UNBOUNDED TRANSACTIONAL MEMORY

TRANSACTIONAL MEMORY SHOULD BE VIRTUALIZED TO SUPPORT TRANSACTIONS OF ARBITRARY FOOTPRINT AND DURATION. UNBOUNDED TRANSACTIONAL-MEMORY ARCHITECTURES CAN ACHIEVE HIGH PERFORMANCE IN THE COMMON CASE OF SMALL TRANSACTIONS, WITHOUT SACRIFICING CORRECTNESS IN LARGE TRANSACTIONS.

C. Scott Ananian
Krste Asanović
Bradley C. Kuszmaul
Charles E. Leiserson
Massachusetts Institute of Technology

Sean Lie
Advanced Micro Devices

Researchers have proposed using transactional memory as a flexible method by which programs can read and modify disparate primary memory locations atomically in a single operation, much as a database transaction can atomically modify many records on disk.[1-6] The basis of transactional memory is atomic transactions, which offer a method of providing mutual synchronization without the protocol intricacies of conventional synchronization methods such as locks. We can think of a transaction as a sequence of loads and stores performed as part of a program. With transactional memory, unlike databases, we need not concern ourselves with failures, so we can arrange that transactions either commit or abort. If a transaction commits, all the loads and stores appear to have run atomically with respect to other transactions; that is, the transaction's operations don't appear to have interleaved with those of other transactions. If a transaction aborts, none of its stores take effect and the transaction can restart, using a back-off or priority mechanism to guarantee forward progress. All the programmer must specify is where a transaction begins and ends; 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. Both HTM and STM researchers say that transactions need never touch many memory locations, and hence it is reasonable to put a (small) bound on their *footprint*, the set of memory locations accessed by the transaction.

In contrast, this article advances the following thesis: Transactional memory should be virtualized to support transactions of arbitrary footprint and duration. Such support should be provided through hardware and be made visible to software through the machine's instruction set architecture. We call a transactional memory system *unbounded* if the system can handle transactions of arbitrary duration that have footprints nearly as big as the system's virtual memory.

The primary goal of unbounded transactional memory is to make concurrent programming easier without incurring much implementation overhead. We are interested in unbounded transactions because neither programmers nor compilers can easily cope with an architecturally imposed hard limit on transaction size. An implementation might be optimized for transactions below a certain size but must still operate correctly for larger

0272-1732/06/$20.00 © 2006 IEEE

transactions. The transactional hardware's size should be an implementation parameter—like cache size or memory size—that can vary without affecting the portability of binaries.

In an earlier work,[7] we proposed UTM, a specific implementation of unbounded transactional memory. UTM is a general and flexible architecture. We also describe large transactional memory (LTM), an evolutionary step between today's systems and truly unbounded transactional memory. Whereas UTM requires modifications to the memory interface, an LTM processor is pin-compatible with today's processors.

UTM architecture

UTM virtualizes transactions, allowing them to grow (nearly) as large as virtual memory. It also supports a semantics for nested transactions, in which 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, cache, or both, UTM maintains bookkeeping information for a transaction in a memory-resident data structure, the transaction log. This enables transactions to survive time slice interrupts and process migration from one processor to another.

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

- XBEGIN pc, *begin a new transaction.* The pc argument of XBEGIN specifies the address of an abort handler (for example, using a program-counter-relative offset). If at any time during a transaction's execution, the hardware determines that the transaction must fail, it immediately rolls back the processor and memory states to their states before XBEGIN executed and then jumps to pc to execute the abort handler.
- XEND*, end the current transaction.* If XEND completes, 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 XBEGIN as a not-taken branch, entering the transaction's body instead of branching to the handler. Eventually, the processor realizes that the transaction cannot commit and then reverts all processor and memory state back to the misprediction point, and branches to the abort handler.

UTM supports transaction nesting by subsuming the inner transaction. For example, an outer transaction might call a subroutine that contains an inner transaction. UTM simply treats the inner transaction as part of the atomic region defined by the outer transaction. This strategy is correct because it maintains the inner transaction's atomic execution property. We implement subsumed nested transactions by using a counter to keep track of nesting depth. If the nesting depth is positive, XBEGIN and XEND simply increment and decrement the counter, respectively, and perform no other transactional bookkeeping.

Rolling back processor state

If the processor determines that an XBEGIN instruction has "mispredicted" (its transaction must fail) while the instruction is still in the reorder buffer, rolling back the processor state is easy. The hardware simply invokes its mispredicted-branch mechanism. Unlike normal mispredicted branches, however, an XBEGIN instruction can require rolling back the processor state after the instruction has graduated from the reorder buffer. Handling this case relies on a key observation: Although the processor can internally execute ahead speculatively through many transactions, only one uncommitted, outermost XBEGIN instruction can have graduated. Hence, the processor must save only one additional copy of the architectural state to handle transaction failure, in addition to the copies implicitly kept to handle regular mispredictions.

Figure 1 shows UTM's modifications for handling rollback. We assume that the machine has a unified physical register file for both committed and speculative values (with no data in the reorder buffer), which takes snapshots of the rename table at every branch to recover mispredictions. When a regular branch instruction graduates, the rename table discards its snapshot. In contrast, when

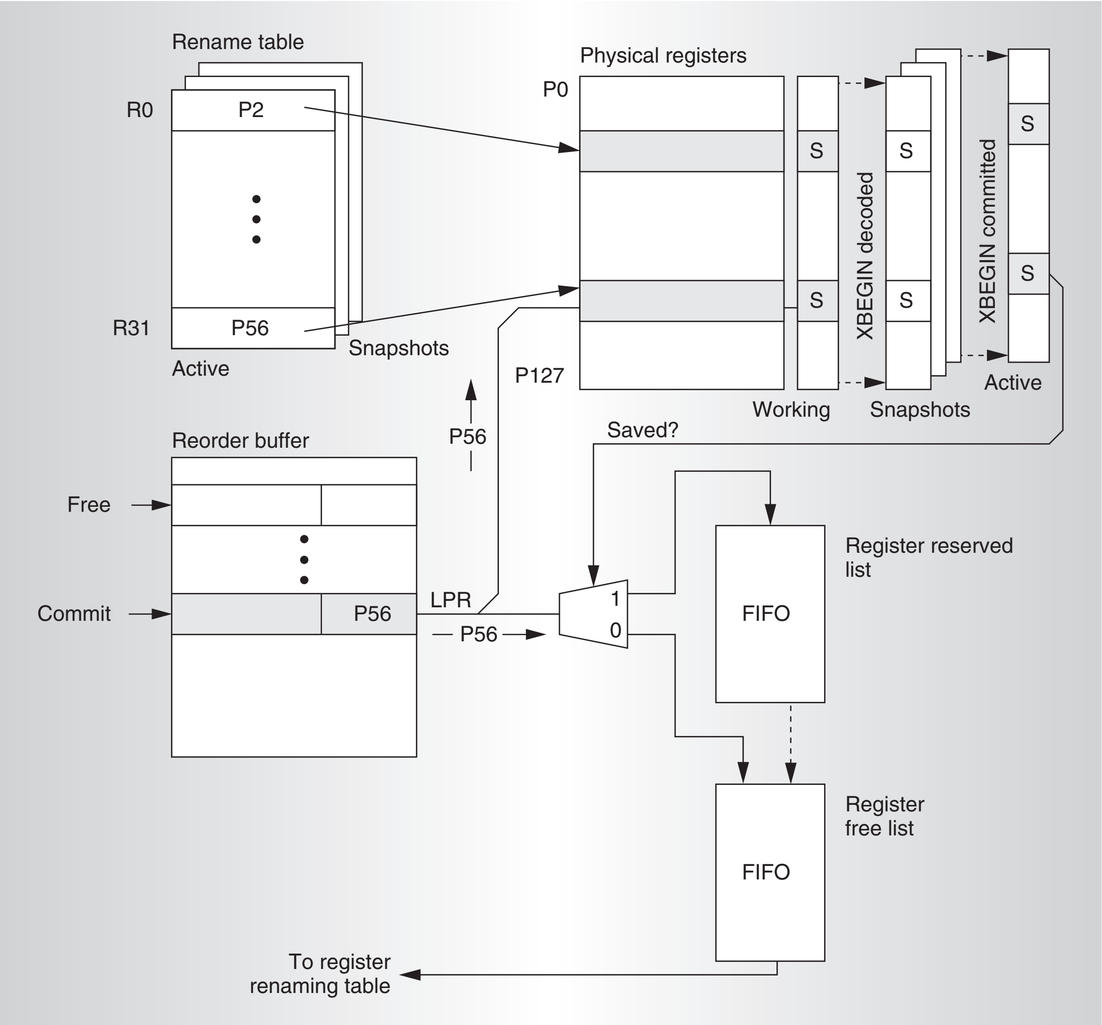

Figure 1. UTM processor modifications. The *S* bit vector tracks active physical registers. For each rename table snapshot, there is an associated *S* bit vector snapshot. The register-reserved list holds the otherwise free physical registers until the transaction commits. The last-physical-register (LPR) field identifies the physical register to free when an instruction graduates (the last physical register referenced by the destination architectural register).

an XBEGIN graduates, the rename table retains its snapshot until the corresponding XEND graduates, committing the transaction. The physical registers named in the extra snapshot must not be reused until the transaction commits. To keep track of busy registers, the rename stage maintains an *S* (saved) bit for each physical register to indicate which registers are part of the working architectural state, and it includes the *S* bits with every renaming-table snapshot.

When an XBEGIN instruction graduates, we say the transaction is active, and the associated snapshot identifies 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, however, the physical register is added to the register-reserved list instead of the normal register-free list, preventing the reuse of physical registers containing saved data. When the transaction's XEND graduates, the active snapshot's *S* bits are cleared and the register-reserved list drains into the register-free list. If the active transaction aborts, UTM restores the architectural-register state using the saved rename table, sets

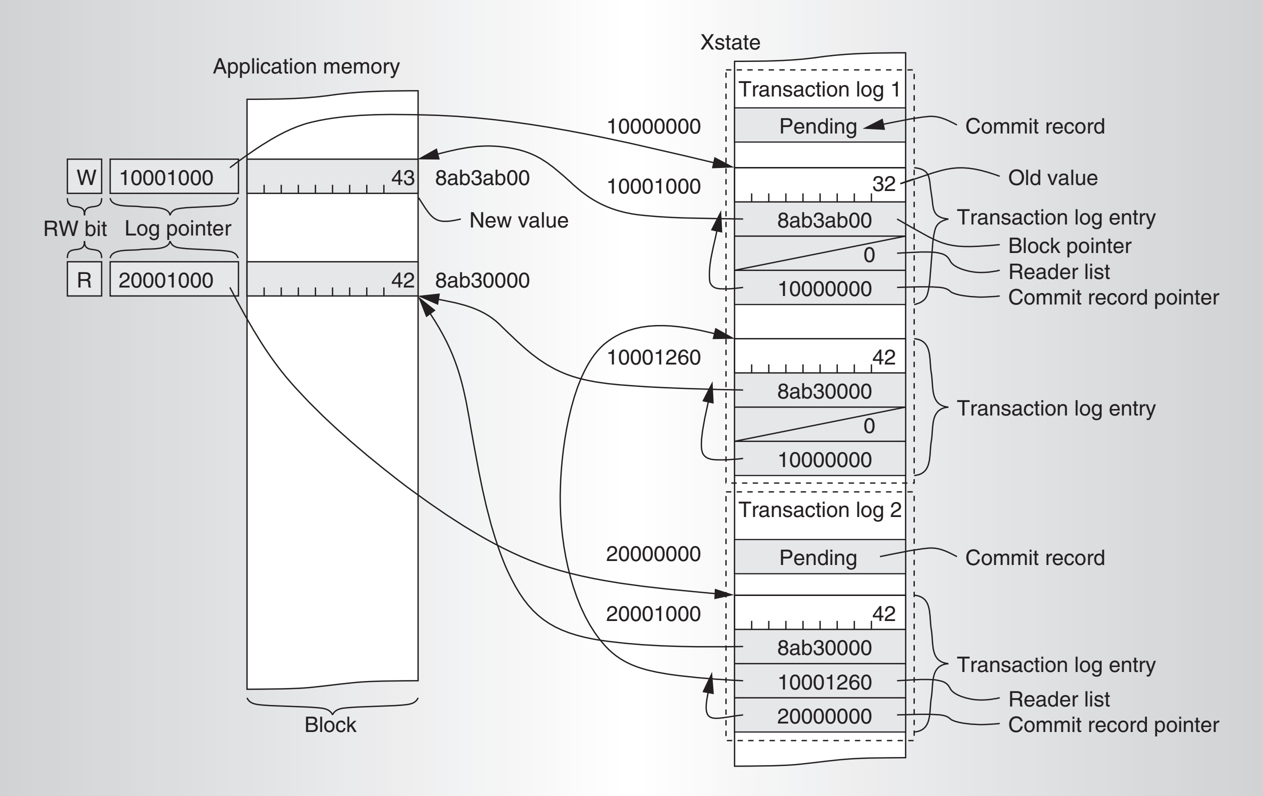

Figure 2. X-state data structure. Each transaction's log contains a commit record and a vector of log entries. The log pointer of a memory block points to a log entry, which contains the block's old value and a pointer to the transaction's commit record.

the reorder buffer to an empty state, and branches to the abort handler.

An active transaction's abort handler address, nesting depth, and snapshot are part of its transactional state. UTM makes them visible to the operating system so that they can be saved and restored on context switches.

Memory state

Previously proposed HTM systems[1,2] 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 represents the set of active transactions with a single data structure held in system memory, the *x-state* (short for "transaction state"). UTM systems use the cache to gain performance, but the correctness of UTM doesn't depend on having a cache. Here, we first describe the x-state and how the system uses it when there is no caching. Then we describe how caching accelerates x-state operations.

Figure 2 shows an abstract x-state implementation. In practice, we would optimize the x-state representation, but here present it unoptimized to aid understanding. The x-state contains a transaction log for each active transaction in the system. The operating system allocates a transaction log for each thread, and two processor control registers hold the base and bounds of a running 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 memory block that the transaction has read or written to. The entry provides a pointer to the block and the block's old value, 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 x-state consists of a log pointer and one read-write bit for each memory block (and any swapped out to disk if paging). If the RW bit is R, any transactions that have accessed the block did so with a load. If the RW bit is W, the block is the target of a transaction's store. When a processor running a transaction reads or writes a block, the block's log pointer points to a transaction log entry for that block. Further, if the access is a write, the block's RW bit 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: use the old value, use the new value, or abort the transaction.

When a processor makes an update as part of a transaction, it stores the new value in memory and the old value in a transaction log entry. In principle, there is one log entry for every load or store the transaction performs. 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 as to whether a log entry for the block already exists as part of the running transaction.

By following the log pointer to the log entry and then following the log entry pointer to the commit record, the user can determine each block's transaction status. 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 and clearing the log pointer for each block mentioned, thereby finalizing the block's contents. 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 have the system store a block's old value 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 the RW bit is R. In the latter case, it might be necessary to create a log entry for this block and add the entry to the reader list. When a transaction performs a store, the system checks that the log pointer references no other transaction (in other words, that the log pointer is cleared or that the current transaction log contains all log entries linked to this block). If the conflict check fails, some of the conflicting transactions are aborted.

To guarantee forward progress, UTM writes a time stamp into the transaction log the first time a processor attempts a transaction. Then, when choosing which transactions to abort, older transactions take priority. (Alternatively, the system could use a back-off scheme.) In practice, the transaction priority could be part of the operating system's own administrative priority scheme, allowing high-priority transactions to prevail over low-priority ones, breaking ties with the time stamp.

When a writing transaction wins a conflict, it might require aborting multiple reading transactions. The system finds these transactions efficiently by following the block's log pointer to an entry and traversing its reader list, which enumerates all entries for that block in all transaction logs.

Caching

Any memory system needs caching to achieve acceptable performance. In the common case of a transaction that fits in cache, UTM, like earlier proposed HTM systems, monitors 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 of the entire 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. In many cases, the system doesn't even need to write back a transaction log because the transaction log's contents are undefined after the transaction commits or aborts. Thus, a small transaction that commits without intervention from another transaction requires no additional interprocessor communication beyond coherence traffic for the nontransactional case.

When a transaction is too big to fit in cache or the cache protocol indicates interactions with other transactions, the x-state 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 the block is evicted.

After a transaction commits or aborts, the system can discard log entries of unspilled cached blocks and mark the log pointer of each such block clean to avoid write-back traffic for the log pointer, which is no longer needed. The uncommon case bears most of the overhead, allowing the common case to run fast.

The system can optimize the transaction log's on-processor representation as long as the view of the x-state is properly maintained at the cache interface. For convenience, the transaction block size can match the cache line size.

System issues

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

Treating the x-state as a data structure solves part of the problem directly. For a transaction to run for an indefinite period, it must be able to survive a time slice interrupt. Adding the log pointer to the processor state and storing everything else in a data structure makes it easy to suspend a transaction and run another thread with its own transaction. Similarly, transactions can migrate 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 x-state out of main memory, the UTM data structures use global virtual addresses for their pointers. Global virtual addresses are unique systemwide 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 with ordinary virtual-memory systems. Hardware first translates a process's virtual address into a global virtual address, which then translates into a physical address. Examples of architectures providing global virtual addresses include the Hewlett-Packard Precision Architecture and the IBM PowerPC.

The UTM system stores the log pointer and state bits for each user memory block, which are typically not visible to a user-level programmer, in addressable memory so that the operating system can 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 translation look-aside buffer.

During execution of a single load or store instruction, the processor potentially can touch many disparate memory locations in the x-state, any of which can be paged out to disk. To ensure forward progress, the system must allow load or store instructions to restart in the middle of the x-state traversal. Alternatively, if the processor allows only precise interrupts, the operating system must ensure that all pages required by an x-state traversal can be resident simultaneously so that the load or store can 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 executes during a transaction.

LTM microarchitecture

UTM requires significant changes to both the processor and the memory subsystem of current computer architectures. The LTM microarchitecture provides an evolutionary path from today's processors, because it requires changes only to the processor chip. We implemented a detailed, cycle-accurate LTM simulation using the UVsim processor simulator.[8]

LTM avoids the intricacies of virtual memory by limiting a transaction's footprint to (nearly) the size of physical memory. In addition, a transaction's duration must be less than a time slice, and transactions cannot migrate between processors. With these restrictions, we can implement LTM by modifying only the cache and processor core; we do not modify main memory, the cache coherence protocols, or even the contents of cache coherence messages.

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.[1,2] LTM also employs an architectural state-saving 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, allowing LTM to handle transactions too large for the cache without the x-state data structure's implementation complexity. Previous work on thread-level speculation also used an overflow region in physical memory to buffer speculative state.[9]

LTM and UTM have similar semantics, and the formats and behaviors of their XBEGIN and XEND instructions are the same. LTM puts the information that UTM keeps in the transaction log in three places: the processor, cache, and an area of physical memory allocated by the operating system.

For small transactions, LTM uses the cache to store the speculative transactional state. For large transactions, transactional state spills into an overflow data structure in main memory. LTM adds a bit (*T*) per cache line to indicate that 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 that it has overflowed. When a transactional cache line is evicted from the cache for capacity reasons, the bit is set.

In LTM, the main memory always contains the original state of data being modified transactionally, and all speculative transactional state goes into the cache and overflow hash table. The system commits a transaction by simply clearing all the *T* bits in cache and writing all overflowed data back to memory. LTM uses the cache coherency protocol to detect conflicts. When an incoming cache intervention hits a transactional cache line, the system aborts the running transaction by clearing all the *T* bits and invalidating all modified transactional cache lines.

Evaluation

Because no large-scale applications that use transactional memory currently exist, we developed translation tools to convert C and Java programs that use locks into transactional programs. We converted the Linux 2.4.19 kernel, written in C and running under user-mode Linux, and the SPECjvm98 benchmarks, written in Java, to use transactions. Although this methodology produces applications that retain some vestiges of locking, 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 ran three versions of the SPECjvm98 benchmark suite on one processor of a cycle-accurate multiprocessor simulator to measure synchronization overheads with locks and transactions. The benchmarks indicated that the LTM hardware spends little time handling overflows, but large transactions that cause overflow do occur.

Our execution-driven experiments[8] used UVsim, a multiprocessor simulator based on Rsim.[10] The cycle-accurate processor model is based on a MIPS R10K 4-issue out-of-order superscalar processor, extended with 96 physical registers and the additional register and cache support for LTM. We modeled a 2-GHz CPU, 32-Kbyte 4-way associative instruction and data L1 caches with 64-byte cache lines, a 1-Mbyte 4-way unified L2 cache with 128-byte cache lines, and a 400-Mbps double-data-rate (DDR2) SDRAM memory system. The system has a distributed, directory-based cache coherence protocol based on the SGI Origin multiprocessor, with a 10-cycle-per-hop interprocessor network latency. To run the microbenchmarks and the

Table 1. Experimental results of converting the Linux 2.4.19 kernel and the SPECjvm98 benchmarks into transactional programs.

Program	Input size (%)	No. of total memory operations	Cache miss (%)	No. of transactions	No. of oversized transactions	Transactional operations (%)	Transaction miss (%)	Overflow (%)	Biggest transaction (no. of cache lines)
make_linux	NA	315,776,028	0.56	6,964,277	3,368	41.0	0.017	NA	8,144
dbench	NA	100,928,220	0.43	1,863,426	88	49.5	0.001	NA	7,047
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

SPECjvm98 benchmarks on UVsim, we compiled them into MIPS Irix binaries with instruction extensions for transactions.

Table 1 shows the results of our trace analysis of the Linux 2.4.19 kernel and the Java benchmarks. For the Java benchmarks, we show results for runs with 1 and 100 percent of the full input size. For the percentages, we write "0" for numbers that are exactly zero, and a zero percentage, such as "0.00," for small nonzero values.

The number of total memory operations column shows 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 garbage collection. Cache miss is the fraction of memory operations that caused cache misses. An oversized transaction did not fit entirely within the cache. The transaction operations column gives the fraction of memory operations that were in transactions. Transaction miss 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 cache sets that overflowed. We measured this at the end of each transaction and averaged it over all transactions, obtaining a rough measure of the likelihood that a cache intervention request from another processor would need to look at the overflow buffer. Biggest transaction is the largest number of distinct cache lines that any transaction touched. A fully associative, bounded-size hardware transaction scheme would need a cache of at least this size.

For both kernel workloads, make_linux and dbench, the transaction operations column shows that over 40 percent of the kernel's memory operations take place in transactions, indicating that a software transactional memory would likely be too slow. Many of the SPECjvm98 benchmarks exhibit similar numbers. The oversized transactions, biggest transaction, and overflow columns show that some applications contain transactions whose footprints would overflow any reasonable-size cache. But these big transactions do not cost much—for example, the cache miss percentage is typically far greater than transaction miss percentage.

We studied the make_linux and dbench kernel benchmarks more closely to understand how cache size affects the overflow of transactional state in UTM and LTM. Figure 3 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 percent need fewer than 54 cache lines.

We instrumented the Linux benchmarks to measure lock and cache contention for insight into the available concurrency of the locking and transactional kernels. For make_linux, the processor held the hottest lock (the kernel lock) about four times longer than the hottest cache line, corroborating the findings of our cycle-accurate simulation that transactions increase concurrency, as also reported in the literature. Reducing the dependence on the kernel lock has been the focus of years of effort by kernel developers, but progress has been slow. Inspection of the code that manipulates the hottest cache line reveals that minor data restructuring using transactional memory would yield an additional 25 percent improvement in concurrency. This optimization is not easily available to kernel programmers, however, because it would make obeying the protocol that dictates the lock acquisition order difficult. Thus, transactional memory's main claim—that it makes concurrent programming easier—appears to be valid because it provides greater concurrency compared with locks and it makes enhancing concurrency easier.

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

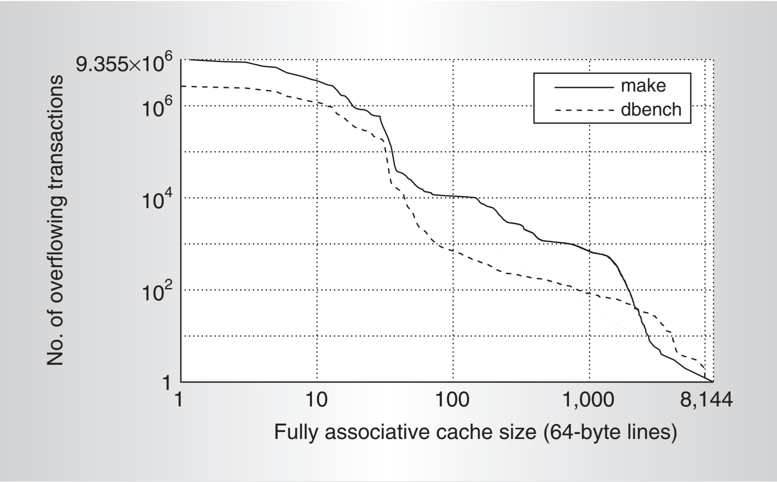

Figure 3. Fully associative cache size requirements for make_linux and dbench. Both axes are log-log.

We have made a case for virtualizing transactional memory systems to support unbounded transactions in hardware. UTM represents an ambitious but fully scalable point in the design space. LTM represents an immediately buildable alternative that provides many of the same advantages. Undoubtedly, other engineering trade-offs are possible. In addition, many 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 operations? Certain inherently serial I/O operations seem to require the use of mutual exclusion. Newer devices tend to provide multithreaded interfaces that allow bundling of nonmodal commands whose execution can be initiated with a single atomic I/O operation. It may be that we can make our hardware memory commit mechanism 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 whether additional support is desirable, or barriers and other conventional interthread ordering techniques built with transactional primitives suffice.

Another open question is whether an HTM system can implement a more optimistic concurrency control mechanism,[11] instead of the pessimistic concurrency control mechanism presented here. Various optimistic concurrency protocols are now widely deployed in database systems, suggesting that the benefits of the protocols' increased concurrency often outweigh their implementation complexity.

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

Acknowledgments

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

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. T. Knight, "An Architecture for Mostly Functional Languages," *Proc. ACM Conf. LISP and Functional Programming* (LFP 86), ACM Press, 1986, pp. 105-112.
2. M. Herlihy, J. Eliot, and B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. 20th Ann. Int'l Symp. Computer Architecture* (ISCA 93), IEEE Press, 1993, pp. 289-300.
3. J.M. Stone et al., "Multiple Reservations and the Oklahoma Update," *IEEE Parallel and Distributed Technology*, vol. 1, no. 4, Nov. 1993, pp. 58-71.
4. R. Rajwar and J.R. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 5-17.
5. N. Shavit and D. Touitou, "Software Transactional Memory," *Proc. 14th Ann. Symp. Principles of Distributed Computing* (PODC 95), ACM Press, 1995, pp. 204-213.
6. M. Herlihy et al., "Software Transactional Memory for Dynamic-Sized Data Structures," *Proc. 22nd Ann. Symp. Principles of Distributed Computing* (PODC 03), ACM Press, 2003, pp. 92-101.
7. C.S. Ananian et al., "Unbounded Transactional Memory," *Proc. 11th Int'l Symp. High-Performance Computer Architecture* (HPCA 05), IEEE CS Press, 2005, pp. 316-327.
8. S. Lie, *Hardware Support for Unbounded Transactional Memory*, master's thesis, Electrical Eng. and Computer Science Dept., MIT, 2004.
9. M. Prvulovic et al., "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization," *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA 01), ACM Press, 2001, pp. 204-215.
10. C.J. Hughes et al., "Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors," *Computer*, vol. 35, no. 2, Feb. 2002, pp. 40-49.
11. H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Systems*, vol. 6, no. 2, June 1981, pp. 213-226.

C. Scott Ananian is a PhD candidate in electrical engineering and computer science at MIT. His research interests include compilers, language design, embedded systems, and transactions. Ananian has a BSE in electrical engineering from Princeton University and an MSc in electrical engineering and computer science from MIT. He is a member of the IEEE and the ACM.

Krste Asanović is an associate professor in the Department of Electrical Engineering and Computer Science at MIT and a member of the MIT Computer Science and Artificial Intelligence Laboratory. His research interests include computer architecture and VLSI design. Asanović has a BA in electrical and information sciences from the University of Cambridge and a PhD in computer science from the University of California, Berkeley. He is a member of the IEEE and the ACM.

Bradley C. Kuszmaul is a research scientist in the Supercomputing Technologies Group of the MIT Computer Science and Artificial Intelligence Laboratory. His research interests

include developing computer systems with provably good performance. Kuszmaul has a PhD in computer science from MIT. He is a member of the IEEE and the ACM.

Charles E. Leiserson is a professor of computer science and engineering in the MIT Computer Science and Artificial Intelligence Laboratory, a member of its Theory of Computation Group, and head of its Supercomputing Technologies Group. His research interests include algorithms, multithreading, and computing machinery theory. Leiserson has a BS in computer science and mathematics from Yale University and a PhD in computer science from Carnegie Mellon University. He is a member of the ACM, the IEEE, and SIAM.

Sean Lie is a design engineer in the architecture group at Advanced Micro Devices. His research interests include high-performance computer architecture and VLSI design. He has a BS and an MEng, both in electrical engineering and computer science, from MIT, where he participated in the work described here.

Direct questions and comments to Charles E. Leiserson, MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., #32-G768, Cambridge, MA 02139; cel@mit.edu.