand any particular ordering. When programmers can determine that this is the case, they can use slight variations on the basic `t_for` construct, `t_for_unordered` and `t_for_unordered_n`, to allow the main loop body transactions to run and commit in any order. While the transaction lengths are equal in this example, allowing unordered commits is most useful when the transaction lengths are dynamically variable, because it will decrease unnecessary time spent waiting for commit permission between transactions.

The test clause of any `t_for_unordered` has an additional requirement that it must be possible to determine the termination result in a distributed manner, so that all processors will be able to detect it, no matter what order they complete. For example, in a loop with an `i++ incrementor`, a test clause of "`i < 1000`" would be fine, since all processors will detect the end-of-loop condition after `i` reaches 1000, but "`i != 1000`" would result in only

a single processor detecting the termination condition because `i` would subsequently be incremented to 1001.

For loops with an indeterminate number of loop iterations, there is also a `t_while` loop structure, which is similar to a standard `while` loop, and comes in the same four flavors as `t_for`, with the same restrictions:

```
t_while<_mode> (i < 1000) {
        . . . is equivalent to . . .
t_for<_mode> (; i < 1000;) {
```

3.2. Fork-Based Parallelization

While most sequential programs can be easily divided into parallel transactions using only the simple parallel loop API, there are some less structured programs that need to generate transactions in a more flexible manner. Therefore, in addition to the loop parallelization, the TCC programming interface provides `t_fork`, a transactional fork similar to most conventional thread creation APIs. The interface is centered around a standard C function that starts a new, parallel transaction:

```
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 off a child function of the form: */
void child_function(void *input_data);
```

This call forces the "parent" transaction to commit, to guarantee that we cannot roll back and "cancel" the `t_fork`, and then creates two completely new, parallel transactions. One (the "new parent") continues execution of the code immediately following the `t_fork`, while the other (the "child") starts executing a `child_function`-style function at `child_function_ptr` with the input pointer `input_data`. Other input parameters control ordering of forked transactions in relation to other transactions, and are discussed in more detail below in Section 3.3.

To demonstrate this function, consider a simple example where we want to simulate a two stage processor pipeline in parallel. This is simulated using the functions `opcode = i_fetch(PC)` for instruction fetch, `increment_PC(opcode, PC)` to adjust the PC and handle branching, and `execute(opcode)` to execute other instructions. The "child" transaction executes each instruction while the "new parent" transaction goes ahead and fetches the next one:

```
/* Define an ID number for the EX sequence */
#define EX_SEQ  1

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

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

This example creates a sequence of overlapping transactions like those in Figure 2c. While it is more complicated to use than the looping structures, `t_fork` gives enough flexibility to divide a program into transactions in virtually any way. It can even be used to build the various `t_for` and `t_while` constructs, although those are useful so often that it is helpful to have the fundamental constructs described previously.

3.3. Explicit Transaction Commit Ordering

When using fork-based parallelization and some loop-based parallelizations, the simple "ordered" and "unordered" ordering modes may not be sufficient. For example, a programmer may desire a parallel loop that is only *partially* ordered, executing unordered iterations most of the time, but occasionally forcing one transaction or another to complete before others. In our experience with loop-based TCC programming, these ordering requirements are usually rare, but support must be provided for the occasional exception. On the other hand, forked transactions usually require explicit ordering of some sort.

In order to provide ordering support, we define a simple interface that controls transaction ordering by assigning two parameters to each transaction: the *sequence* and *phase* of transactions. These are two numbers assigned to each transaction that control the ordering of transaction commits. The *sequence* specifies which transactions require that commits must be ordered in relation to each other (i.e., have the same sequence ID number) and which are completely independent in their ordering (i.e., have a different sequence ID). The `child_sequence_num` parameter in a `t_fork` call can be used to produce two sequence groups of transactions that are ordered independently. When this is a positive integer, a "child" transaction with the new sequence number is produced, while the "new parent" transaction continues to use the old sequence number. Alternatively, the "child" and "new parent" may also be kept within the same ordering sequence using the `T_SAME_SEQUENCE` constant. Within each sequence, the *phase* indicates the relative "age" of each transaction. TCC hardware will only commit transactions in the oldest active phase (lowest value) from within any sequence that is executing on the system. Using this notation, an ordered loop is just a sequence of transactions with the phase incremented by one every time, while an unordered loop has transactions all with the same phase number.

Unlike the simpler loop constructs, a `t_fork` call allows complete control over the phase of the subsequent new parent and child transactions using the `parent_phase_increment` and `child_phase_increment` parameters, respectively. For the new parent transaction, this value is always a phase increment over the phase of the parent. For the child, this can either be a phase increment over the parent, if we use the `T_SAME_SEQUENCE` constant for selecting the sequence, or over the phase of the youngest active transaction already present within the child's sequence, if a sequence is explicitly supplied. For example, in the instruction simulator example from Section 3.2, we specified a phase increment of 1 for the "new parent" to ensure that the PC increments performed by the various "parent" transactions were all ordered properly within the parent sequence, and also specified an increment of 1 for the "children" so the various EX stages were required to commit their results to memory in the order they were forked within the separate `EX_SEQ` sequence.

More arbitrary phase ordering of transactions can also be imposed using the following two 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 a phase incremented by the `phase_increment` parameter. The most common `phase_increment` parameter used is 0, which is simply used to take a new checkpoint and to flush out the write buffers, at the programmer's request. However, it can also be used with a `phase_increment` of 1 or more in order to force an explicit transaction commit ordering. One use for this is to emulate a conventional barrier among all transactions within a sequence using transactional semantics. Unlike normal barriers, deadlock is *not* possible with this interface, as the "oldest" transaction in the sequence can *always* commit. Of course, these `t_commit`s can still cause "fast" processors to stall while waiting to commit the transaction *following* the `t_commit`, potentially reducing performance. This stall time is usually much less than that caused by conventional barriers, however, due to the fact that processors only stall if they get more than a full transaction ahead of their slower partners. In essence, these "transactional barriers" *automatically* act like the speculative barriers described in [26].

The `t_wait_for_sequence` call performs the same function as a `t_commit`, but also waits for all transactions within the `wait_for_sequence_num` sequence to complete before starting the next new transaction in the caller's sequence. This call is usually used to allow a "parent" sequence of transactions to wait for a "child" sequence to complete, similar to a thread join in conventional parallel programming. For example, the instruction simulator example from Section 3.2 requires the following code within the `increment_PC` routine to ensure that data-dependent branches are properly executed:

```
if (opcode == ANY_BRANCH_INSTRUCTION)
{
    t_wait_for_sequence(1, EX_SEQ);
    execute_branch(opcode, &PC);
}
else
    PC = PC + 1;
```

In this case, if the instruction is a branch, then the "parent" transaction sequence, which is handling the PC increments, is forced to wait until all previously initiated instruction simulations in the `EX_SEQ` sequence have completed, so that the branch instruction may only execute with the properly updated input values. For the common case of a non-branch instructions, however, the parent and child instruction sequences are allowed to commit transactions asynchronously, allowing us to take advantage of possible parallel speedup.

3.4. Performance Tuning

After a programmer divides a program into transactions with the fundamental TCC language constructs, most problems that occur will tend to be with *performance*, and not correctness. Since transactions automatically ensure that accesses to variables occur atomically, the *only* two ways that a programmer can write an incorrect TCC program are either by relaxing commit scheduling constraints too much or by accidentally putting a transaction break in the middle of a critical region. The former error can be avoided simply by always using explicitly ordered transactions if there is *any* doubt as to whether or not transactions can be safely executed in an unordered manner. Conventional parallel programming requires not only that one avoid breaking critical regions, but also that all critical regions be *explicitly marked* with locking code, a much more error-prone task than the TCC alternative.

Because it is very easy to get a program to at least run in parallel under TCC—although perhaps inefficiently—the usual way that TCC programs are optimized is to use actual execution to test initial parallelization attempts. Results from these initial runs can then provide useful feedback to aid further optimization through reports summarizing all violations that occur. These violation reports summarize time lost to transaction pairs causing conflicts, including information about the individual load-store pairs and data addresses that are violating. These addresses can be fed into a symbolic debugging environment like `gdb` to determine which variables and variable accesses caused the violations. This report can be summarized based upon the amount of execution time lost to each violation in order to prioritize and pinpoint the most important violation problems. This information tells programmers *exactly* which variables—and even which accesses to them—are limiting parallelism, unlike current shared memory systems, that tend to give feedback mostly in terms of coherence protocol statistics. This information can greatly increase programmer productivity by automatically directing programmers to the data dependencies that cause the most violations between transactions, instead of making them pore over all of the code in a program to determine the critical dependencies manually. Just as helpful, programmers may simply choose to *ignore* minor dependencies that only occur rarely, leaving these unimportant dependencies for the transactional hardware to handle with an occasional violation when they do occur. Information about load imbalance is also reported in terms of the time spent waiting at the end of each transaction defined by the source code.

Violation and waiting reports from initial program runs might lead only to some fine tuning of how certain variables are used, or it could lead to more major code changes. A common technique is to simply combine small transactions into larger ones (usually using the `_n` options). The program might also need some minor code restructuring to remove false sharing between transactions or code motion to allow better partitioning of transactions in order to avoid load imbalance or to reduce the amount of serial code between parallel regions, since a TCC system is still subject to Amdahl's law speedup limitations from such code. During the course of parallelizing several applications, we also found that several common techniques were very useful, many adapted from ones also used during traditional parallelization:

Reduction Privatization: Operations that reduce values to a loop-carried **sum** variable are frequently used within loops that are otherwise good targets for transactional parallelization. These operations show up in our statistics through frequent violations on the **sum** variable. However, many of the operations are associative, such as addition, multiplication, and minimum/maximum, and can therefore be reordered to maximize parallelism. By privatizing the **sum** variable within each processor and only combining these variables to a sequential sum after the end of the loop, significant paral-

lelism can be exposed. This is a process that can often be automated in floating-point applications. The low-level runtime environment needs to provide only a pair of functions that return system parameters (`int t_processor_count()` and `int t_processor_id()`) to support this functionality.

Shared Buffer Privatization: While our TCC compilation and hardware could automatically privatize most stack variables, some loops use large temporary buffers elsewhere in memory. In these cases, it may be necessary to allocate N buffers for N parallel processors in order to avoid spurious violations when unrelated variables are allocated at the same addresses within a shared buffer. The same techniques used to generate privatized **sum** variables for reductions can also be used to provide private temporary buffers. Hardware, compiler, or runtime environment support to automate privatization is a topic for further research. Without this support, it was necessary to manually privatize some buffers in our selection of applications.

Loop Level Adjustment: We may choose to parallelize at different levels of loop nests in order to take advantage of the characteristics that different levels may offer. Outer loops provide large granularity, but can sometimes get too large, because realistic TCC systems have finite amounts of per-transaction buffering. Inner loops can often be too small to effectively use without combining iterations, due to startup/commit overheads. Either may have critical loop-carried dependencies that prevent it from being a good target.

Loop Fusion/Fission/Renesting: In order to help make better transactions, any of these common parallelizing compiler tricks that adjust the execution pattern of loops may prove helpful. While the techniques are the same, the patterns used are usually somewhat different, with "optimal" transaction sizes being the usual goal.

Splitting Transactions into Transactional Groups: Normally, transactions do not commit results until after they complete entirely. While this is often desirable behavior, there are times when it is more helpful to break "obvious" transactions, such as loop iterations, into two or more parts. This can be performed simply by using a `t_commit(0)` call to break the transaction into two smaller transactions. We call the resulting transactions, that are forced to execute one after another on the same processor, a "transactional group." Because other, parallel transactions may commit results between the hardware transactions within a transactional group, the programmer must ensure that no `t_commit`s are placed in the middle of critical regions of the code that *must* execute atomically, or incorrect execution may result. Despite this limitation, we found that this technique was quite helpful in solving three different kinds of problems. First, inserting a `t_commit` takes a new rollback checkpoint, limiting the amount of work that can be lost if a violation occurs. This can be very important for applications with long "obvious" transactions and frequent violations. Second, flushing out the write state clears the TCC write buffer associated with the processor. If an application's transactions tend to produce a lot of data, then judicious `t_commit` operations can prevent the system serialization and slowdown that would otherwise occur when write buffers overflow. Finally, it may be desirable to update changes to global memory as early as possible, and a `t_commit` can flush out new changes to the globally visible state at any point within a transactional group, providing a sort of "forwarding" of modified data to other transactions running in parallel.

Table 1: Key parameters of our simulations. All cycle values are in CPU cycles.

System	Description	Inter-CPU Bandwidth (bytes/cycle)	Commit Overhead (cycles)	Violation Delay (cycles)
Ideal	"Perfect" TCC multiprocessor	∞	0	0
CMP	Realistic multiprocessor, if on a single chip	16	5	0
SMP	Realistic multiprocessor, if on a board	4	25	20

4. PERFORMANCE EVALUATION

In order to evaluate the utility of the transactional programming constructs, we used them to parallelize C and Java applications from a variety of domains. Then, we used simulation to evaluate and tune their performance for large and small-scale TCC systems.

4.1. Methodology

Our evaluation infrastructure includes an execution-driven simulator that models a processor that executes at a fixed rate of one instruction per cycle. The simulator produces an execution trace for all transactions in the program, including ones from sequential code regions, and captures statistics for all loads and stores except for stack references, which are guaranteed to be private within each transaction. We then use a trace analyzer to simulate the behavior of running transactions in parallel on a parameterized TCC system that includes multiple processors (4–32) connected through a network that supports broadcasts (e.g., a bus). We do not model further details of the core processor pipeline in this study because the TLP-oriented benefits of TCC parallelization are fairly orthogonal to the acceleration of each individual transaction's execution that occurs when using ILP techniques *within* individual processors. As a result, ILP-based execution of individual transactions faster (or slower) than 1.0 instructions per cycle should simply improve (or decrease) TCC system performance proportionally, at least until the available commit packet broadcast network is saturated, when no further improvement from either ILP or TLP is possible. In addition to varying the number of processors, we also vary the value of three key TCC system parameters: the amount of bandwidth available on the network, the overhead of arbitration for commit order, and the latency of violation recovery. Table 1 presents the values selected for the parameters in order to describe three potential TCC configurations: ideal (infinite bandwidth, zero overheads), single-chip/CMP (high bandwidth, low overheads), and single-board/SMP (medium bandwidth, higher overheads).

Table 2 presents the applications used for this study and the transaction programming constructs employed to parallelize them. We selected these applications because they represent a diverse set of concurrency patterns including dense loops (*LUFactor*), sparse loops (*equake*), task parallelism (*SPECjbb*), and producer-consumer parallelism (*MPEGdecode*). This diversity is important to assess the usefulness and completeness of TCC programming. For the Java applications, we used the Kaffe JVM [38] within the execution driven simulator.

The Kaffe JVM required some simple modifications to be transaction friendly. We removed Java monitors used for synchronized

Table 2: Characteristics of applications used for our analysis.

Source Language	Benchmark	Application Description	Source	Input	Lines of Code	Primary TCC Parallelization
Java	Assignment	Resource allocation solver	jBYTEmark [6]	51x51 array	556	*Loop:* 2 ordered, 9 unordered
	MolDyn	N-body code modeling particles	Java Grande [17]	2048 particles	615	*Loop:* 9 unordered
	LUFactor	LU factorization and triangular solve	jBYTEmark [6]	101x101 matrix	516	*Loop:* 2 ordered, 4 unordered
	RayTrace	3D ray tracer	Java Grande [17]	150x150 pixel image	1,233	*Loop:* 9 unordered
	SPECjbb	Transaction processing server	SPECjbb [34]	230 iterations w/o random	27,249	*Fork:* 5 calls (one per transaction type)
C	art	Image recognition / neural network	SPEC2000 FP [35]	ref.1	1,270	*Loop:* 11 unordered & chunked
	equake	Seismic wave propagation simulation	SPEC2000 FP [35]	ref	1,513	*Loop:* 3 unordered
	tomcatv	Vectorized mesh generation	SPEC95 FP [35]	256x256	346	*Loop:* 7 unordered
	MPEGdecode	Video bitstream decoding	Mediabench [24]	mei16v2.m2v	9,834	*Fork:* 1 call

Table 3: Summary of optimizations used in our applications. Results are for the 8 processor CMP configuration.

Benchmark	Optimization	Cumulative Lines Changed	Cumulative Speedup	Useful %	Wait %	Violate %	Idle %	Median / Mean Transaction Size (instructions)		75% Write State (64B lines)
Assignment	Base	11	2.58	32.3	11.9	51.9	4.0	850	1097	1
	+ unordered	20	2.75	34.4	3.7	55.3	6.6			1
	+ private reductions	32	6.46	80.8	9.7	—	9.5			1
MolDyn	Base	4	5.93	74.1	8.6	1.9	15.4	76	77	0
	+ unordered	8	6.33	79.1	2.0	2.0	16.8			0
	+ private reductions	20	6.46	80.8	2.1	—	17.2			0
LUFactor	Base	6	5.03	62.9	9.3	15.9	12.0	44	459	2
	+ unordered	10	5.24	65.5	7.3	14.4	12.9			2
RayTrace	Base	1	3.73	46.6	19.0	0.4	34.0	147	167	2
	+ unordered	2	4.28	53.5	3.2	1.3	42.0			2
	+ private reductions	6	4.48	56.0	3.4	—	40.6			2
SPECjbb	Base + objCount	13	1.87	23.4	—	37.7	38.9	—	148,244	195
	+ privatization	33	3.87	48.4	0.1	46.3	5.2	—	144,054	190
	+ two `t_commit`	35	5.62	70.3	0.2	22.9	6.6	—	76,728	94
art	Base (chunked loop)	11	1.05	13.1	1.8	82.4	2.8	1880	1777	33
	+ private reductions	101	6.21	77.6	9.1	—	13.3		1781	33
	+ loop fusion	113	6.91	86.4	0.6	—	13.0		2065	33
equake	Base	6	3.60	45.6	0.4	54.0	—	130	471	2
	+ `t_commit`	8	7.38	91.2	4.6	4.2	—		146	1
	+ loop fusion	12	7.49	91.5	4.2	4.3	—	131	180	1
tomcatv	Inner loop (chunked)	16	2.26	28.3	4.6	57.9	9.2	59	88	3
	Outer loop	7	7.83	97.9	0.3	—	1.8	12,268	14,584	97
MPEG-decode	Base + prescanning	454	5.89	73.6	17.1	0.1	9.2	—	626,060	154

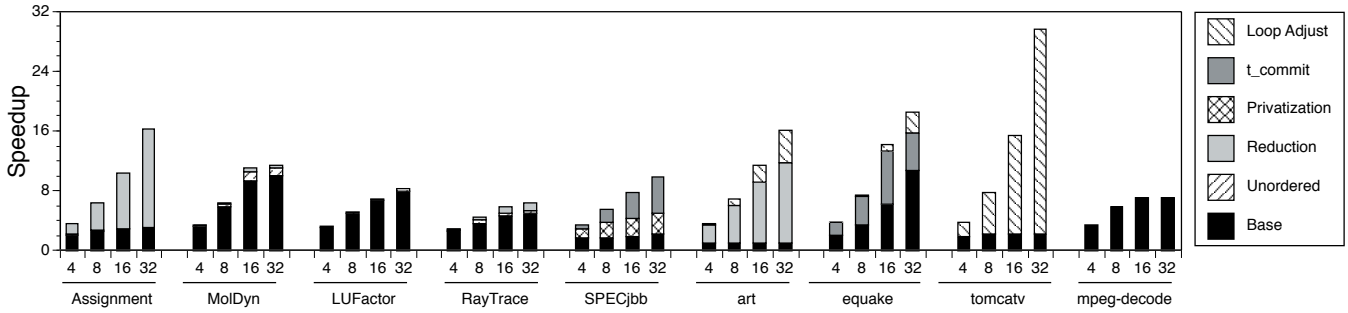

Figure 3: Improvements in CMP configuration speedup provided by the various optimizations, for varying numbers of processors. The 8-processor column graphs values from column 4 of Table 3.

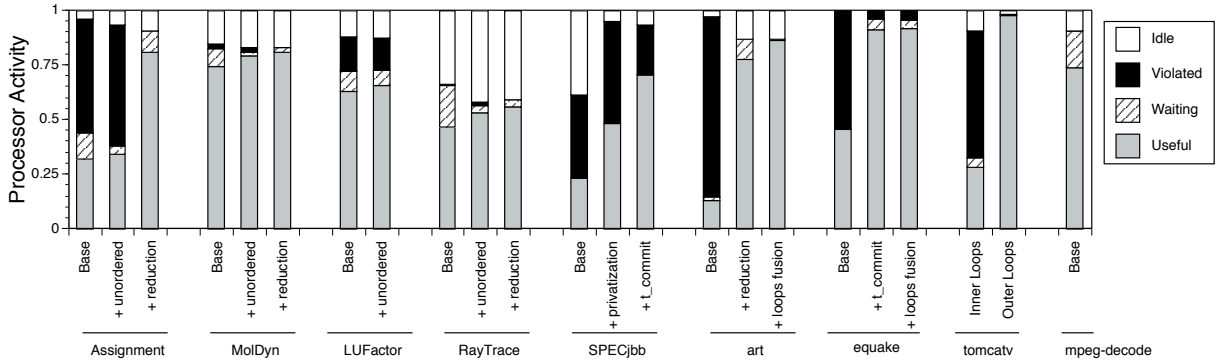

Figure 4: How the time breakdowns changed as optimizations were applied to the 8-processor CMP configuration. These are the values from columns 5–8 of Table 3, in graphical form.

objects, because transactions can be used to inherently guarantee atomic access. We also modified memory allocation routines to eliminate the chance of spurious violations caused by multiple processors accidentally allocating the same portion of the heap as different data objects simultaneously. This just required dividing the otherwise shared heap into separate "allocation pools" for each processor's memory routines, a change that is also necessary to build efficient non-transactional parallel memory allocation routines.

4.2. Application Studies

Table 3 presents the performance results for the 8 processor CMP configuration. For each benchmark, it lists the number of lines changed and the speedup over a single processor system. Figure 3 shows the same speedups graphically, along with ones for CMP configurations with different numbers of processors. Table 3 and Figure 4 also break down the average execution time on each processor into time executing useful transactions, time waiting for transactions to commit, time spent on transactions that violate, and time idling due to lack of parallelism (i.e. sequential code). We also list the 75th percentile of the buffer space required for a transaction (reported in 64-byte cache lines). Most applications require less than one Kbyte of write buffer. However, even the largest transactions for an unoptimized version of SPECjbb require less than 32 KBytes, which are reasonable to implement in modern processors. Further analysis of buffer space requirements is in [12].

Table 3 and the two figures include both the baseline results and those after applying optimizations described in Section 3. The baseline results show significant speedup with minimum modifications to the sequential code, but are typically not optimal. We obtained feedback information about transaction violations and overheads from the baseline TCC runs and used it to focus performance tuning on transactions with problematic behavior. The optimized applications achieve speedups ranging from 4.5 to 7.8 on an 8 processor CMP-configuration TCC system. The main contribution of the optimizations was to reduce the amount of time lost to violations, to adjust transaction sizes, and/or to limit buffer requirements. Note that the optimizations required that less than 5% of the original sequential source code lines be modified in all applications except *assignment* and *art*, which required slightly more.

The following sections discuss our insights on parallelizing and optimizing each application. The analysis is broken up into four parts, based on input source language (Java or C) and the type of parallelization (loop or fork) used.

4.2.1. Automatically Parallelized Java Loops

We automatically parallelized loop-based Java applications using the Jrpm dynamic compilation system [7]. Jrpm used ordered transactional loops in all cases. After examining the initial results from the ordered loops, we focused on the most significant loops to determine whether they could safely be executed in an unordered manner. Where possible and safe, we guided Jrpm to use unordered loops for performance improvement. Finally, after successfully parallelizing each application with Jrpm, we simulated each benchmark on top of the modified Kaffe JVM described in Section 4.1.

Assignment: This application includes largely parallel loops with only occasional dependencies, which are difficult to analyze statically using a conventional compiler. Most loops are 2-level nested. Jrpm usually generated transactions from the outer loop, because the inner loops were typically too small to parallelize. The baseline

speedup with ordered transactions is 2.58. Reduction privatization eliminates nearly all transaction violations and boosts speedup to 6.46. Reduction privatization is even more critical for greater numbers of processors, as the length of the critical dependency arcs imposed by the unmodified reduction limits speedups to less than 3, no matter how many processors are used. Higher speedup cannot be achieved primarily due to sequential portions of the application.

MolDyn: Moldyn includes parallel loops with infrequent dynamic dependencies. The most significant optimization was loop chunking due to the small size of loop iterations. Jrpm automatically performed chunking in the baseline version. Unordered transactions and reduction privatization provide small additional benefits.

*LUFactor:*This application contains many parallel loops, with two of them dominating the execution time. Unlike most other applications in our suite, the lengths of the transactions within each parallel loop varies significantly. Both have loop-carried dependencies, but the distance is large enough so that it only limits performance for large processor counts.

RayTrace: This application spends most of its time in a single loop. Unordered transactions and reduction privatization provide some improvement over the baseline speedup. Nevertheless, the overall speedup (4.48) is limited by the large sequential code regions.

4.2.2. Java with Forking: SPECjbb

SPECjbb is a server-side Java benchmark that simulates order processing in a 3-tier enterprise system. Its execution is divided up into separate "warehouses," which are essentially explicitly parallel. Hence, parallelization across warehouses is trivial for all kinds of parallel systems, including TCC. For this study, we parallelized SPECjbb *within* each warehouse, which is much more difficult with traditional means. Each warehouse involves task-queue processing with hard-to-analyze dependencies through inventory records. The main loop of SPECjbb for a warehouse iterates over five task types: new orders, payments, order status, deliveries, and stock levels. New orders and payments are weighted to occur ten times more often than other tasks, and the actual order of transactions is randomized. We parallelized this loop by generating unordered transactions for each task using the fork mechanism, achieving an initial speedup of 1.87.

Feedback-based tuning of this application consisted of several different steps targeting idle time reduction and removal of violations. The first step removed some remnants of an unnecessary object counter that caused spurious violations. We also found two key variables that needed privatization in order to work well in an unordered manner. The first was the seed to the random number generator used to generate tasks. The second was scratch-space objects, which were reused from one task to the next in the original code in order to avoid memory allocation. Finally, we found two opportunities where the judicious use of `t_commit` allowed us to split payment and new order tasks into two transactions, one for inventory processing and one for output display formatting. Because of the large size of transactions in SPECjbb, the smaller transactions that were created by this splitting significantly reduced the amount of useful work that was discarded when violations did occur, a factor that was especially critical with larger numbers of processors. Splitting also had the additional benefit of reducing write buffer requirements for the application by about half.

4.2.3. Manually Parallelized C Loops

We parallelized three SPEC floating-point applications with loop level parallelism. All three have nested loops of various depths that can be parallelized using variations of `t_for`.

art: This program first trains a neural network and then uses it to match images. Both of its phases iterate over a routine that includes a five-level nested loop that computes the output of the neural network. We parallelized the loops that iterate over neurons in the F1 layer because all other loop levels had limited parallelism or loop-carried dependencies. These loops contain critical reduction variables, hence significant speedups were achieved only after privatizing reductions. Loop fusion provided some additional improvement for an overall speedup of 6.91. The use of unordered transactions, while possible, did not provide any significant benefits for this application.

equake: This benchmark consists of series of loops operating on a sparse matrix. The baseline version used unordered transactions to automatically handle load balancing across transactions and achieved a speedup of 3.60. The main optimization for equake was transaction splitting, which reduced the frequency of violations, at the cost of an increase in commit overhead due to the significantly shorter transactions. To counter this, we fused loops together where possible to lengthen the transactions again, achieving a slight increase in performance that became more significant with larger numbers of processors. The overall speedup was 7.49.

tomcatv: This application includes five two-level nested loops that operate on the entire mesh. The inner loops require ordered transactions to handle the loop-carried dependencies correctly, which tend to limit performance to less than 3 no matter how many processors are used. Outer loops, on the other hand, can be parallelized with unordered transactions, leading to near perfect speedup. The drawback of outer-loop parallelization is that the large transactions need larger write buffers, which could be an issue with larger datasets since the size of these outer loop transactions is proportional to the size of the input array.

4.2.4. C with Forking: MPEG-2 Decode

MPEG-2 decode exhibits non-trivial concurrency patterns with complex dependencies. Its code iterates over frames, slices, and macroblocks. At the macroblock level, parallelism is too fine-grained and is better targeted by ILP techniques in each processor. On the other hand, parallelizing at the frame level would exceed buffer limitations in most reasonable TCC systems. Hence, we parallelized at the slice level, which results in very large transactions, but ones that are still feasible because they write out only a reasonable amount of state (less than 10KB, allocated as 64-byte lines) and do not have any dependencies.

The primary difficulty in MPEG-2 decode is the sequential dependencies due to the use of the variable-length codes in the input bitstream. Until a slice has been decoded, the starting position of the next one is unknown. This behavior lends itself naturally to forking, where a serial parent does bitstream decoding and then forks a transaction to process the decoded data for each slice. This follows the same model as the example in Figure 2c, with the bitstream decoding in the IF blocks and data processing in the EX blocks. To reduce the serialization effect due to sequential bitstream decoding, we also applied a prescan algorithm that requires the sequential

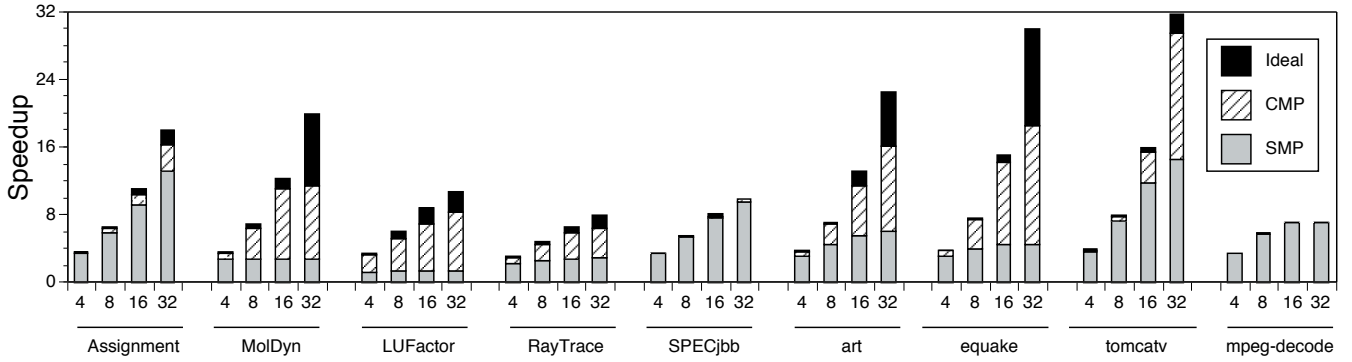

Figure 5: Overall speedup obtained in different hardware configurations.

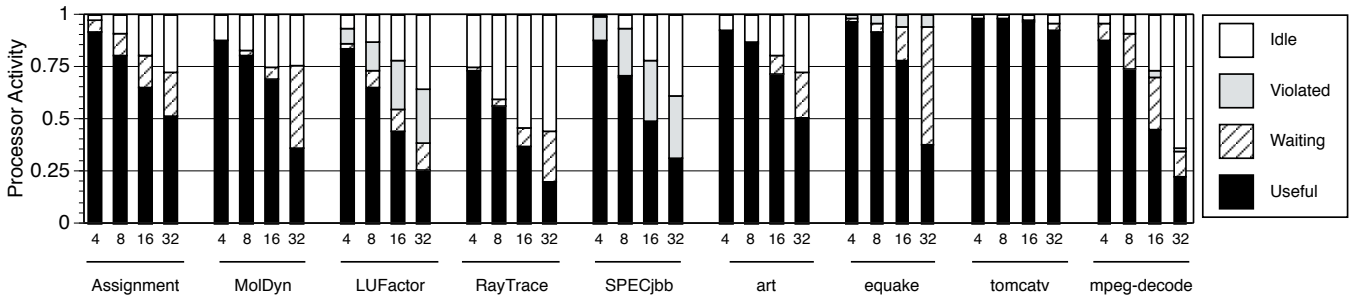

Figure 6: Breakdown of how different numbers of parallel processors spent their time, in the CMP configuration.

parent to identify only slice boundaries, and not fully decode the input bitstream [4]. The overall speedup for MPEG-2 decode was 5.89, with most of the source code changes going into implementing the prescanning algorithm.

4.3. Overall Performance

Figure 5 presents the best achieved speedups for three configurations of a TCC system (ideal, CMP, SMP) with the number of processors ranging from 4 to 32. Figure 6 complements these results with the final execution time breakdown for all different processor counts in the case of the CMP configuration.

For six out of nine benchmarks (*Assignment*, *LUFactor*, *RayTrace*, *SPECjbb*, *tomcatv*, and *MPEG-2 decode*), CMP performance closely tracks ideal performance for all processor counts. All six applications achieve speedups of 8 or above with 16 processors. *LUFactor* and *SPECjbb* are limited for large processor counts by a combination of unavoidable violations and some regions that lacked enough parallelism for the increased processor count, causing extra processors to idle. *Assignment* and *RayTrace* are mostly limited by the large sequential code regions, which result in an immense amount of processor idle time with larger numbers of processors. On the other hand, *tomcatv* scales perfectly to 32 processors, with only small increases in idle and waiting time and virtually no transaction violations. Finally, *MPEG-2 decode* speedup flattens at 16 processors, because in our sample input stream there were only 16 slices per frame, so any additional processors were idle. Overall, these results demonstrate that high performance can often be achieved with TCC through only minor modifications to sequential code.

For *MolDyn*, *art*, and *equake*, the ideal configuration has a significant advantage over the CMP configuration for the 32 processor case. As shown in Figure 6, the 32-processor CMP configuration incurs a large amount of waiting time. In this case, it is not due to load imbalance, but instead due to insufficient commit bandwidth. Too many of the 32 processors are trying to commit simultaneously at any point in time, resulting in serialization for access to the commit network. Luckily, solving bandwidth problems like this should generally be straightforward within a chip, especially for TCC systems that exchange only large messages between nodes.

The SMP TCC configuration achieves good speedups for 4–8 processors across the board, but exhibits little benefit with more processors for most applications. Of our applications, only *assignment*, *SPECjbb*, and *tomcatv* managed to use the additional processors to significant advantage, with *SPECjbb* notably almost completely unaffected by the configuration limits. This is actually a promising result, as online server applications like *SPECjbb* would probably be the primary application class that might drive the development of TCC on systems larger than CMPs. For the rest of the applications, the major bottleneck is again commit bandwidth, as four bytes per cycle is often too little for communicating between 16–32 parallel transactions. However, with higher commit bandwidth, the SMP configuration could achieve speedups similar to the CMP one, since the higher overheads incurred by board-level arbitration had much less effect on the overall speedup than the commit bandwidth.

5. RELATED WORK

TCC builds on ideas from previous hardware-based transaction work, designs for hardware thread-level speculation, database transaction processing, software-only transactional memory, and other forms of transactional programming, generally from the area of reliability research. This section discusses related work from these varied fields, focusing mostly on issues relating to parallel application development.

Previous Hardware Transaction Proposals: Transactions have been used in previous proposals to allow execution through locks or past barriers in order to reduce the pressure for optimal placement of synchronization primitives in conventional multithreaded code [15, 20, 26, 30]. TCC improves on this work by using transactions all the time. In addition to simplifying hardware by eliminating conventional cache coherence, it completely eliminates the need for conventional locks and barriers in the application code, using the abstraction of transactions to describe *all* synchronization.

Thread-level Speculation (TLS): Several groups have demonstrated speculative parallelization of sequential code using TLS hardware [11, 22, 33, 36]. Even though TCC draws upon TLS work, TCC differs in two basic ways. From the hardware perspective, TLS layers speculative execution of threads on top of a conventional cache coherence and consistency protocol, and generally allows speculative threads to communicate results to other speculative threads continuously. On the other hand, TCC completely replaces the underlying coherence and consistency protocol, and allows transactions to make their write state visible *only* at commit time, which should generally make it scalable to larger numbers of processors. From the software perspective, TLS always requires that threads commit in a single, predefined order [29], while TCC allows programmers to have full control over transaction sequencing, so that they can specify no or partial ordering for transaction commit. The similarity of support for speculation in TCC and TLS leads to similar issues during performance tuning. Techniques for minimizing data dependencies and optimizing transaction size to deal with overheads and buffering requirements, much like those we used with TCC-based parallelization, have also been considered in the context of TLS in [27, 37].

Database Transaction Processing: The usefulness of transactions as an abstraction for concurrency, atomicity, consistency, and isolation has been extensively studied by the database community [9]. TCC borrows the notion of transactions from databases in order to simplify the development of general parallel programs. However, database transactions are typically heavyweight and are only used in large transaction processing systems. In contrast, TCC transactions are lightweight, have direct hardware support, and are fully visible to application programmers. Nevertheless, the speculative execution of transactions in TCC is based on the idea of optimistic concurrency control from databases [23].

Software-only Transactional Memory: Several researchers have explored software-only transactional memory or transactional datastructures [13, 14, 32]. The focus of this work has been to replace locks with non-blocking primitives implemented with software buffering or versioning. In contrast, our transactions execute at all times and are accelerated by direct hardware support. In addition, TCC transactions provide memory consistency similar to that of lazy and delayed consistency models [8, 19, 21], which postpone communicating and processing of consistency events until release points in the program code.

Transactional Programming for Reliability: The Tran-C programming language in the IBM Encina system includes built-in transactional semantics [16]. However, Tran-C uses transactions for application-level failure atomicity and recovery. For parallel programming, Tran-C relies on conventional threads with locks and mutexes. In a similar vein, the Java language provides the "try-catch-finally" programming construct which allows programmers to specify application level error detection and recovery code [3]. Oplinger and Lam used this construct along with TLS mechanisms to provide a low-overhead transactional programming model for error recovery [28]. In contrast to these efforts, TCC's all-transactional approach provides a unified approach for both optimistic parallelism *and* failure atomicity, an area of further research for us. The use of a single abstraction for both goals greatly simplifies application development.

6. CONCLUSIONS

We have shown that by using TCC, it is possible to parallelize a wide range of applications with a few simple programming language constructs. Instead of using explicit synchronization with locks, these constructs rely instead on the atomic property of transactions to guarantee exclusive access to shared variables. Furthermore, because TCC allows for speculative transactions, the programmer does not have to guarantee that the transactions are independent to get correct program behavior. On the other hand, one aspect of transaction programming that the programmer must specify for correct program operation is the commit ordering of transactions (ordered, unordered or partially ordered); however, we view this as the key insight that programmers should be able to provide when parallelizing programs. Of course, one can always retain ordered transactions for safety purposes if there is ever any uncertainty. Furthermore, we believe that reasoning about transaction order is much simpler than reasoning about individual loads and stores, which is required with programming for conventional memory consistency models [1].

Improving the parallel performance of an application is a key component of parallel programming, and is often required for TCC programs. However, unlike conventional shared memory parallel programing, where performance improvement is largely a process of trial and error, we expect TCC systems to provide specific performance feedback in the form of transaction rollback statistics. These statistics are much more meaningful to the programmer than cache misses would be in a conventional shared memory multiprocessor because they are associated with programmer-defined transactions and variables. We have shown that these statistics can be used to direct the use of programming optimization techniques such as loop chunking, reduction and buffer privatization, and transaction splitting to improve TCC application performance significantly. The performance we have obtained with these optimizations over a wide range of applications is excellent, and demonstrates both the programming ease and performance potential of TCC. Also, although we performed these optimizations manually, since many require only fairly mechanical adjustments to code that can be pinpointed by feedback, we expect that most will eventually be applied automatically by either a static or dynamic compiler, which will further reduce the burden on the programmer.

Finally, we view the use of TCC hardware and TCC programming language support as a synergy of hardware and software that will provide a much gentler transition from sequential programming to parallel programming than that provided by any previous parallel programming paradigm. As a result, TCC hardware and software will be an important catalyst in transforming parallel processing for speedup of individual applications from a niche activity to a widespread technique.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, Vol. 29 No. 12, pp. 66–76, December 1996.

[2] V. Agarwal, S. Hrisikesh, D. Burger, and S. Keckler, "Clock Rate vs IPC: The End of Road for Conventional Microarchitectures," *27th Intl. Symposium on Computer Architecture*, pp. 248–259, Vancouver, Canada, June 2000.

[3] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language, Third Edition*, Addison-Wesley Professional, 2000.

[4] A. Bilas, J. Fritts, and J. P. Singh, "Real-time parallel MPEG2 decoding in software," *Proc. 11th International Parallel Processing Symposium*, Geneva, Switzerland, 1997.

[5] Broadcom Corp., "The Broadcom BCM-1250 Multiprocessor," Presentation at *2002 Embedded Processor Forum*, April 2002.

[6] Byte Magazine, *jBYTEmark Benchmark*, http://www.byte.com, CMP Media LLC, 1999.

[7] M. K. Chen and K. Olukotun, "The Jrpm System for Dynamically Parallelizing Java Programs," *Proceedings of the 30th International Symposium on Computer Architecture (ISCA-30)*, pp. 434–445, June 2003.

[8] M. Dubois, et. al., "Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs," *Proceedings of Supercomputing '91*, November 1991.

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

[10] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, 1994.

[11] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE MICRO Magazine*, pp. 71–84, March-April 2000.

[12] 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," *Proceedings of the 31st Annual Symposium on Computer Architecture (ISCA-31)*, pp. 102–113, June 2004.

[13] T. Harris and K. Fraser, "Language Support for Lightweight Transactions," *18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2003)*, October 2003.

[14] M. P. Herlihy, V. Luchangco, M. Moir, and W. M. Scherer, "Software Transactional Memory for Dynamic-sized Data Structures," *Proceedings of the 22nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, July 2003.

[15] M. Herlihy and J. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proceedings of the 20th International Symposium on Computer Architecture (ISCA-20)*, pp. 289-300, 1993.

[16] IBM Corporation, *Encina Transactional-C Programmer's Guide and Reference for AIX*, SC23-2465-02, 1994.

[17] Java Grande Forum, *Java Grande Benchmark Suite*, http://www.epcc.ed.ac.uk/javagrande/, 2000.

[18] R. Kalla, B. Sinharoy, and J. Tendler, "Simultaneous Multi-threading Implementation in POWER5," *Conference Record of Hot Chips 15 Symposium*, Palo Alto, CA, August 2003.

[19] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pp. 279–283, Orlando, FL, 1999.

[20] T. Knight, "An Architecture for Mostly Functional Languages," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, August 1986.

[21] L. I. Kontothanassis, M. L. Scott, and R. Bianchini, "Lazy Release Consistency for Hardware-Coherent Multiprocessors," *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

[22] V. Krishnan and J. Torrellas, "A Chip Multiprocessor Architecture with Speculative Multithreading," *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, Vol. 48 No. 9, pp. 866–880, September 1999.

[23] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems,* Vol. 6 No. 2, June 1981.

[24] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communication systems," *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-97)*, Research Triangle Park, NC, December 1997.

[25] B. Lewis and D.J. Berg, *Multithreaded Programming with Pthreads*, Prentice Hall, 1998.

[26] J. Martinez and J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Parallel Applications," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pp. 18–29, October 2002.

[27] K. Olukotun, L. Hammond, and M. Willey, "Improving the Performance of Speculatively Parallel Applications on the Hydra CMP," *Proceedings of the 1999 ACM International Conference on Supercomputing (ICS-99)*, pp. 21–30, Rhodes, Greece, June 1999.

[28] J. Oplinger and M. S. Lam, "Enhancing Software Reliability with Speculative Threads," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pp. 184–196, October 2002.

[29] M. K. Prabhu and K. Olukotun, "Using Thread-Level Speculation to Simplify Manual Parallelization," *Proceedings of the Principles and Practice of Parallel Programming (PPoPP)*, pp. 1–12, June 2003.

[30] R. Rajwar and J. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pp. 5–17, October 2002.

[31] R. Raman, "UltraSparc Gemini: Dual CPU Processor," *Conference Record of Hot Chips 15 Symposium*, Palo Alto, CA, August 2003.

[32] N. Shavit and S. Touitou, "Software Transactional Memory," *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pp. 204–213, Ottawa, Canada, August 20–23, 1995.

[33] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar processors," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425, Santa Margherita Ligure, Italy, June 1995.

[34] Standard Performance Evaluation Corporation, *SPECjbb2000 v1.01*, http://www.spec.org/jbb2000/, Warrenton, VA, 2000.

[35] Standard Performance Evaluation Corporation, *SPEC**, http://www.specbench.org/, Warrenton, VA, 1995–2000.

[36] J. Steffan and T. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pp. 2–13, Las Vegas, Nevada, 1998.

[37] T.N. Vijaykumar and G. S. Sohi, "Task Selection for a Multiscalar Processor," *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, pp. 81–92, December 1998.

[38] T. Wilkinson, *Kaffe Virtual Machine*, http://kaffe.org, 1997–2002.