Unbounded Transactional Memory

C. Scott Ananian Krste Asanović Bradley C. Kuszmaul Charles E. Leiserson Sean Lie
MIT Computer Science and Artificial Intelligence Laboratory
The Stata Center, 32 Vassar St., Cambridge, MA 02139
{cananian,krste,bradley,cel,sean_lie}@mit.edu

Abstract

Hardware transactional memory should support ***unbounded transactions****: transactions of arbitrary size and duration. We describe a hardware implementation of unbounded transactional memory, called UTM, which exploits the common case for performance without sacrificing correctness on transactions whose footprint can be nearly as large as virtual memory. We performed a cycle-accurate simulation of a simplified architecture, called LTM. LTM is based on UTM but is easier to implement, because it does not change the memory subsystem outside of the processor. LTM allows nearly unbounded transactions, whose footprint is limited only by physical memory size and whose duration by the length of a timeslice.*

We assess UTM and LTM through microbenchmarking and by automatically converting the SPECjvm98 Java benchmarks and the Linux 2.4.19 kernel to use transactions instead of locks. We use both cycle-accurate simulation and instrumentation to understand benchmark behavior. Our studies show that the common case is small transactions that commit, even when contention is high, but that some applications contain very large transactions. For example, although 99.9% *of transactions in the Linux study touch* 54 *cache lines or fewer, some transactions touch over* 8000 *cache lines. Our studies also indicate that hardware support is required, because some applications spend over half their time in critical regions. Finally, they suggest that hardware support for transactions can make Java programs run faster than when run using locks and can increase the concurrency of the Linux kernel by as much as a factor of* 4 *with no additional programming work.*

1. Introduction

Conventionally, atomicity in shared-memory multiprocessors is provided via mutual-exclusion ***locks*** (see, for example, [37]). Although locks are easy to implement using test-and-set, compare-and-swap, or load-linked/store-conditional instructions, they introduce a host of difficulties. To avoid deadlock when locking multiple objects, the locks must be acquired in a consistent linear order, which makes programming with locks error-prone and sometimes introduces significant overheads in managing the lock acquisition protocol. Moreover, locking can introduce other overheads, because a thread must always grab a lock to gain exclusive access to a shared object, regardless of whether another thread is actually attempting to access the same object.

An alternative to locking is ***nonblocking synchronization*** [9, 16, 17, 22, 27], which provides mutual exclusion without using locks. Systems implemented in nonblocking fashion [9, 27] seem to perform better than those that use locks, but conventional nonblocking programming is not only more difficult than programming with locks, it appears to be even more error-prone. A common solution to this programming problem is to encapsulate the nonblocking protocols in library primitives, but this strategy limits the generality with which nonblocking programming can be employed by ordinary programmers.

Transactional memory [14, 15, 20, 32, 33, 36] has been proposed as a general and flexible way to allow programs to read and modify disparate primary memory locations atomically as a single operation, much as a database transaction can atomically modify many records on disk. The basic idea of transactional memory rests on atomic ***transactions*** [8, 25], which offer a method for providing mutual synchronization without the protocol intricacies of conventional synchronization methods. A transaction can be thought of as a sequence of loads and stores performed as part of a program. Unlike in databases, we need not concern ourselves with failures, and so we can arrange that transactions either ***commit*** or ***abort***. If a transaction commits, then all of the loads and stores appear to have run atomically with respect to other transactions, that is, the transaction's operations do not appear to have interleaved with those of other transactions. If a transaction aborts, then none of its stores take effect and the transaction may be restarted, using a backoff or priority mechanism to guarantee forward progress. From the programmer's perspective, all that needs to be specified is where a transaction begins and where it ends, and the transactional support, whether in hardware or software, handles all the complexities.

Hardware transactional memory (HTM) supports atomicity through architectural means, whereas ***software transactional memory*** (STM) supports atomicity through languages, compilers, and libraries. Researchers of both HTM and STM commonly express the opinion that transactions need never touch many memory locations, and hence it is reasonable to put a (small) bound on their size. For HTM implementations, they conclude that a small piece of additional hardware—typically in the form of a fixed-size content-addressable memory and supporting logic—should suffice. For STM implementations, some researchers argue additionally that transactions occur infrequently, and hence the software overhead would be dwarfed by the other processing done by an application.

In contrast, this paper advances the following thesis:

> *Transactional memory should support transactions of arbitrary size and duration. Such support should be provided with hardware assistance, and it should be made visible to the software through the machine's instruction-set architecture (ISA).*

We define a transaction's ***footprint*** to be the set of memory locations accessed by the transaction. We say that a transactional memory system is ***unbounded*** if the system can handle transactions of arbitrary duration and with footprints that are nearly as big as the virtual memory of the system.

The primary goal of unbounded transactional memory is to make concurrent programming easier, hopefully without incurring much overhead in its implementation. We are interested in unbounded transactions because neither programmers nor compilers can easily cope when an architecture imposes a hard limit on transaction size. An implementation might be optimized for transactions below a certain size, but must still operate correctly for larger transactions. The size of transactional hardware should be an implementation parameter, like cache size or memory size, which can vary without affecting the portability of binaries.

Since "ease of programming" is largely subjective, how does one make a case for architectural change, especially since no existing programs use the new programming abstraction? This paper focuses on answering this question. Section 2 describes UTM ("Unbounded Transactional Memory"), a general and flexible architecture to support unbounded transactions. Since the generality of UTM comes at a cost, Section 3 describes LTM ("Large Transactional Memory"), a simplified architecture that we have implemented using the UVSIM [40] processor simulator. LTM handles nearly unbounded transactions with much lower implementation cost. Section 4 describes several microbenchmark and application studies that confirm the assumptions made by UTM and LTM about common-case behaviors of transactions, as well as documenting that mechanisms to support unbounded transactional memory have a minimal impact on processor performance. Section 5 describes related research, and Section 6 provides some concluding remarks.

2. The UTM architecture

This section describes a system called UTM that implements unbounded transactional memory in hardware. UTM allows transactions to grow (nearly) as large as virtual memory. It also supports a semantics for nested transactions, where interior transactions are subsumed into the atomic region represented by the outer transaction. Unlike previous schemes that tie a thread's transactional state to a particular processor and/or cache, UTM maintains bookkeeping information for a transaction in a memory-resident data structure, the ***transaction log***. This enables transactions to survive timeslice interrupts and process migration from one processor to another. We first present the software interface to UTM, and then describe the implementation details.

New instructions

UTM adds two new instructions to a processor's instruction set architecture:

`XBEGIN pc`: Begin a new transaction. The `pc` argument to `XBEGIN` specifies the address of an ***abort handler*** (e.g., using a PC-relative offset). If at any time during the execution of a transaction the hardware determines that the transaction must fail, it immediately rolls back the processor and memory state to what it was when `XBEGIN` was executed, then jumps to `pc` to execute the abort handler.

`XEND`: End the current transaction. If `XEND` completes, then the transaction is committed, and all of its operations appear to be atomic with respect to any other transaction.

Semantically, we can think of an `XBEGIN` instruction as a conditional branch to the abort handler. The `XBEGIN` for a transaction that fails has the behavior of a mispredicted branch. Initially, the processor executes the `XBEGIN` as a not-taken branch, falling through into the body of the transaction. Eventually the processor realizes that the transaction cannot commit, at which point it reverts all processor and memory state back to the point of misprediction and branches to the abort handler.

UTM supports the nesting of transactions by "subsuming" the inner transaction. For example, within an "outer" transaction, a subroutine may be called that contains an "inner" transaction. UTM simply treats the inner transaction as part of the atomic region defined by the outer one. This strategy is correct, because it maintains the property that the inner transaction executes atomically. Subsumed nested transactions are implemented by using a counter to keep track of nesting depth. If the nesting depth is positive, then `XBEGIN` and `XEND` simply increment and decrement the counter, respectively, and perform no other transactional bookkeeping.

Rolling back processor state

The branch mispredict mechanism in conventional superscalar processors can roll back register state only for the small window of recent instructions that have not graduated from the reorder buffer. To circumvent the window-size restriction and allow arbitrary rollback for unbounded transactions, the processor must be modified to retain an additional snapshot of the architectural register state. A UTM processor saves the state of its architectural registers when it graduates an `XBEGIN`. The snapshot is retained either until the transaction aborts, at which point the snapshot is restored into the architectural registers, or until the matching `XEND` graduates indicating that the transaction has committed.

UTM's modifications to the processor core are illustrated in Figure 1. We assume a machine with a unified physical register file, and so rather than saving the architectural registers themselves, UTM saves a snapshot of the register-renaming table and ensures the corresponding physical registers are not reused until the transaction commits. The rename stage maintains an additional "saved" bit for each physical register to indicate which registers are part of the working architectural state, and takes a snapshot as each branch or `XBEGIN` is decoded and renamed. When an `XBEGIN` instruction graduates, activating the transaction, the associated "`S` bit" snapshot will have bits set on exactly those registers holding the graduated architectural state. Physical registers are normally freed on graduation of a later instruction that overwrites the same architectural register. If the `S` bit on the snapshot for the active transaction is set, the physical register is added to a FIFO called a ***Register Reserved List*** instead of the normal ***Register Free List***. This prevents physical registers containing saved data from being overwritten during a transaction. When the transaction's `XEND` commits, the active snapshot's `S` bits are cleared and the Register Reserved List is drained into the regular Register Free List. In the event that the transaction aborts, the saved register-renaming table is restored and the reorder buffer is rolled back, as in an exception. After restoring the architectural register state, the branch is taken to the abort handler. Even though the processor can internally speculatively execute ahead through multiple transactions, transactions only affect the global memory system as instructions graduate, and hence UTM requires only a single snapshot of the architectural register state.

The current transaction abort handler address, nesting depth, and register snapshot are part of the transactional state. They are made visible to the operating system (as additional processor control registers) to allow them to be saved and restored on context switches.

Memory state

Previous HTM systems [15, 20] represent a transaction partly in the processor and partly in the cache, taking advantage of the coincidence between the cache-consistency protocol and the underlying consistency requirements of transactional memory. Unlike those systems, UTM transactions are represented by a single ***xstate*** data structure held in the memory of the system. The cache in UTM is used to gain performance, but the correctness of UTM does not depend on having a cache. In the following paragraphs, we first describe the xstate and how the system uses it assuming there is no caching. Then, we describe how caching accelerates xstate operations.

The xstate is illustrated in Figure 2. The xstate contains a transaction log for each active transaction in the system. A transaction log is allocated by the operating system for each thread, and two processor control registers hold the base and bounds of the currently active thread's log. Each log consists of a ***commit record*** and a vector of ***log entries***. The commit record maintains the transaction's status: `PENDING`, `COMMITTED`, or `ABORTED`. Each log entry corresponds to a block of memory that has been read or written by the transaction. The entry provides a pointer to the block and the old (backup) value for the block so that memory can be restored in case the transaction aborts. Each log entry also contains a pointer to the commit record and pointers that form a linked list of all entries in all transaction logs that refer to the same block.

The final part of the xstate consists of a ***log pointer*** and one ***RW bit*** for each block in memory (and on disk, when paging). If the RW bit is `R`, any transactions that have accessed the block did so with a load; otherwise, if it is `W`, the block may have been the target of a transaction's store. When a processor running a transaction reads or writes a block, the block's log pointer is made to point to a transaction log entry for that block. Further, if the access is a write, the RW bit for the block is set to W. Whenever another processor references a block that is already part of a pending transaction, the system consults the RW bit and log pointer to determine the correct action, for example, to use the old value, to use the new value, or to abort the transaction.

When a processor makes an update as part of a transaction, the new value is stored in memory and the old value is stored in an entry in the transaction log. In principle, there is one log entry for every load or store performed by the transaction. If the memory allocated to the log is not large enough, the transaction aborts and the operating system allocates a larger transaction log and retries the transaction. When operating on the same block more than once in a transaction, the system can avoid writing multiple entries into the transaction log by checking the log pointer to see whether a log entry for the block already exists as part of the running transaction.

By following the log pointer to the log entry, then following the log entry pointer to the commit record, one can determine the transaction status (pending, committed, or aborted) of each block. To commit a transaction, the system simply changes the commit record from PENDING to COMMITTED. At this point, a reference to the block produces the new value stored in memory, albeit after some delay in chasing pointers to discover that the transaction has been committed. To avoid this delay, as well as to free the transaction log for reuse, the system must clean up after committing. It does so by iterating through the log entries, clearing the log pointer for each block mentioned, thereby finalizing the contents of the block. Future references to that block will continue to produce the new value stored in memory, but without the delay of chasing pointers. To abort a transaction, the system changes the commit record from PENDING to ABORTED. To clean up, it iterates through the entries, storing the old value back to memory and then clearing the log pointer. We chose to store the old value of a block in the transaction log and the new value in memory, rather than the reverse, to optimize the case when a transaction commits. No data copying is needed to clean up after a commit, only after an abort.

When two or more pending transactions have accessed a block and at least one of the accesses is a store, the transactions conflict. Conflicts are detected during operations on memory. When a transaction performs a load, the system checks that either the log pointer refers to an entry in the current transaction log, or else that the RW bit is R (additionally creating an entry in the current log for the block if needed). When a transaction performs a store, the system checks that no other transaction is referenced by the log pointer (i.e., that the log pointer is cleared or that the

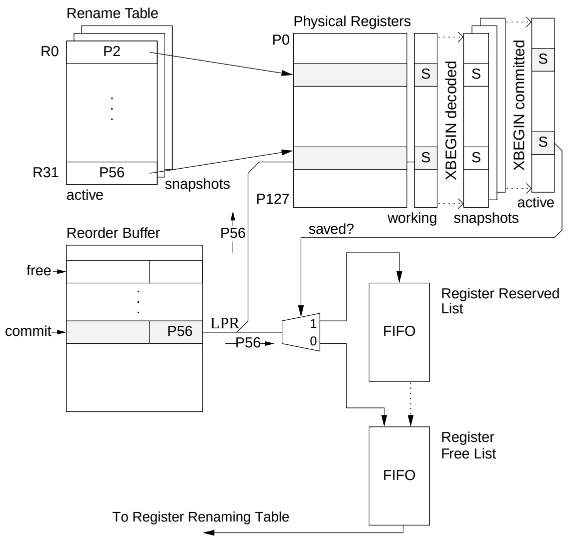

Figure 1: UTM processor modifications. The S bit vector tracks the active physical registers. For each rename table snapshot, there is an associated S bit vector snapshot. The Register Reserved List holds the otherwise free physical registers until the transaction commits. The LPR field is the next physical register to free (the last physical register referenced by the destination architectural register).

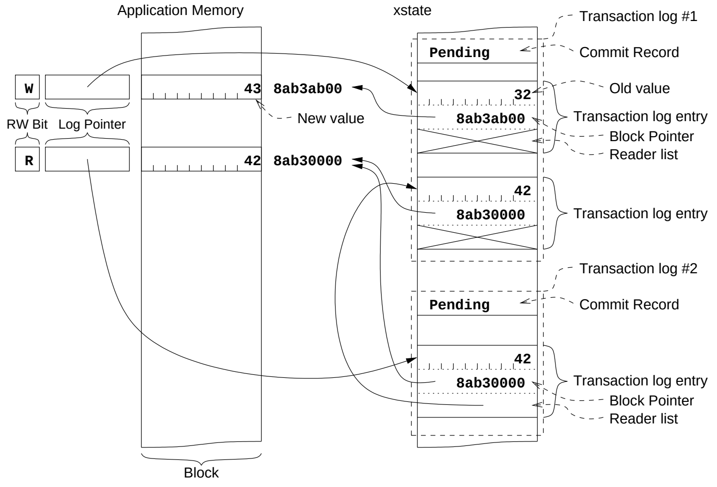

Figure 2: The xstate data structure. The transaction log for a transaction contains a commit record and a vector of log entries. The log pointer of a block in memory points to a log entry, which contains the old value of the block and a pointer to the transaction's commit record. Two transaction logs are shown here; generally, the xstate includes the active transaction logs for the entire system.

linked list of log entries corresponding to this block are all contained in the current transaction log). If the conflict check fails, then some of the conflicting transactions are aborted. To guarantee forward progress, UTM writes a timestamp into the transaction log the first time a transaction is attempted. Then, when choosing which transactions to abort, older transactions take priority. As an alternative, a backoff scheme [28] could also be used.

When a writing transaction wins a conflict, there may be multiple reading transactions that must be aborted. These transactions are found efficiently by following the block's log pointer to an entry and traversing the linked list found there, which enumerates all entries for that block in all transaction logs.

Caching

Although UTM can support transactions of unbounded size using the xstate data structure, multiple memory accesses for each operation may be required. Caching is needed to achieve acceptable performance. In the common case of a transaction that fits in cache, UTM, like the earlier proposed HTM systems [15, 20], monitors the cache-coherence traffic for the transaction's cache lines to determine if another processor is performing a conflicting operation. For example, when a transaction writes to a memory location, the cache protocol obtains exclusive ownership on the whole cache block. New values can be stored in cache with old values left in memory. As long as nothing revokes the ownership of any block, the transaction can succeed. Since the contents of the transaction log are undefined after the transaction commits or aborts, in many cases the system does not even need to write back a transaction log. Thus, for a small transaction that commits without intervention from another transaction, no additional interprocessor communication is required beyond the coherence traffic for the nontransactional case. When the transaction is too big to fit in cache or interactions with other transactions are indicated by the cache protocol, the xstate for the transaction overflows into the ordinary memory hierarchy. Thus, the UTM system does not actually need to create a log entry or update the log pointer for a cached block unless it is evicted. After a transaction commits or aborts, the log entries of unspilled cached blocks can be discarded and the log pointer of each such block can be marked clean to avoid writeback traffic for the log pointer, which is no longer needed. Most of the overhead is borne in the uncommon case, allowing the common case to run fast.

The in-cache representation of transactional state and the xstate data structure stored in memory need not match. The system can optimize the on-processor representation as long as, at the cache interface, the view of the xstate is properly maintained. For convenience, the transaction block size can match the cache line size.

System issues

The goal of UTM is to support transactions that can run for an indefinite length of time (surviving time slice interrupts), can migrate from one processor to another along with the rest of a process's state, and can have footprints bigger than the physical memory. Several system issues must be solved for UTM to achieve that goal. The main technique that we propose is to treat the xstate as a system-wide data structure that uses global virtual addresses.

Treating the xstate as data structure directly solves part of the problem. For a transaction to run for an indefinite length of time, it must be able to survive a time-slice interrupt. By adding the log pointer to the processor state and storing everything else in a data structure, it is easy to suspend a transaction and run another thread with its own transaction. Similarly, transactions can be migrated from one processor to another. The log pointer is simply part of the thread or process state provided by the operating system.

UTM can support transactions that are even larger than physical memory. The only limitation is how much virtual memory is available to store both old and new values. To page the xstate out of main memory, the UTM data structures might employ global virtual addresses for their pointers. Global virtual addresses are system-wide unique addresses that remain valid even if the referenced pages are paged out to disk and reloaded in another location. Typically, systems that provide global virtual addresses provide an additional level of address translation, compared to ordinary virtual memory systems. Hardware first translates a process's virtual address into a global virtual address. The global virtual address is then translated into a physical address. Multics [2] provided user-level global virtual addressing using segment-offset pairs as the addresses. The HP Precision Architecture [23] supports global virtual addresses in a 64-bit RISC processor.

The log pointer and state bits for each user memory block, while typically not visible to a user-level programmer, are themselves stored in addressable physical memory to allow the operating system to page this information to disk. The location of the memory holding the log pointer information for a given user data page is kept in the page table and cached in the TLB.

During execution of a single load or store instruction, the processor can potentially touch a large number of disparate memory locations in the xstate, any of which may be paged out to disk. To ensure forward progress, either the system must allow load or store instructions to be restarted in the middle of the xstate traversal, or, if only precise interrupts are allowed, the operating system must ensure that all pages required by an xstate traversal can be resident simultaneously to allow the load or store to complete without page faults.

UTM assumes that each transaction is a serial instruction stream beginning with an `XBEGIN` instruction, ending with a `XEND` instruction, and containing only register, memory, and branch instructions in between. A fault occurs if an I/O instruction is executed during a transaction.

3. The LTM implementation

UTM is an idealized design for HTM that requires significant changes to both the processor and the memory subsystem of a current computer architecture. By scaling back on the degree of "unboundedness," however, a compromise between programmability and practicality can be achieved. This section presents such an architectural compromise, called LTM, for which we have implemented a detailed cycle-level simulation using UVSIM [40].

LTM's design is easier to implement than UTM, because it does not support transactions of virtual-memory size. Instead, LTM avoids the intricacies of virtual memory by limiting the footprint of a transaction to (nearly) the size of physical memory. In addition, the duration of a transaction must be less than a time slice and transactions cannot migrate between processors. With these restrictions, LTM can be implemented by only modifying the cache and processor core and without making changes to the main memory, the cache-coherence protocols, or even the contents of the cache-coherence messages. Unlike a UTM processor, an LTM processor can be pin-compatible with a conventional processor. The design presented here is based on the SGI Origin 3000 shared-memory multiprocessor, with memory distributed among the processor nodes and cache coherency maintained using a directory-based write-invalidate protocol.

The UTM and LTM schemes share many ideas. Like UTM, LTM maintains data about pending transactions in the cache and detects conflicts using the cache-coherency protocol in much the same way as previous HTM proposals [18, 20]. LTM also employs an architectural state-save mechanism in hardware. Unlike UTM, LTM does not treat the transaction as a data structure. Instead, it binds a transaction to a particular cache. Transactional data overflows from the cache into a hash table in main memory, which allows LTM to handle transactions too big to fit in the cache without the full implementation complexity of the xstate data structure.

LTM has similar semantics to UTM, and the format and behavior of the `XBEGIN` and `XEND` instructions are the same. The information that UTM keeps in the transaction log is kept partly in the processor, partly in the cache, and partly in an area of physical memory allocated by the operating system.

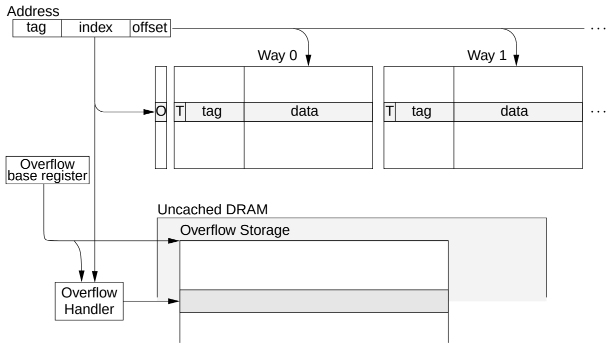

Figure 3: LTM cache modifications. The T bit indicates if the line is transactional. The O bit indicates if the set has overflowed. Overflowed data is stored in a data structure in uncached DRAM.

LTM requires only a few small modifications to the cache, as shown in Figure 3. For small transactions, the cache is used to store the speculative transactional state. For large transactions, transactional state is spilled into an overflow data structure in main memory. An additional bit (T) is added per cache line to indicate if the data has been accessed as part of a pending transaction. When a transactional-memory request hits a cache line, the T bit is set. An additional bit (O) is added per cache set to indicate if it has overflowed. When a transactional cache line is evicted from the cache for capacity reasons, the O bit is set.

In LTM, the main memory always contains the original state of any data being modified transactionally, and all speculative transactional state is stored in the cache and overflow hash table. A transaction is committed by simply clearing all the T bits in cache and writing all overflowed data back to memory. Conflicts are detected using the cache-coherency protocol. When an incoming cache intervention hits a transactional cache line, the running transaction is aborted by simply clearing all the T bits and invalidating all modified transactional cache lines.

The overflow hash table in uncached main memory is maintained by hardware, but its location and size are set up by the operating system. If a request from the processor or a cache intervention misses on the resident tags of an overflowed set, the overflow hash table is searched for the requested line. If the requested cache line is found, it is swapped with a line in the cache set and handled like a hit. If the line is not found, it is handled like a miss. While handling overflows, all incoming cache interventions are stalled using a NACK-based network protocol.

The LTM overflow data structure uses the low-order bits of the address as the hash index and uses linear probing to resolve conflicts. When the overflow data structure is full, the hardware signals an exception so that the operating system can increase the size of the hash table and retry the transaction.

LTM was designed to be a first step towards a truly unbounded transactional memory system such as UTM. LTM has most of the advantages of UTM while being much easier to implement: one student was able to implement LTM in a detailed simulator in one semester and obtain results that provide insight into the behavior of unbounded transactions. LTM is interesting its own right, perhaps providing a more practical implementation of quasi-unbounded transactional memory that suffices for real-world concerns. For its part, the idealized UTM scheme stakes out the extreme of unbounded transactional support, providing a good measuring stick against which LTM and other potential compromises can be evaluated.

4. Evaluation

In this section, we evaluate the efficacy of our mechanisms for unbounded transactional memory. Using both cycle-accurate simulation and trace analysis of application programs, including the SPECjvm98 Java benchmarks and the Linux 2.4.19 kernel, we conclude that the common case is indeed small transactions that commit, even when contention is high, but that some applications contain very large transactions. Hardware support is needed, because some applications spend over half their time in critical regions. Moreover, the data suggest that transactions increase concurrency compared with locking.

Transactional applications

Because no large-scale applications currently exist that use transactional memory, we developed translation tools to convert C and Java programs that use locks into transactional programs. Using this methodology we converted the Linux 2.4.19 kernel, written in C and running under User-Mode Linux [4], and the SPECjvm98 benchmarks [35], written in Java, to use transactions. Although this methodology produces applications that retain some of the vestiges of locking (such as protocols to avoid deadlock), it provides conservative numerical results for estimating whether the assumptions of the UTM architecture are valid. Moreover, it allowed us to measure a full operating system and real Java programs. We also developed some synthetic microbenchmarks, one of which is reported here. Other results are presented in [24].

We compiled the SPECjvm98 benchmark suite with FLEX, our state-of-the-art Java compiler infrastructure [1] using version 0.06 of the GNU Classpath [6] Java standard libraries and the "PreciseC" FLEX backend. FLEX translates standard Java `synchronized` blocks into `atomic` blocks [5, 11]. Although the semantics of the program change slightly with this transformation, the effect tends to be consistent with the programmer's original intent [5]. We transformed standard Java monitor synchronization into transactions, and nested locks were transformed into nested subsumed transactions. Although the SPECjvm98 benchmarks are largely single-threaded, since they use the thread-safe Java standard libraries, they contain synchronized code. We gather information about these transformed synchronized regions from single-threaded executions of the benchmarks. Some SPECjvm98 benchmarks were omitted from our investigations. The `227_mtrt` and `205_raytrace` benchmarks were omitted from the execution-driven experiments due to thread-system incompatibilities with the simulator infrastructure. The `228_jack` parser generator was omitted from the trace-driven study, because the instrumented version tickled a bug in the Irix assembler that we could not fix. The methodology of the trace-driven study also excluded the `227_mtrt` benchmark, but as `227_mtrt` is identical to `205_raytrace` except for a command-line argument specifying the number of threads, we expect `227_mtrt` to have transaction properties similar to `205_raytrace`.

We evaluated the Linux 2.4.19 kernel using User-Mode Linux [4], which runs Linux as a set of processes on a host operating system. We configured User-Mode Linux to use two processes to emulate two processors, and ran it on a 2-processor SMP host machine. Since the locks in Linux are properly nested and are never held across context switches or input/output, we could easily "transactify" the kernel by using transactions instead of locks. Our source-to-source translator simply replaced all locks with transactions, subsuming nested transactions whenever nested locks were encountered. We began our traces after initial system boot and setup. We studied two workloads: a parallel `make` that compiles the kernel, and `dbench` running three clients [38].

Our execution-driven experiments used UVSIM [40], a multiprocessor simulator based on RSIM [30]. The cycle-accurate processor model is based on a MIPS R10K 4-issue out-of-order superscalar processor [39], extended with 96 physical registers and the additional register and cache support for LTM. We model a 2GHz CPU, 32KB 4-way associative instruction and data L1 caches with 64-byte cache lines, a 1MB 4-way unified L2 cache with 128-byte cache lines, and a 400Mb/s DDR2 SDRAM memory system. The system has a distributed directory-based cache coherence protocol based on the SGI Origin, with a 10 cycle-per-hop interprocessor network la-

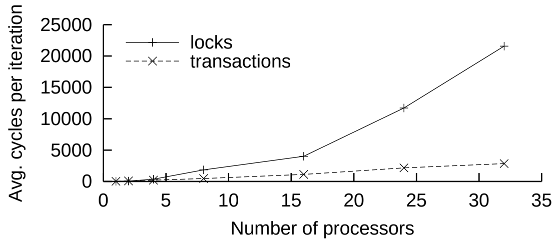

Figure 4: `Counter` performance on UVSIM.

Benchmark application	Base time (cycles)	Locks time (% of Base time)	Trans time (% of Base time)	Time in trans (% of Trans time)	Time in overflow (% of Trans time)
`200_check`	8.1M	124%	101%	32.5%	0.0085%
`201_compress`	608.3M	103%	106%	3.9%	3.9%
`202_jess`	75.0M	141%	108%	59.4%	0.0072%
`209_db`	11.8M	142%	105%	54.0%	0%
`213_javac`	30.7M	170%	114%	84.2%	10%
`222_mpegaudio`	99.0M	100%	100%	0.8%	0%
`228_jack`	261.4M	175%	104%	32.1%	0.0056%

Figure 5: SPECjvm98 performance on a 1-processor UVSIM simulation. The "Time in trans" and "Time in overflow" are the times spent actually running a transaction and handling overflows, respectively. The input size is `-s1`. For our tests, each UVSIM processor is configured with a 1MB 4-way set-associative L2 cache using 128-byte cache lines.

tency [24]. To run on UVSIM, the microbenchmarks and the SPECjvm98 benchmarks were compiled into MIPS Irix binaries with instruction extensions for transactions.

Microbenchmark on LTM

The microbenchmark `Counter` was designed to provide insight into the behavior of locks and LTM under high contention. It shows that small transactions are likely to commit even when contention is high. The microbenchmark has one shared variable which each processor increments repeatedly in a critical section. Thus, each transaction is only a few instructions long. If an access to the variable fails, the abort handler simply retries immediately without backing off. In the locking version, each processor obtains a global spin-lock using a load-linked/store-conditional (LLSC) sequence.

Both the locking and transactional versions of `Counter` were run on UVSIM with LTM, and the results are shown in Figure 4. The locking version scales poorly, because the LLSC causes many cache interventions even when the lock cannot be obtained. On the other hand, the transactional version scales much better, despite having no back-off. When a transaction obtains a cache line, it is likely to be able to execute a few more instructions before receiving an intervention, since the network latency is high. Therefore, small transactions can start and commit (and perhaps even start and commit subsequent transactions) before the cache line is taken away. Similar behavior is expected from UTM, because small transactions effectively use the cache the same way. This benchmark indicates that small transactions that commit is indeed the common case, even under high contention. Moreover, it confirms other researchers' findings [10, 31, 32] that transactions can execute with lower overhead than locks.

SPECjvm98 on LTM

We ran the Java benchmarks on LTM, and the results are shown in Figure 5. For these measurements, we used our modified Java compiler to compile three versions of the SPECjvm98 benchmark suite to run under our UVSIM implementation of LTM. We compiled a ***Base*** version that uses no synchronization, a ***Locks*** version that uses spin-locks for synchronization, and a ***Trans*** version that uses LTM transactions for synchronization. To measure overheads, we ran these versions of the SPECjvm98 benchmark suite on one processor of UVSIM.

As shown in the figure, the overhead of adding transactions to the base code is typically under 10%. Locking adds much more overhead, and the impact on performance is more variable. One reason that the locking versions of some applications (e.g., `javac`) are so much slower than the transactional verions is that the subsumption of nested transactions enables multiple lock operations on the various objects involved in a critical region to be replaced with a single outer transaction. Many of the benchmarks spend much of their time in transactions, which indicates a need for hardware support for transactional memory, at least for these kinds of legacy benchmarks. The benchmarks also indicate that the LTM hardware spends little time handling overflows, but large transactions that cause overflow do occur.

These results also predict that the UTM architecture should also have minimal overhead, since the UTM data structure behaves much like the LTM hash table. For a miss in the cache, LTM requires one additional memory access to index the hash table (when there is no hash-table conflict). Similarly, UTM requires one memory access in addition to retrieving the requested cache line. UTM may require an additional memory access, however, to retrieve the transactional record entry. Since UTM requires at most one more memory access than LTM when there is overflow, and overflow is not common, the performance of UTM should be similar to that of LTM.

Trace-driven studies

We also instrumented the SPECjvm98 benchmarks and the Linux 2.4.19 kernel to produce a trace of the memory references and transactional operations. For the Linux kernel we used a source-to-source translator for C programs [29] to instrument every load and store, and we modified the kernel headers to instrument the locks. For the SPECjvm98 benchmarks, we modified a Java compiler [1] to instrument load, store, and synchronization operations. In both cases, we ran the instrumented program to get a trace and then ran the trace through a simulator we developed to measure the transaction properties. The simulation used a 1MB 4-way set-associative cache with 64-byte cache lines.

Figure 6 shows the the results of our trace analysis of the Linux 2.4.19 kernel and SPECjvm98 benchmarks. For both kernel workloads, `make_linux` and `dbench`, the "Xops %" column shows that over 40% of all the kernel's memory operations take place in transactions, which means that a software transactional memory would be too slow. Many of the SPECjvm98 benchmarks exhibit similar numbers. The "Oversized xaction," "Biggest xaction," and "Overflow %" columns show that some applications contain transactions whose footprints would overflow any reasonably-sized cache. But these big transactions don't cost much, as can be seen in a variety of ways—for example, "Cache miss %" is typically far greater than "Xmiss %."

The Java benchmarks have disparate numbers and sizes of transactions. One extreme is the `213_javac` Java compiler benchmark, which contains a small number of extremely large transactions, one of which has a footprint of over a million cache lines. Closer examination reveals that the method `Javac.compile()`, which implements the entire compilation process, is marked as synchronized: the programmer has explicitly requested that the entire compilation occur atomically. Although one could argue that the Java compiler should be rewritten if concurrency is desired, even difficult cases, such as the Java compiler, should at least work correctly. The variability again demonstrates that bounded transactions are insufficient when automatic tools transform lock-based protocols into transactions.

We studied the `make_linux` and `dbench` kernel benchmarks more closely to understand how the size of cache affects the overflow of transactional state in UTM and LTM. Figure 7 graphs the results, confirming that there are again some very large transactions, but that most transactions are small. For these benchmarks, almost all the transactions need less than about 100 cache lines, and in fact, 99.9% need fewer than 54 cache lines.

The UVSIM simulations for `Counter` and the SPECjvm98 benchmarks indicate that switching from locks to transactions enhances concurrency. To study this phenomenon more deeply, we instrumented the Linux benchmarks to measure ***lock contention***, the probability that a particular lock is held at any time during the locking execution, and ***cache contention***, the probability that a particular cache line is held in the write state by a transaction during the transactional execution. The idea is that, under locking, there is some "hottest" lock that prevents the kernel from running on more than a certain number of processors. Similarly, under transactions, there is a "hottest" cache line, because the transaction mechanism implicitly locks cache lines. If locks are the only limits to concurrency, we can use the hottest location to provide an upper bound to the concurrency of an application. For example, if a particular lock is held 5% of the time by each processor, then the application can not use more than 20 processors before the lock becomes a bottleneck. This measure of concurrency provides insight into the available concurrency of the locking and transactional versions of the kernel.

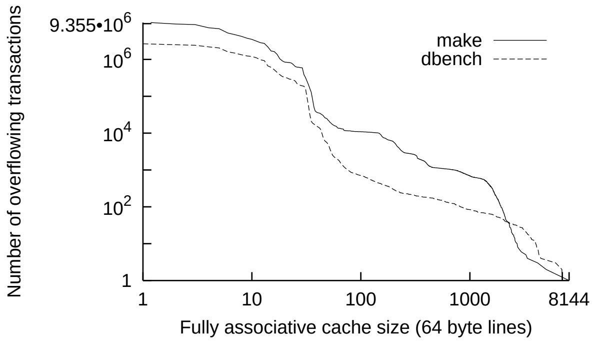

Figure 7: Fully associative cache-size requirements for `make_linux` and `dbench`. Both axes are log-log.

For `make_linux`, the hottest lock (the kernel lock) was held about 4 times longer than the hottest cache line, corroborating the findings of our cycle-accurate simulation and the literature that transactions increase concurrency. Reducing the dependence on the kernel lock has been the focus of years of effort by the kernel developers, but progress has been slow. Inspection of the code that manipulates the hottest cache line reveals that it contains counters for the page allocator, and that minor data restructuring using transactional memory would yield a 25% improvement in concurrency. This optimization is not easily available to the kernel programmers, however, because it would be hard to obey the locking protocol which dictates the order in which locks must be acquired. Thus, the primary claim of transactional memory—that it makes concurrent programming easier—appears to be valid both because of greater concurrency compared with locks and the ease with which concurrency can be enhanced.

Summary

Our Linux and Java studies strongly suggest that the

Program	Input size	Total memory ops	cache miss %	Xactions	Oversized xactions	Xops %	Xmiss %	Overflow %	Biggest xaction
make_linux		315,776,028	0.56%	6,964,277	3368	41.0%	0.017%		8144
dbench		100,928,220	0.43%	1,863,426	88	49.5%	0.001%		7047
201_compress	1	229,332,212	0.10%	524	0	0.0%	0	0	54
	100	2,981,777,890	0.10%	2,272	0	0.0%	0	0	52
202_jess	1	1,972,479	3.13%	82,103	0	43.3%	0	0	428
	100	405,153,255	2.71%	4,892,829	0	9.1%	0	0	1,064
205_raytrace	1	14,535,905	1.83%	1,125	1	49.6%	0.648%	0.0889%	110,579
	100	420,005,763	1.65%	4,177	1	1.7%	0.022%	0.0239%	110,509
209_db	1	393,455	2.01%	14,191	0	45.8%	0	0	187
	100	848,082,597	10.14%	45,222,742	288	23.0%	0.350%	0.0005%	67,569
213_javac	1	1,605,330	1.88%	460	1	89.5%	0.517%	0.2087%	24,559
	100	472,416,129	1.78%	668	4	99.9%	1.652%	0.5988%	1,275,590
222_mpegaudio	1	26,551,440	0.03%	1,049	0	0.1%	0	0	53
	100	2,620,818,214	0.00%	2,992	0	0.0%	0	0	54

Figure 6: Experimental results for transactifying the Linux kernel and Java. For Java we show results for runs with 1% and 100% of the full input size. For the percentages, we write "0" for numbers that are exactly zero, and a "zero" percentage, such as "0.0%," for small nonzero values. The columns are as follows: ***Total memory ops*** is the total number of loads and stores executed. For Linux, we measured the kernel's memory operations. For Java, we measured the application's memory operations, not including operations performed by native methods and gc. ***Cache miss %*** is the fraction of memory operations that caused cache misses. ***Xactions*** is the total number of transactions. ***Oversized xactions*** is the total number of transactions that did not fit entirely within the cache. ***Xops %*** is the fraction of memory operations that were in transactions. ***Xmiss %*** is the fraction of transactional loads and stores that did not fit into the cache, and hence invoked the overflow mechanism. ***Overflow %*** is the fraction of the cache sets that overflowed. We measured this at the end of each transaction, and averaged it over all the transactions. This gives a rough measure of the likelihood that a cache intervention request from another processor would need to look at the overflow buffer. ***Biggest xaction*** is the largest number of distinct cache lines that any transaction touched. A fully associative bounded-hardware transaction scheme would need a cache of at least this size.

assumptions behind the UTM and LTM architectures are correct: transactions are frequent and require hardware support, most transactions fit in the cache, and a few large transactions must be handled by exceptional mechanisms supporting unbounded transaction sizes. The concurrency results suggest that automatic translation of locks to transactions is viable and desirable for legacy code. In addition, it appears that transactional memory is not limited to specialized parallel applications, but can be exploited by ordinary Java and C programs. Indeed, our studies show that transactional memory can be exploited by operating systems, perhaps the most frequently run multithreaded programs.

5. Related Work

This section describes related research in the literature.

Transactions are described in the database context by Gray [7], and [8] contains a thorough treatment of database issues. HTM was first proposed by Knight [20], and Herlihy and Moss coined the term "transactional memory" and proposed HTM in the context of lock-free data structures [15, 18]. The BBN Pluribus [34, Ch. 23] provided transactions, with an architectural limit on the size of a transaction. Experience with Pluribus showed that the headaches of programming correctly with such limits can be at least as challenging as using locks. The ***Oklahoma Update*** is another variation on transactional operations with an architectural limit on the number of values in a transaction [36].

Transactional memory is sometimes described as an extension of Load-Linked/Store-Conditional [19] and other atomic instruction sequences. In fact, some CISC machines, such as the VAX, had complex atomic instructions such as enqueue and dequeue [3]. Lamport proposed lock-free data structures [22], and Herlihy proposed wait-free programming [12, 13].

Of particular relevance to our work are ***Speculative Lock Elision*** (SLE) [31] and ***Transactional Lock Removal*** (TLR) [32], which speculatively identify locks and use the cache to give the appearance of atomicity. SLE and TLR handle mutual exclusion through a standard programmer interface (locks), dynamically translating locks into transactional regions. Our research thrust differs from theirs in that we hope to free programmers from the protocol complexities of locking, not just optimize existing practice. Despite the difference in outlook, however, our quantitative results from Section 4 confirm their finding that transactional hardware can be more efficient than locks.

Martinez and Torrellas proposed ***Speculative Synchronization*** [26], which allows some threads to execute atomic regions of code speculatively, using locks, while guaranteeing forward progress by maintaining a nonspeculative thread. These techniques gain many of the performance advantages of transactional memory, but they still require new code to obey a locking protocol to avoid deadlock. We believe programmers would enthusiastically abandon the locking paradigm if a transactional memory implementation of atomic regions was available and had good performance.

The recent work on ***Transactional memory Coherence and Consistency*** (TCC) [10] is also relevant to our work. TCC uses a broadcast bus to implement the transaction protocols, performing all the writes of a particular transaction in one atomic bus operation. This strategy limits scalability, whereas both UTM and LTM can employ scalable cache-consistency protocols to implement transactions. One important conclusion is the same for both TCC and our work: most transactions are small, but some are very large. TCC supports large transactions by locking the broadcast bus and stalling all other processors when any processor buffer overflows, whereas UTM and LTM allow overlapped execution of multiple large transactions with local overflow buffers. TCC is similar to LTM in that transactions are bound to processor state and cannot extend across page faults, timer interrupts, or thread migrations.

Many researchers have pursued software transactional memory (STM) [11, 14, 33]. Some constrain the programmer and make transactions difficult to use by, for example, requiring all object memory touched by a transaction to either be known in advance [33] or explicitly "opened" during the transaction [14]. Harris and Fraser built a software transaction system on a flat word-oriented transactional memory abstraction [11], roughly similar to simulating Herlihy's original hardware transactional memory proposal in software. Their data structure bears some resemblance to the in-memory data kept by our UTM scheme. Unfortunately, software transactional memory appears to be too slow to support legacy code automatically.

Herlihy and Moss [15] suggested that small transactions might be handled in cache with overflows handled by software. These software overflows must interact with the transactional hardware in the same way that the hardware interacts with itself, however. This approach may work, but designing a safe software interface to the consistency protocol on which hardware transactions depends seems to be a difficult open problem for current architectures.

6. Conclusion

This paper has made a case that transactional memory systems should support unbounded transactions in hardware. UTM represents a radical, but fully scalable point in the design space, whereas LTM represents a buildable alternative that achieves many of the same advantages. Undoubtedly, other engineering tradeoffs could be made. In addition, many more fundamental questions remain about how to design and use transactional memory.

Our designs have prohibited the use of I/O operations during a transaction. Is there a way for transactional memory to support mutual exclusion while performing I/O? Certain inherently serial I/O operations seem to require the use of mutual exclusion. Newer devices tend to provide "multithreaded" interfaces which allow bundling of nonmodal commands whose execution can be initiated with a single atomic I/O operation. It may be that our hardware memory commit mechanism can be made atomic with the I/O commit to allow integration of transactions and I/O.

UTM and LTM sequence transactions within each thread, but provide no mechanism to impose a particular ordering of transactions across threads. It is unclear if additional support is desirable or whether barriers and other conventional inter-thread ordering techniques built using transactional primitives suffice.

Another open question is whether an HTM system can implement a more optimistic concurrency control [21], instead of implementing a pessimistic concurrency control system as we have presented here. Various optimistic concurrency protocols are now widely deployed in database systems, suggesting that the benefits from the protocols' increased concurrency often outweigh their implementation complexity.

Unbounded transactions potentially represent a big step toward making parallel computing practical and ubiquitous. They promise to simplify or eliminate many of the problems with coordination and synchronization that programmers now face when dealing with concurrency. Transactions should make it far easier for all programmers, not just those who are specialists in today's arcane practices of parallel computing, to write correct high-performance multithreaded programs. We hope that unbounded transactional memory may eventually become, like cache or virtual memory, an expected subsystem of any computer architecture.

Acknowledgments

Marty Deneroff, formerly of SGI, contributed several ideas to our HTM design, including the idea of using a special overflow area and marking cache sets with an overflow bit.

Kunal Agrawal, Jeremy Fineman, Dana S. Henry, Zardosht Kasheff, I-Ting (Angelina) Lee, Sid Sen, and Jim Sukha provided helpful suggestions on the writing.

Martin Rinard advised the Java transaction implementation.

This research was supported in part by a DARPA HPCS grant with SGI, DARPA/AFRL Contract F33615-00-C-1692, NSF Grants ACI-0324974 and CNS-0305606, NSF CAREER grant CCR00093354, and the Singapore-MIT Alliance.

References

[1] C. S. Ananian. The FLEX compiler project. `http://flex-compiler.csail.mit.edu/`.

[2] A. Bensoussan, C. Clingen, and R. Daley. The Multics virtual memory: Concepts and design. *CACM*, 15(5):308–318, May 1972.

[3] Digital Equipment Corporation. *VAX MACRO and Instruction Set Reference Manual*, Nov. 1996.

[4] J. Dike. A user-mode port of the Linux kernel. In *The 4th Annual Linux Showcase and Conf.*, Oct. 10-14 2000.

[5] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '98*, pp. 338–349, June 1998.

[6] The GNU Classpath project. `http://classpath.org/`.

[7] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, pp. 144–154, Sept. 1981.

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

[9] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *USENIX OSDI '96*, pp. 123–136, Oct. 1996.

[10] L. Hammond, V. Wong, et al. Transactional memory coherence and consistency. In *ISCA 31*, pp. 102–113, June 2004.

[11] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pp. 388–402, Oct. 2003.

[12] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, Jan. 1991.

[13] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM TOPLAS*, 15(5):745–770, Nov. 1993.

[14] M. Herlihy, V. Luchangco, et al. Software transactional memory for dynamic-sized data structures. In *PODC '03*, pp. 92–101, July 2003.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA 20*, pp. 289–300, May 1993.

[16] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88*, pp. 276–290, Aug. 1988.

[17] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pp. 522–529, May 2003.

[18] M. P. Herlihy and J. E. B. Moss. Transactional support for lock-free data structures. Tech. Rep. 92/07, Digital Cambridge Research Lab, Dec. 1992.

[19] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Tech. Rep. UCRL-97663, LLNL, Nov. 1987.

[20] T. Knight. An architecture for mostly functional languages. In *LFP*, pp. 105–112. ACM Press, 1986.

[21] H. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, June 1981.

[22] L. Lamport. Concurrent reading and writing. *CACM*, 20(11):806–811, Nov. 1977.

[23] R. B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, Jan. 1989.

[24] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, MIT, May 2004.

[25] N. Lynch, M. Merritt, et al. *Atomic Transactions*. Morgan Kaufmann, San Mateo, CA, 1994.

[26] J. F. Martinez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X*, pp. 18–29, Oct. 5–9 2002.

[27] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Tech. Rep. CUCS-005-91, Columbia Univ., June 1991.

[28] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *CACM*, 19(7):395–404, July 1976.

[29] R. C. Miller. A type-checking preprocessor for Cilk 2, a multithreaded C language. Master's thesis, MIT, May 1995.

[30] V. Pai, P. Ranganathan, and S. Adve. RSIM reference manual, version 1.0. Tech. Rep. 9705, Rice Univ., Aug. 1997.

[31] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO-34*, pp. 294–305, Dec. 2001.

[32] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X*, pp. 5–17, Oct. 5–9 2002.

[33] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95*, pp. 204–213, Aug. 1995.

[34] D. Siewiorek, G. Bell, and A. Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, 1982.

[35] SPEC JVM Client 98 (SPECjvm98). `http://www.spec.org/jvm98/`, 1998.

[36] J. M. Stone, H. S. Stone, et al. Multiple reservations and the oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, Nov. 1993.

[37] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[38] A. Tridgell. Emulating Netbench. `http://samba.org/ftp/tridge/dbench/README`, 1999.

[39] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[40] L. Zhang. UVSIM reference manual. Tech. Rep. UUCS-03-011, Univ. of Utah, Mar. 2003.