TRANSACTIONAL MEMORY COHERENCE AND CONSISTENCY (TCC)

TABLE OF CONTENTS

1 INTRODUCTION

Parallel processors have become increasingly common and more densely packed in recent years. In the near future, these systems will provide many conventional processors packed together into chip multiprocessors (CMPs) or single-board systems interconnected with some form of high-bandwidth communication bus or network. With these systems, enough bandwidth can be provided between processors to even allow the *broadcast* of significant amounts of data and/or protocol overhead between all of the processor nodes over a low-latency unordered interconnect [4, 5, 7, 19, 24, 26]. Overwhelmingly, designers of today□s parallel processing systems have chosen to use one of two common models to coordinate communication and synchronization in their systems: message passing or shared memory. Given the advent of newer systems with immense interprocessor bandwidth, however, we wondered if it would be possible to take advantage of this bandwidth to simplify the protocols used to manage communication and synchronization between processors in a system.

Message-passing is a system that supports relatively simple hardware configurations, such as clusters of workstations, but makes programmers work hard to take advantage of the hardware. The programming model is one of many independent nodes that must pass explicit messages between each other when communication is necessary. Messages also implicitly synchronize processors as they are sent and received. This technique typically makes the underlying

hardware much simpler by making programmers concentrate their communication into a relatively small number of large data packets that can flow throughout the system with relatively relaxed latency requirements. To facilitate this, programmers must divide data structures *and* execution into independent units that can execute efficiently on individual processor nodes.

In contrast, shared memory adds additional hardware to provide programmers with an illusion of a single shared memory common to all processors, avoiding or minimizing the problem of manual data distribution. This is accomplished by tracking shared cache lines as they move throughout the system either through the use of a snoopy bus-coherence protocol over a shared bus [13, 28] or through a directory-based coherence mechanism over an unordered interconnect [3, 23]. Programmers must still divide their computation into parallel tasks, but all tasks can work with a single, common dataset resident in memory. While this model significantly reduces the difficulty inherent in parallel programming, especially for programs that exhibit dynamic communication or fine grain sharing, the hardware required to support it can be very complex [7]. In order to provide a **coherent** view of memory, the hardware must track where the latest version of any particular memory address can be found, recover the latest version of a cache line from *anywhere* on the system when a load from it occurs, and efficiently support the communication of large numbers of small, cache-line-sized packets of data between processors. All this must be done with minimal latency, too, since individual load and store instructions are dependent upon each communication event. Achieving high performance despite the presence of long interprocessor latencies is therefore a problem with these

systems. Further complicating matters is the problem of *sequencing* the various communication events constantly passing throughout the system on the granularity of individual load and store instructions. Unfortunately, shared memory does not provide the implicit synchronization of message passing, so hardware rules — memory **consistency** models [1] — have been devised and software synchronization routines have been carefully crafted around these rules to provide the necessary synchronization. Over the years, the memory consistency model has progressed from the easy-to-understand but sometimes performancelimiting sequential consistency [12, 22] to more modern schemes such as relaxed consistency [2, 9, 11]. The complex interaction of coherence, synchronization and consistency can potentially make the job of parallel programming on shared memory architectures difficult.

Both of these models therefore have drawbacks — message passing makes software design difficult, while shared memory requires complex hardware to get only a slightly simpler programming model. Ideally, we would like a communication model that, without raising memory consistency issues, presents a shared memory model to programmers and significantly reduces the need for hardware to support frequent, latency-sensitive coherence requests for individual cache lines. At the same time, we would like to be able to take advantage of the inherent synchronization and latency-tolerance of message passing protocols. Replacing conventional, cache-line oriented coherence protocols and conventional shared memory consistency models with a *Transactional memory Coherence and Consistency* (TCC) model can accomplish this.

2 *SYSTEM OVERVIEW*

Processors such as those in Fig. 1 operating under a TCC model continually execute speculative *transactions*.

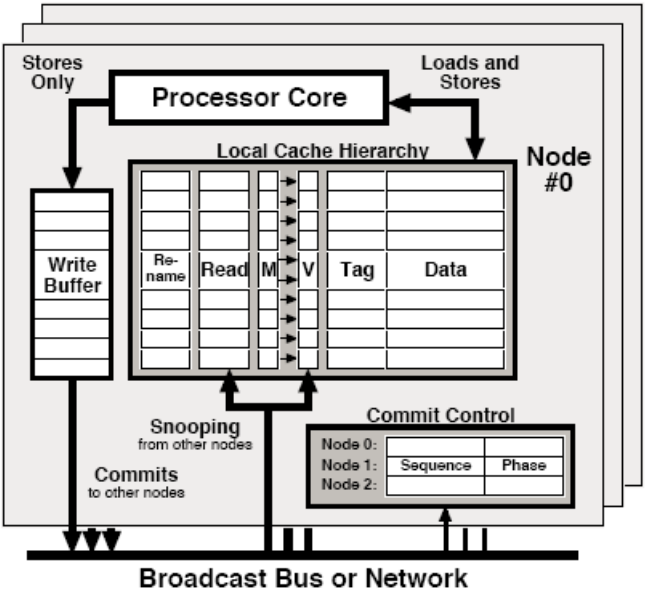

Figure 1: A sample 3-node TCC system.

A transaction is a sequence of instructions that is guaranteed to execute and complete only as an atomic unit. Each transaction produces a block of writes called the write state which are committed to shared memory *only* as an atomic unit, after the transaction completes execution. Once the

transaction is complete, hardware must arbitrate system-wide for the permission to commit its writes. After this permission is granted, the processor can take advantage of high system interconnect bandwidths to simply broadcast all writes for the entire transaction out as one large packet to the rest of the system. Note that the broadcast can be over an unordered interconnect, with individual stores separated and reordered, as long as stores from *different* commits are not reordered or overlapped. Snooping by other processors on these store packets maintains coherence in the system, *and* allows them to detect when they have used data that has subsequently been modified by another transaction and must rollback — 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 write state is a one-way operation. At the same time, since we only need to control the sequencing between entire transactions, instead of individual loads and stores, we leverage the commit operation to provide inherent synchronization *and* a greatly simplified consistency protocol.

This continual speculative buffering, broadcast, and (potential) violation cycle, illustrated in Fig. 3a, allows us to replace conventional coherence and consistence protocols simultaneously:

Consistence: Instead of attempting to impose some sort of ordering rules between individual memory reference instructions, as with most consistency models, TCC just imposes a sequential 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. As far as the

global memory state and software is concerned, *all* memory references from a processor that commits earlier happened "before" *all* memory references from a processor that commits afterwards, even if the references actually executed in an interleaved fashion. A processor that reads data that is subsequently updated by another processor□s commit, before it can commit itself, is forced to violate and rollback in order to enforce this model. Interleaving between processors□ memory references is *only* allowed at transaction boundaries, greatly simplifying the process of writing programs that make fine-grained access to shared variables. In fact, by imposing an original sequential program□s original transaction order on the transaction commits, we can effectively let the TCC system provide an *illusion of uniprocessor execution* to the sequence of memory references generated by parallel software.

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, they perform conventional invalidation (if the commit packet only contains 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 transaction during their currently executing transaction, they are forced to restart and reload the correct data. This hardware mechanism

protects against true data dependencies automatically, without requiring programmers to insert locks or related constructs. 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, as well.

Although some of the details and implementation alternatives add more complexity, this simple cycle is the backbone of the TCC system and underlies all other descriptions of the system throughout the rest of this paper.

2.1 Programming Model

For programmers, there is really only one requirement for successful transactional execution: the programmer must insert transaction boundaries into their parallel code occasionally (possibly with some hardware aid, see Section 4.2). No complex sequences of special instructions such as locks, semaphores, or monitors are *ever* necessary to control low-level interprocessor communication and synchronization. In many respects, this model is very similar to the technique of performing manual parallelization with assistance from thread-level speculation (TLS, see Section 3.2) hardware that we previously investigated in [29]. There is only one hard rule that programmers must keep in mind. Transaction breaks should *never* be inserted during the code between a load and

any subsequent store of a shared value (i.e. during a conventional lock□s critical region). Unlike with conventional parallelization, other "errors" will only cause reduced performance, instead of incorrect execution.

As a result of this model, parallelizing code with TCC is a very different process from conventional parallel programming because it allows programmers to make *intelligent* tradeoffs between programmer effort and performance. Basic parallelization can quickly and easily be done by identifying potentially interesting transactions, and then programmers can use feedback from runtime violation reports to refine their transaction selection in order to get significantly greater speedups. In a simplified form, parallel programming with TCC can be summarized as a threestep 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 pro-
• cessors. This is similar to conventional parallelization, which also requires that programmers find and mark parallel regions. However, the actual process is much 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 at runtime.

Specify Order: The programmer can optionally specify an ordering between transactions to maintain a program order that must be enforced. By default, no order is imposed between the commits of the various transactions, so different processors may proceed independently and commit as they encounter end-of-transaction instructions. However, most parallel applications have places where certain transactions

must complete before others. This situation can be addressed by assigning hardware-managed *phase numbers* to each transaction. At any point in time, only transactions from the "oldest" phase present in the system are allowed to commit. Transactions from "newer" phases are simply forced to stall and wait if they complete before all "older" phases have completed.

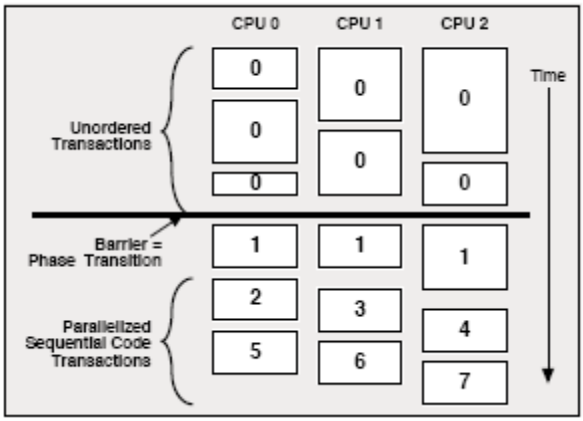

Figure 2: Timing illustration of how transactions (numbered blocks) running on three different processors are forced to commit by phase number sequence.

Fig. 2 illustrates how most important transaction sequencing events can be handled using phase numbers. The top half of the figure shows groups of unordered transactions, for which we simply keep the phase numbers identical. To form a barrier, all processors increment the phase number of transactions by 1 as they cross the barrier point, so that all pre-barrier transactions are forced to commit before any post-barrier transactions can complete. To parallelize sequential

code in a TLS-like fashion, we simply increment the phase number by 1 for each transaction created from the original sequential code, as is illustrated in the bottom half of Fig. 2. This forces the commits to occur in-order, which in turn guarantees that the parallel execution will mimic the load/store behavior of the original program. In addition to being easy to implement in hardware, this scheme is also guaranteed to be deadlockfree, since at least one processor is always running the "oldest" phase, and therefore able to commit when it completes. Also, in order to allow several phase-progression sequences to occur simultaneously in different parts of the system, we could add an optional *sequence number* to the hardware in order to separate out these different groups of phasings.

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. These optimizations usually improve code by making it follow these guidelines:

1. Transactions should be chosen to maximize parallelism and minimize the number of inter-transaction data dependencies. A few occasional violations are acceptable, but regularly occuring ones will largely eliminate the possibility for speedup in most systems.
2. Large transactions are preferable, when possible, as they amortize the startup and commit overhead time better than small ones, but . . . .
3. Small transactions should be used when violations are frequent, to minimize the amount of lost work, or when large

numbers of memory references tend to overflow the available memory buffering.

This streamlined parallel coding model therefore allows parallel programmers to focus on providing better performance, instead of spending most of their time simply worrying about correctness.

2.2 *Basic TCC System*

TCC will work in a wide variety of multiprocessor hardware environments, including a variety of CMP configurations and smallscale multichip multiprocessors. TCC cannot scale infinitely, for two reasons. First, TCC requires system-wide arbitration for the commit permission, either through a centralized arbiter or distributed algorithm. Second, TCC relies on *broadcast* to send the commit packets throughout the system. The algorithm is currently dependent upon some form of broadcast, although we examine some mechanisms to reduce bandwidth requirements in Section 6.3. Our scheme can work within any system that can support these two requirements in an efficient manner.

Individual processor nodes within a TCC system must have some features to provide speculative buffering of memory references and commit control, as was illustrated in Fig. 1. Each "node" consists of a processor core plus its own local cache hierarchy. The exact structure of the local cache hierarchy makes no difference to the coherence scheme, as long as all of the included lines maintain the following information in some way:

Read bits: These bits are set on loads to indicate that a cache line (or portion of a line) has been read speculatively during a

transaction. These bits are snooped while other processor nodes commit to determine when data has been speculatively read too early. If a write committed by another processor modifies an address cached locally with its read bit set, then a violation has been detected and the processor is interrupted so that it can revert back to its last checkpoint and start re-executing from there. In a simple implementation, one read bit per line is sufficient. However, it may be desirable to include multiple bits per line in order to eliminate false violation detections caused by reads and writes to different words in the same line.

Modified bit: There must be one for every cache line. These are set by stores to indicate when *any* part of the line has been written speculatively. These are used to invalidate *all* speculatively written lines at once when a violation is detected.

In addition, we can optionally include an extra set of bits in each cache line to help avoid false violations that could be caused by reads and writes to the *same* part of a cache line:

Renamed bits: These optional bits must be associated with individual words (or even bytes) within each cache line. They act much like "modified" bits, except that they can only be set if the *entire* word (or byte) is written by a store, instead of just any part of the associated region. Because individual stores can typically only write a small part of a cache line at a time, there must almost always be a large number of these bits for each line. If set, any subsequent reads from these words (bytes) do not need to set read bits, because they are guaranteed to only be reading locally generated data that cannot cause violations. Since these bits are optional, they can

be omitted entirely or only partially implemented (for example, in a node's L1 cache but not in its L2).

Cache lines with set read or modified bits may not be flushed from the local cache hierarchy in mid-transaction. If cache conflicts or capacity constraints force this to occur, the discarded cache lines must be maintained in a victim buffer (which may just hold the tag and read bit(s) if a line is unmodified) or the processor must be stalled temporarily. In the latter case, it must request commit permission, a process that may take some time if processors with "older" phases are present, and then hold this permission until the transaction completes execution *and* commits. This solution works because read and modified bits do not need to be maintained once commit permission has been obtained, as all "earlier" commits will have been guaranteed to complete at that point. However, since holding the commit permission for extended periods of time can have a severely detrimental impact on the overall system performance, it is critical that this mechanism only be used for infrequent, very long transactions.

The processor core must also have a way to checkpoint its register state at each commit point in order to provide rollback capabilities. This could be done either in hardware, by flash-copying the register state to a shadow register file at the beginning of each transaction, or in software, by executing a small handler to flush out the live register state at the start of each transaction. The hardware scheme could be incorporated into traditional register renaming hardware by flash-copying the register rename tables instead of the registers themselves. The software scheme would not require any modifications of the core at all, but such a scheme would obviously incur a higher overhead on the processor core at each commit.

Finally, the node must have a mechanism for collecting all of its modified cache lines together into a commit packet. This can be implemented as a write buffer completely separate from the caches or as an address buffer that maintains a list of the line tags that contain data that needs to be committed. We shall examine the size of write state to determine the amount of hardware that would be required to implement a typical write buffer beside the caches. The interface between this buffer and the system network should have a small "commit control" table that tracks the state (phase numbers) of other processors in the system in order to determine when it is within the "oldest" phase and free to arbitrate for commits. This simple mechanism can eliminate a great deal of spurious arbitration request traffic.

3 RELATED WORK

This paper draws upon ideas from two existing bodies of work, database transaction processing systems and ***thread-level speculation (TLS)***, and applies them to the field of cache coherent, shared memory parallel architectures. This section compares TCC with some key ideas from these two fields of knowledge.

3.1 Database Transaction Processing

Transactions are a core concept in database management systems (DBMS) that provide significant benefits to the database programmer [15]. In DBMS, transactions provide the properties of atomicity, consistency, isolation, and durability

(ACID). We have borrowed the fully transactional programming model from databases because we think that these properties will greatly simplify the development of generic parallel programs. The main difference between the transactions defined by the database programmer and those used by parallel programmers is size. The number of instructions executed and the amount of state generated by most parallel program transactions is much smaller than those used in database transactions. Therefore, a key element to using transactions for general purpose parallel programming is an efficient, hardware-based transaction execution environment.

The designers of DBMS have explored a wide range of implementation options for executing transactions while providing high transaction throughput. The work on optimistic concurrency [21] is the most relevant to the ideas we explore in this paper. Optimistic concurrency controls access to shared data without using locks by detecting conflicts and backing up transactions to ensure correct operation. In transactional coherence and consistency we extend the ideas of optimistic concurrency from DBMS to memory system hardware.

3.2 Previous Work in Transactions and TLS

From the hardware side, the origin of this work was in the early transactional memory work done by Herlihy [17] a decade ago. Our transactions have identical semantics to the model proposed in this paper. However, they proposed only using transactions occasionally, replacing only the critical regions of locks. As a result, it was more of an adjunct to existing shared memory consistency protocols than a complete

replacement. By running transactions at all times, instead of just occasionally, we are able to use the same concepts to completely replace conventional coherence and consistency techniques. However, the larger number of transaction commits in our model puts a great deal of pressure on interprocessor communication bandwidth, so practically speaking it would have been difficult to implement a model like ours a decade ago.

Our work also draws upon the wide variety of thread-level speculation (TLS) literature that has been published over the course of the past several years from the Multiscalar project [33], Stampede [36], Torrellas at the University of Illinois [20], and the Hydra project [16]. In fact, a TCC system can actually implement a very loosely coupled TLS system if all transactions are ordered sequentially. In this respect, it most closely resembles the Stampede design of a TLS system, as their TLS threads only flush out data from the cache to global memory at the end of each thread, much like our commits. However, Stampede layers the TLS support on top of a conventional cache coherence protocol. The other TLS systems provide much tighter coupling between processors and more automatic forwarding of data between executing threads, and are thereby further removed from TCC. As long as forwarding of data between speculative threads is not critical for an application, however, TCC□s performance in "all ordered transaction" mode can actually be competitive with these dedicated TLS designs.

Looking at the proposed hardware implementations, our example implementation is most similar to the Stampede or Hydra designs in its focus on a multiprocessor with a few flash-clearable bits attached to the private caches. However,

we chose this example design solely because it is an easy first step. A buffering scheme such as the ARB [10] or SVC [14], as proposed by the Multiscalar group, would also be able to handle the speculative buffering tasks required by TCC.

Comparisons can also be made between TCC and other proposals to adapt speculative mechanisms to improve the performance of conventional parallel programming models. For example, Martinez and Torrellas [25], Rajwar and Goodman [30, 31], and Rundberg and Stenstrom [32] have independently proposed how to speculate through locks and past barriers in recent papers. TCC performs both of these operations during normal operation. All execution now consists of transactions that can speculate through one or several different conventional locks at once, while speculation past phase barriers can occur if the implementation of TCC is double-buffered.

4 TCC IMPROVEMENTS

The basic protocols used to construct a TCC system and how they compare with existing coherence and consistency protocols are presented here. There are extensions and improvements to these basic protocols that could improve performance or reduce the bandwidth requirements of TCC in a real system environment. Let's see some of these potential improvements.

4.1 Double Buffering

Double buffering implements extra write buffers and additional sets of read and modified bits in every cache line,

so that successive transactions can alternate between sets of bits and buffers. This mechanism allows a processor to continue working on the next transaction even while the previous one is waiting to commit or committing, as is illustrated in Fig. 3 for several combinations.

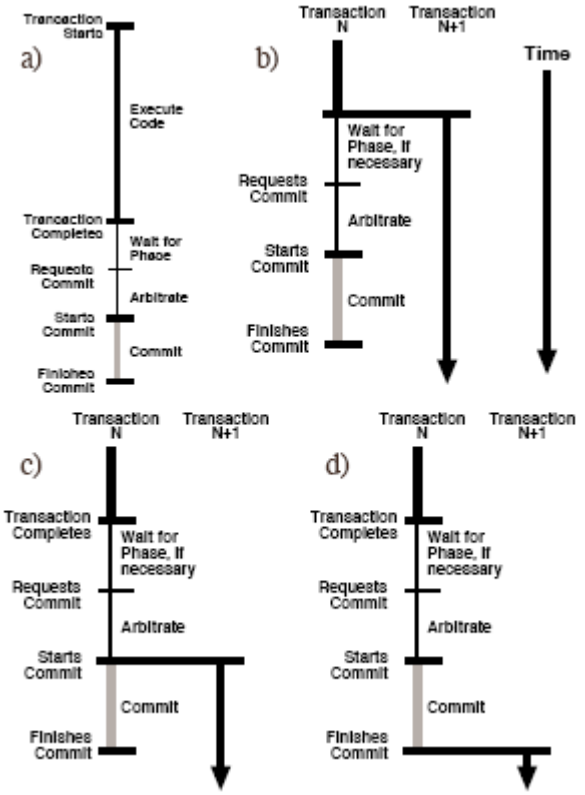

Figure 3: The effect of double buffering: a) a sample transaction timeline, b) double buffering of all speculative state, c) double buffering for write buffer but not read bits in cache, and d) pure single buffering.

In addition, this extension automatically lets processors that arrive at barriers early continue to speculate past them, as in [25, 30, 31, 32], without any additional hardware or API considerations.

The major expense of this scheme is in replicating additional sets of speculative cache control bits and write buffers. Adding buffers probably will not scale well past one additional set, but that first set (providing double buffering) should provide most of the potential benefit. If hardware register checkpoints are used, then additional sets of shadow registers would also be required to allow checkpoints to be taken at the beginning of each speculative transaction supported by hardware. Only one copy of the optional renamed bits is ever necessary, since these do not serve any further function after a transaction finishes executing instructions. All of these special bits should be flash-clearable at the end of transactions, to avoid tying up the hardware for many cycles on each commit, and it is also helpful if the modified bits can flashinvalidate their lines when the transaction aborts on a violation. More sophisticated versioning protocols, similar to those used in the SVC implementation of buffering for TLS [14], could be used to eliminate many of these circuit design issues, but we believe that flash-clearable bits are feasible in tag SRAMs.

4.2 Hardware-Controlled Transactions

In the base TCC system, programmers explicitly mark all transaction boundaries. However, it is also possible for hardware to play a role in marking transaction boundaries or sequencing the transaction commits once they have been initiated. There are three situations where hardware assistance would be helpful.

Hardware could divide program execution into transactions automatically as the speculative buffers overflow. This could achieve the optimal transaction size by not letting transactions get so large that buffering becomes a problem, while keeping them large enough to minimize the impact of any commit overhead. The most common situation when this might be helpful is for code that divides "naturally" into very large transactions, but when these transactions can be freely subdivided into smaller transactions. This is a common situation in programs that have already been parallelized in a conventional manner. While it is fairly easy for a software programmer to insert extra commit points into the middle of these transactions in order to keep the speculative buffering requirements manageable, it would be simpler for hardware to automatically insert transaction commits whenever the speculative buffers are filled, thereby automatically breaking up the large transaction into transactions that are sized perfectly for the available speculative buffer sizes. The only limitation on this technique is that it no longer guarantees atomic execution semantics within the large transaction, as the hardware is free to insert an extra transaction commit point anywhere. This limitation can be overcome, however, by allowing programmers to explicitly mark the critical regions within the large transaction where hardware *cannot* insert commits. If the buffers overflow within these regions, then the processor acquires commit permission early and holds it until the end of the "atomic region," when it finally inserts a commit.

Instead of breaking large transactions into smaller ones, we might also want to have the hardware automatically merge small transactions together into larger ones. This would allow us to automatically gain some of the advantages of larger

transactions, but at the potential risk of having the hardware merge critical transactions that need to complete and propagate results quickly.

Another useful way that hardware could interact with transactions is to occasionally insert "barriers" into long stretches of unordered transactions. This would simply consist of incrementing the phase number assigned to all new transactions, even if the software does not request a barrier explicitly. These occasional pseudobarriers would force all currently executing transactions to commit before allowing the system to progress further, effectively forcing all processors to make forward progress. Without this mechanism there is the possibility of starvation: a long transaction that makes no forward progress because it gets into an infinite loop of violations and restarts.

4.3 Localization of Memory References

One of the assumptions made so far is that all loads and stores must be speculatively buffered and broadcast throughout the system. However, it is often possible for programmers or compilers to give hints to the hardware that could reduce the need for buffering and, especially, for broadcast. For example, one way to reduce bandwidth is by marking some loads and stores as "local" ones that do not need to be broadcast. We applied this optimization to stack references in our analysis, because these references are guaranteed to be local within processors in most parallel systems, and do not need to be snooped by other processors. However, there are also other data structures that might be known as "local-only" to the programmer and/or compiler. These data structures could be marked either by locating them together in memory pages

marked by the OS as "local-only" or by accesses using special "load local" and "store local" opcodes. Either would allow the hardware to filter out these local references and, while still marking any changes for speculative rollback if necessary, avoid adding them to the list of data to be broadcast.

4.4 I/O Handling

A TCC system can handle I/O very easily. The key constraint is that a transaction cannot violate and rollback after input is obtained. When an attempt is made to read input, the current transaction immediately requests commit permission, just as if a buffer had overflowed. The input is only read after commit permission is obtained, when the transaction is *guaranteed* to never roll back. Outputs that require writes to occur in a specific order (like a network interface) can use a similar "pseudo-overflow" technique to force the writes to propagate out from the processor immediately, as stores are made. On the other hand, outputs that can accept potentially reordered writes (such as a frame buffer) may simply be updated at commit time, along with normal memory writes, thereby allowing higher performance. As a result, existing I/O handlers will work on TCC systems, although it will probably improve performance if transaction breakpoints are carefully placed within them. "Pseudo-overflows" to end transactions prematurely may also be helpful when events such as system calls and exceptions occur, but this is not necessarily required.

5 SIMULATION METHODOLOGY

As this paper is an initial evaluation to determine the overall potential of TCC, we chose to simulate a variety of parallel

integer and floating point benchmarks using a simplified hardware model that included many adjustable parameters to model a wide range of potential implementations of TCC systems. Our selection of applications and their associated datasets are summarized in Table 1. These applications come from a wide variety of different application domains: hand parallelized SPLASH-2 [38] programs, several floating point SPEC95 and 2000 [34] benchmarks parallelized semi-automatically with help from either a compiler (for Fortran) or TLS (for C), the **SPECjbb** transaction processing benchmark [34], and a variety of Java programs parallelized using automated TLS techniques [8] while running on the Kaffe JVM [37]. We parallelized the **SPECjbb** benchmark *within* only one of its warehouses, a more difficult task than the usual technique of parallelizing *between* warehouses, in order to demonstrate how TCC can replace complex locking structures.

Source	Program	Summary	Dataset	Parallelization
SPEC2000 FP [34]	art	image recognition / neural nets	Reference	TLS loops + manual fixes
	equake	seismic wave simulation	Reference	TLS loops + manual fixes
	swim	shallow water model	256x256 grid, 4 iterations	compiler
SPEC95 FP [34]	tomcatv	vectorized mesh generation	256x256 grid, 5 iterations	compiler
SPLASH2 [35; 38]	lu	dense matrix factorization	256x256, blocksize=16	all manual
	radix	radix sort	256K integers, radix 1024	all manual
	water-N2	N-body molecular dynamics	125 molecules	all manual
SPECjbb2000 [34]	SPECjbb	transaction processing	1 warehouse, 230 iterations	all manual

Source	Program	Summary	Dataset	Parallelization
Java Grande [18]	euler	flow equations in irregular mesh	33x9 mesh	automated, TLS-based
	fft	FFT kernel	1024 samples	automated, TLS-based
	moldyn	particle modeling	8x8x8x4	automated, TLS-based
	raytrace	3D raytracer	150x150 image	automated, TLS-based
jBYTEmark [6]	jbyte_B5	resource allocation	51x51 array	automated, TLS-based
	jbyte_B6	data encryption	—	automated, TLS-based
	jbyte_B8	neural network	35x8x8 network	automated, TLS-based
	jbyte_B9	dense matrix factorization	101x101 matrix	automated, TLS-based
SPECjvm98 [34]	mtrt	raytracer	200x200 image	automated, TLS-based
Java Code [27]	shallow	shallow water model	256x256 grid	automated, TLS-based

Some applications analyzed for transactional behavior.

Each of these benchmarks was run through a three-part investigative process. The first part consisted of examining existing benchmarks and inserting markers at the end of transactions and to replace conventional interprocessor synchronization. Afterwards, we ran the applications on an execution-driven simulator fixed to execute at one instruction per cycle, with perfect cache behavior (with real cache misses, this IPC will usually approximate the performance of an aggressive superscalar processor), and produce traces of all executed loads and stores (except stack references, which were guaranteed to be local to each processor) in the benchmarks. Once we had obtained traces of parallel execution from our

selection of benchmarks, we fed these traces into an analyzer that simulated the effects of running them in parallel on a very flexible transactional system. On this system, we were able to adjust parameters such as the number of processors, the commit bus bandwidth, speculative cache line bit configurations, and the overheads associated with various parts of the transactional protocol. The unusual characteristics of TCC allow such a simple simulation environment to still get reasonable performance estimates: the fixed ILP did not matter much because TCC is a thread level parallelism extraction mechanism, largely orthogonal to ILP extraction within the individual processor cores, and the fact that TCC only allows processor interaction at transaction commit made the precise timing of loads and stores *within* the transactions largely irrelevant. In fact, the most critical timing parameter that we obtained from simulation was the approximate cycle time of entire transactions. As a result, we were able to simulate a wide variety of potential system configurations with a reasonable amount of simulation time. While more detailed simulations will be necessary to investigate the full potential of TCC, this relatively simple study has allowed us to show what parts of the parallel computing design space are amenable to conversion to TCC, and to provide some estimates as to the buffering and bandwidth requirements that will be necessary for hardware support of TCC.

6 SIMULATION RESULTS

6.1 Limits of Available Parallelism

Our first results show the limits of parallelism available in our benchmarks that can be extracted with a TCC system. Fig. 4 shows the speedups that can be obtained on these applications as the number of processors varies from 1 to 32 in an "optimal" TCC system.

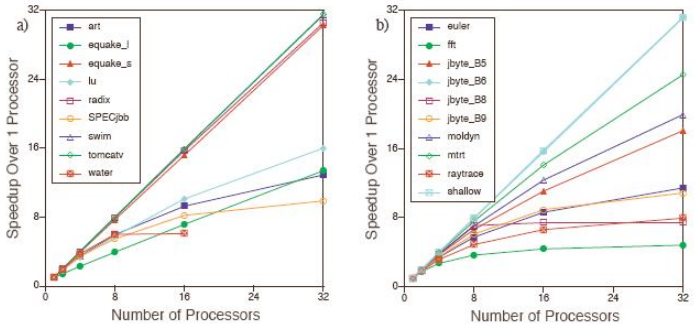

Figure 4: Speedups for varying numbers of processors with our manually parallelized benchmarks (a) and Java benchmarks with automated parallelism (b) on a perfect TCC system with 1 IPC processors, no memory delays, and ∞ commit bandwidth.

This perfect system has infinite commit bus bandwidth between the processors. The speedups achieved with several benchmarks are close to the optimal linear case, and many others are competitive when compared with previously

published results obtained on conventional systems in papers such as [38]. As is illustrated in Fig. 5, there are several reasons why speedups are limited. Sequential code remaining in several of the applications limits speedup through Amdahl's law ("idle" time). Load imbalance in parallel regions slows down **water** and **fft** ("waiting" time). Finally, while we were generally very successful at eliminating dependencies among transactions, some applications still suffer from occasional violations caused by true inter-transaction dependencies remaining in the programs. For example, in **SPECjbb** we eliminated *all* locks protecting various parts of the warehouse databases. As long as multiple transactions do not modify the same objects simultaneously, they may run in parallel, but since we have multiple processors running within the same warehouse, there is always a probability that simultaneous modifications may cause one of the transactions to violate.

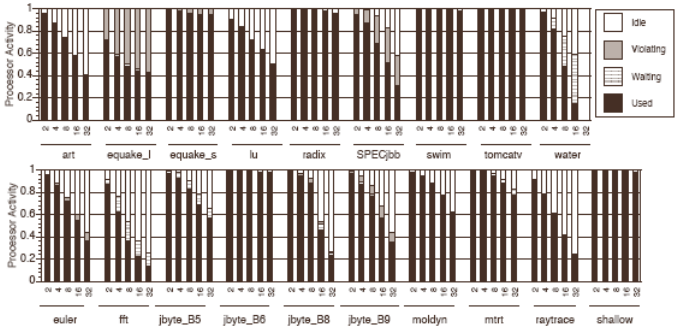

Figure 5: Distribution of execution time on the perfect TCC system's processors between useful work, violated time (failed transactions), waiting time (load imbalance in parallel code), and idle time (time waiting during sequential code).

As a further example of this, we show results for **equake** parallelized into versions with both long (**equake_l**) and short (**equake_ s**) transactions. The longer transactions tended to incur more violations and experienced much less speedup. On benchmarks like this, the positioning and frequency of transaction commits can clearly be critical.

We also parallelized several versions of **radix** to have different transaction sizes. However, **radix** has been manually tuned to eliminate dependencies and load imbalance between processors, so baseline speedups changed very little across the various versions.

6.2 Buffering Requirements for Typical Transactions

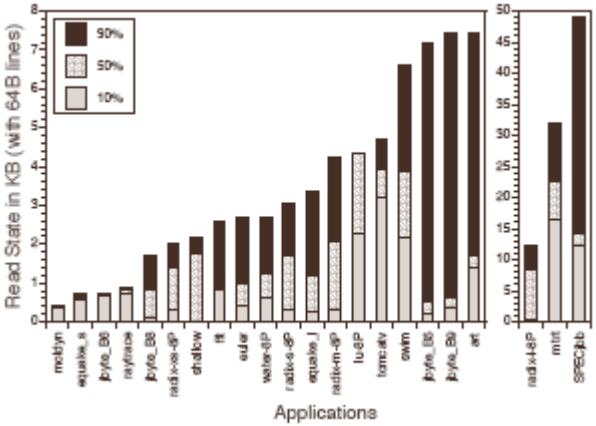

Figure 6: State read by individual transactions with store buffer granularity of 64-byte cache lines. We show state required by the smallest 10%, 50%, and 90% of iterations.

The most significant hardware cost of a TCC system is in the addition of the speculative buffer support to the local cache hierarchy. As a result, it is critical that the amount of state read and/or written by an average transaction be small enough to be buffered on-chip. To get an idea about the size of the state storage) required, Figs. 6 and 7 show the size of the buffers needed to hold the state read or written by 10%, 50%, and 90% of each application's transactions, sorted by the size of the 90% limit.

Virtually all applications have a few very large transactions that will definitely cause overflow, but hardware should have enough room to avoid overflow on *most* transactions in order to keep the number of early commit permission claims to a minimum. 90% or better is a good initial target, but even fewer overflows may be necessary for good performance on systems with many processors.

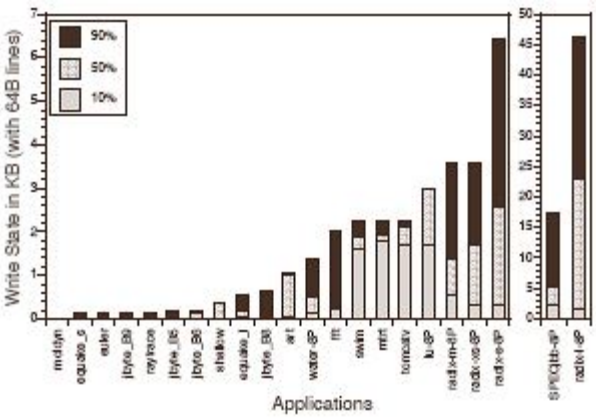

Figure 7: Same as Fig. 6, but for write state.

State size is mostly dependent upon the sizes of "natural" transactional code regions, such as loop bodies, that are available for exploitation within an application. As such, it is very application dependent, but generally quite reasonable. With the exception of **mtrt** and **SPECjbb,** all of our benchmarks worked fine within about 6-12 KB of read state — well within the size of even the smallest caches today — and about 4-8 KB of write state.

The buffer-hungry applications generally still had low 10% and 50% breakpoints, so even those would probably work reasonably well with small buffers, although noticeable serialization from buffer overflow would undoubtedly occur. While our various versions of **radix** did not vary much in terms of speedup, they varied *dramatically* in the size of their read and write state.Our **radix_l** and **radix_xl** (not plotted, because it was so large) variations required very large amounts of state with each transaction. However, it was relatively easy to scale these down to smaller transactions with little impact on the system performance. Based on our examination of the code, many dense-matrix applications such as **swim** and **tomcatv** should have similar properties. Any of these "transaction size tolerant" applications would also be excellent targets for use with hardware commit control, which could help the programmer size transaction regions optimally for the available buffer sizes. This would be especially helpful if widely varying datasets may be used, as transactions that entirely contain inner loops may vary in size along with the dataset.

6.3 *Limited Bus Bandwidth*

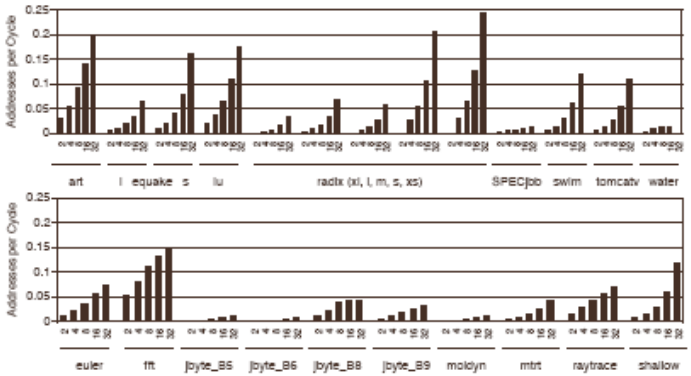

Figure 8: Number of 64 byte cache line addresses broadcast per cycle in our perfect system with 1 IPC, ∞ bus bandwidth, and no cache misses. This indicates essentially the *maximum* snoop rate per avg. processor IPC that could be expected.

Fig. 8 shows the average number of addresses

For our "perfect" sample system, *Fig. 8 shows the average number of addresses* that must be broadcast on every cycle in order to commit all write state produced by all transactions in a system, when the state is stored as 64-byte cache lines. While bus activity in a TCC system is likely to be bursty, the average bandwidths are useful measures because of the ease with which TCC commit packets may be buffered. Because there are no delays in our system for cache misses or communication contention, these should be considered as an upper bound for instruction streams averaging 1 IPC. These numbers can be scaled up to indicate potential maximums for TCC systems composed of wide-issue superscalar cores, or down for simple processors.

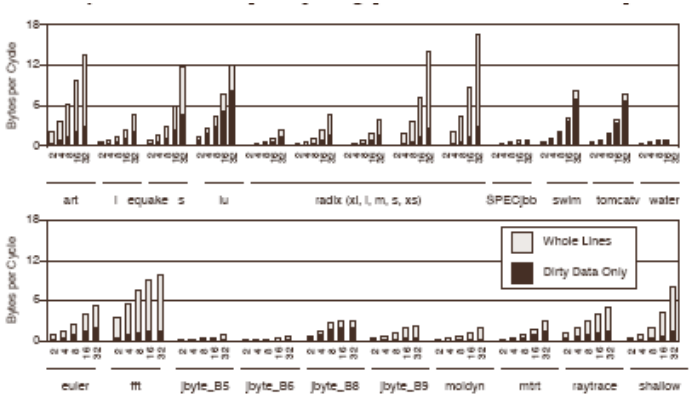

Figure 9: Average bytes per cycle broadcast by a 1 IPC system with infinite bus bandwidth, no cache misses, and TCC with an update protocol. 4 bytes of "address overhead" per 64-byte cache line are assumed.

Fig. 9. the amount of data being broadcast

For all of our applications, the number of addresses per cycle is well below one, so a single snoop port on every processor node should be sufficient for designs of up to 32 processors, and can probably scale up to about 128 simple processors or a smaller number of wide-issue superscalar processors before additional snoop bandwidth would be required. These results also indicate that small TCC systems using an invalidate protocol would usually produce less than about 0.5 bytes/cycle with 32-bit addresses. On the other hand, if an update protocol is used then the amount of data being broadcast may still be prodigious, as is shown in Fig. 9. On some of the applications (about a half of this sample), we may even be broadcasting more data than a processor executing a write through-based cache coherency mechanism, as illustrated in Fig. 10, with a high of nearly 18 bytes per cycle from the versions of **radix** with small transactions (and therefore more frequent commits) for 32 processor systems.

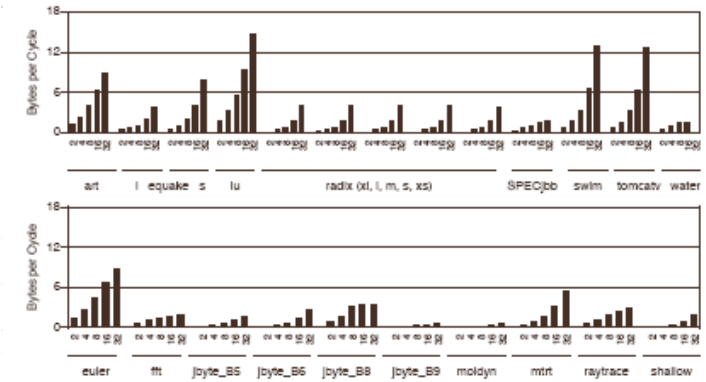

Figure 10: For comparison, average bytes per cycle broadcast on a system using a writethrough-based mechanism (4 address+4 data = 8 bytes/write).

Fig. 10 write through-based mechanism

While TCC allows writes to be combined together into buffered cache lines over the course of a transaction, the committing of extra "clean" sections of partially modified lines in the write state can push up the overall bandwidth requirements dramatically. This problem can be almost completely overcome by modifying the commit broadcast unit to *only* send out modified parts of committing cache lines, limiting bandwidth to just the black part of the bars in Fig. 9 and limiting the amount of broadcast bandwidth required to about 7 bytes per cycle, even on the worst case applications like **lu**, **swim**, and **tomcatv**. For more typical applications, a range of 2–4 bytes per cycle would be sufficient.

6.4 Other Limited Hardware

While the previous runs with "perfect" hardware are helpful for determining if TCC is a viable idea, they do not show how a real TCC system will work in practice, where issues like

finite bus bandwidths, reduced numbers of read state bits, limited buffering, and time to handle the various protocol overheads can all be significant limiting factors on speedup. This section attempts to look at a few of these issues by varying some of the parameters with an 8-processor system.

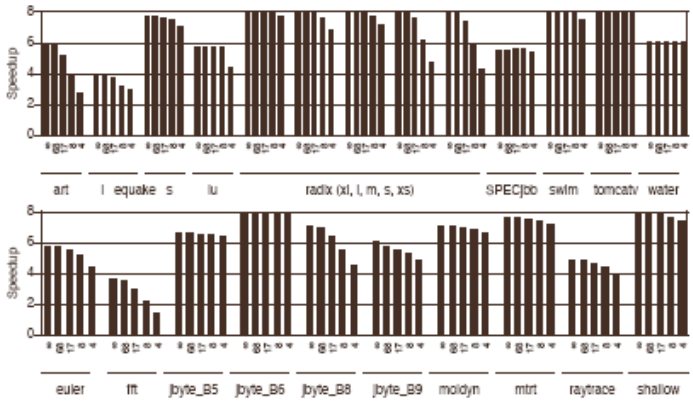

Figure 11: The effect of finite commit bus bandwidth (4, 8, 17, and 68 bytes per cycle) on the speedup of 8 processor systems writing 68-byte (4B header + 64B data) cache lines during commit (or 17, 9, 4, or 1 cycle per committed line).

Fig. 11. full cache line committed per cycle

We simulated finite bus bandwidths ranging from very high (68 bytes/cycle, a full cache line committed per cycle) to levels that would be reasonable in a high-performance CMP or even a potentially board-level system with a high-performance interconnect, and present the results in Fig. 11. Most applications were relatively insensitive to these levels of bandwidth limits, but a few that had a large write state *and* relatively short transactions, notably **fft**, experienced some degradation. Larger numbers of processors or an even more constrained interconnect are necessary for bandwidth to become a major limiting factor for TCC systems.

In addition, we tried making the timing overhead required for commit permission arbitration non-zero, with times ranging from 5 cycles (necessary for arbitration across a large chip) to 200 cycles (which may be necessary on a larger board-size system), and present these results in Fig. 12.

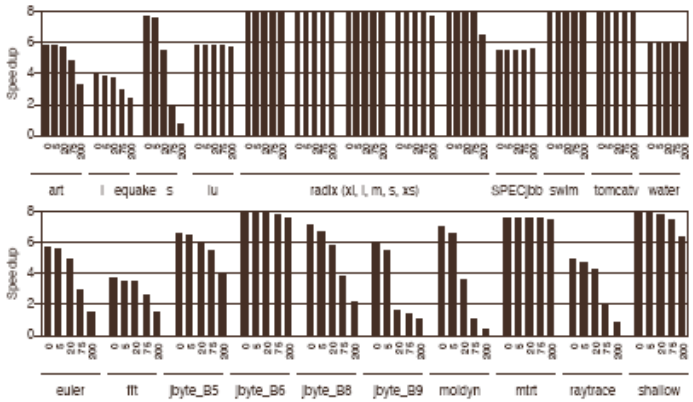

Figure 12: The effect on speedup of increasing commit arbitration time (0, 5, 20, 75, and 200 cycles) for a system with 8 CPUs and a wide, 68 byte bus.

Fig. 12. overhead required for commit arbitration

SPECjbb, SPLASH applications, and compiler-parallelized SPEC FP applications, which have been designed for use on large systems, were almost totally insensitive to this factor. TLS-derived applications, on the other hand, were often quite sensitive, as their transactions tended to be much smaller. Similarly, the versions of **equake** and **radix** that had the smallest transaction sizes showed much more degradation from this overhead than the versions with longer transactions.

We found that optional state proved to be less useful with our selection of applications and hardware parameters. Extra read state bits (on a per-word instead of per-line basis) usually

made no difference, but were essential with a few applications. Most of our manually parallelized applications were carefully tuned to avoid the "false violations," as they were already blocked to avoid false cache sharing, but some of the TLS-parallelized applications, whose data structures had not been modified for parallelism, were dependent upon hardware to avoid extraneous violations. Memory renaming bits were only critical for two of the Java TLS applications, **jbyte_B5** and **mtrt**, as they re-used some "scratchpad" data structures in each transaction. Our analysis also showed little gain from double buffering, surprisingly enough. When we turned it off, not much happened. However, these tests were performed with relatively plentiful system bandwidth. Since double-buffering is primarily a technique to avoid waiting for a busy broadcast medium, it should still prove to be useful in more bandwidth-limited environments.

7 *CONCLUSIONS*

We have analyzed a variety of implementations of TCC systems, including an optimal one, and determined that TCC can be used to obtain good performance over a wide variety of existing parallel application domains, while providing a programming model that significantly simplifies the task of writing parallel programs. Our analysis of TCC with a wide range of applications shows that each processor node requires 6–12 KB of read buffering space in its caches and 4–8 KB of write buffering to achieve high-performance execution on most applications. This buffer memory adds little overhead to the existing cache hierarchy already present within the node. The main limitation of TCC is that it requires high broadcast bandwidth among the processor nodes to maintain all processor's memory in a coherent state. For an 8 processor system, the interprocessor interconnect bandwidth must be large enough to sustain about 2–4 bytes per cycle per average processor IPC to support an update protocol, or usually less then 0.5 bytes per cycle for an invalidate protocol. These rates are easy to sustain within a CMP, and perhaps even a single-board multiprocessor. On these types of systems, we believe that TCC could be a high-performance but much simpler alternative to traditional cache coherence and consistency.

This initial investigation of TCC suggests many potential directions for future work. The most critical is an evaluation of TCC with realistic hardware models for a CMP and/or a board-level system. A detailed evaluation of the TCC programming environment is also a priority, since one of the main advantages of TCC is its simplified parallel programming model. Further out, we see TCC being extended

to be more scalable by imposing levels of hierarchy on the commit arbitration and snoop mechanisms and possibly by allowing some overlap between commits. More functionality may also be added, such as the hardware commit mechanisms, extensions to the data localization, or system reliability mechanisms that use TCC's continuous speculative transactions to roll back the current transaction after transient faults.

We would like to thank the anonymous reviewers for their valuable feedback. 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, Dec. 1996.

[2] S. V. Adve and M. D. Hill, "Weak Ordering: A New Definition," *Proc. of the 17th Annual International Symposium on Computer Architecture*, June 1990.

[3] A. Agarwal, J. L. Hennessy, R. Simoni, and M. A. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988.

[4] A. Ahmed, P. Conway, B. Hughes, F. Weber, "AMD Opteron™ Shared Memory MP Systems," *Conference Record of Hot Chips 14*, Stanford, CA, Aug. 2003

[5] D.Bossen, J. Tendler, K. Reick, "Power4 system design for high reliability," *IEEE MICRO Magazine*, Vol. 22 No. 2 , pp. 16–24, March-April 2002.

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

[7] A. Charlesworth, "Starfire: Extending the SMP Envelope," *IEEE Micro Magazine*, Vol. 18 No. 1, pp. 39-49, Jan.-Feb. 1998.

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

[9] M. Dubois, C. Scheurich, and F. Briggs, "Synchronization, Coherence, and Event Ordering," *IEEE Computer*, February 1988.

[10] M. Franklin and G. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, Vol. 45 No. 5, pp. 552–571, May 1996.

[11] K. D. Gharachorloo, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessey, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th International Symposium on Computer Architecture*, June 1990.

[12] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC," *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 162–171, May 1999.

[13] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 1983.

[14] S. Gopal, T.N. Vijaykumar, J. E. Smith and G. S. Sohi, "Speculative Versioning Cache," *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, Feb. 1998.

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

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

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

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

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

[20] V. Krishnan and J. Torrellas, "A Chip Multiprocessor Architecture with Speculative Multithreading," *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

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

[22] J. P. Lamport, "How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, Vol. 28 No. 9, pp. 690-691, 1979.

[23] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy, "The Stanford DASH Multiprocessor," *Proceedings of the 17th International Symposium on Computer Architecture*, June 1990.

[24] M. Martin, M. Hill, and D. Wood, "Token Coherence: Decoupling Performance and Correctness," *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 182–193, June 2003.

[25] 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)*, October, 2002.

[26] C. McNairy and D. Soltis, "Itanium 2 Processor Microarchitecture," *IEEE MICRO Magazine* , Vol. 23 No. 2, pp. 44–55, March-April 2003.

[27] J. Moreira, S. Midkiff, M. Gupta, and P. Artigas, *Numerically Intensive Java*, IBM at http://www.alphaworks.ibm.com/tech/ninja/, April 1999.

[28] M. Papamarcos and J. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, June 1984.

[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)*, October 2002.

[31] R. Rajwar and J. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, December 2001.

[32] P. Rundberg and P. Stenstrom, "Reordered Speculative Execution of Critical Sections," *Proceedings of the 2002 International Conference on Parallel Processing (ICPP '02)*, Feb. 2002.

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

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

[35] *Stanford Parallel Applications for Shared Memory (SPLASH-2)*, http:// www-flash.stanford.edu/apps/SPLASH/

[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*, Las Vegas, Nevada, 1998.

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

[38] S. Woo, M. Ohara, E. Torrie, J.P.Singh, and A. Gupta, "The SPLASH2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 24–36, June 1995.