People who are really serious about software should make their own hardware. Remember, it's all software, it just depends on when you crystallize it.

Alan Kay

Chapter 6

Transactions in hardware: Unbounded Transactional Memory

The previous chapters have detailed the design of an efficient software-only transaction system for object-oriented programs. Given hardware support, we can construct even more efficient transaction systems for certain types of transactions. In this chapter, we will present UTM, an implementation of unbounded transactional memory [5] which fully virtualizes transactions. We will follow this with LTM, a much simpler design which can be pin-compatible with today's processors. LTM supports more limited transactions, but we conclude by showing how LTM can be combined with the efficient software system we have already demonstrated to yield a hybrid system with a great deal of power and flexibility.[1]

[1] Portions of this chapter are adapted from [5], co-written with Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie.

6.1 The UTM architecture

We begin by describing UTM, a system 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.

6.1.1 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.

In the same manner as our software implementation (chapter 4.2.1), 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.

6.1.2 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 6.1. 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.

6.1.3 Memory state

Previous HTM systems [46, 36] 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 6.2. 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

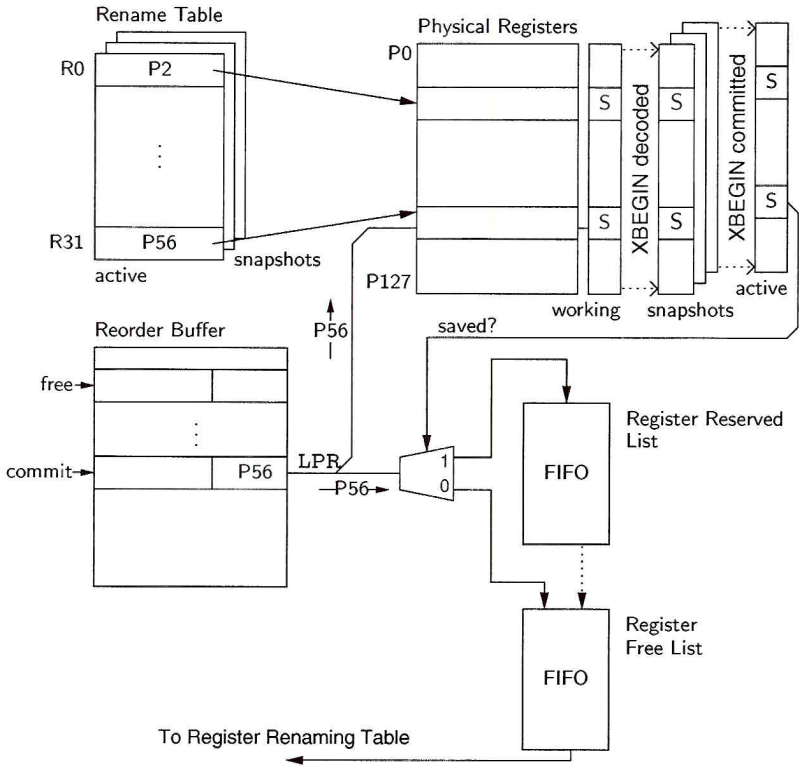

Figure 6.1: UTM processor modifications. The S bit vector tracks the active physical registers. For each rename table snapshot, there is an associated S bit vector snapshot. The Register Reserved List holds the otherwise free physical registers until the transaction commits. The LPR field is the next physical register to free (the last physical register referenced by the destination architectural register).

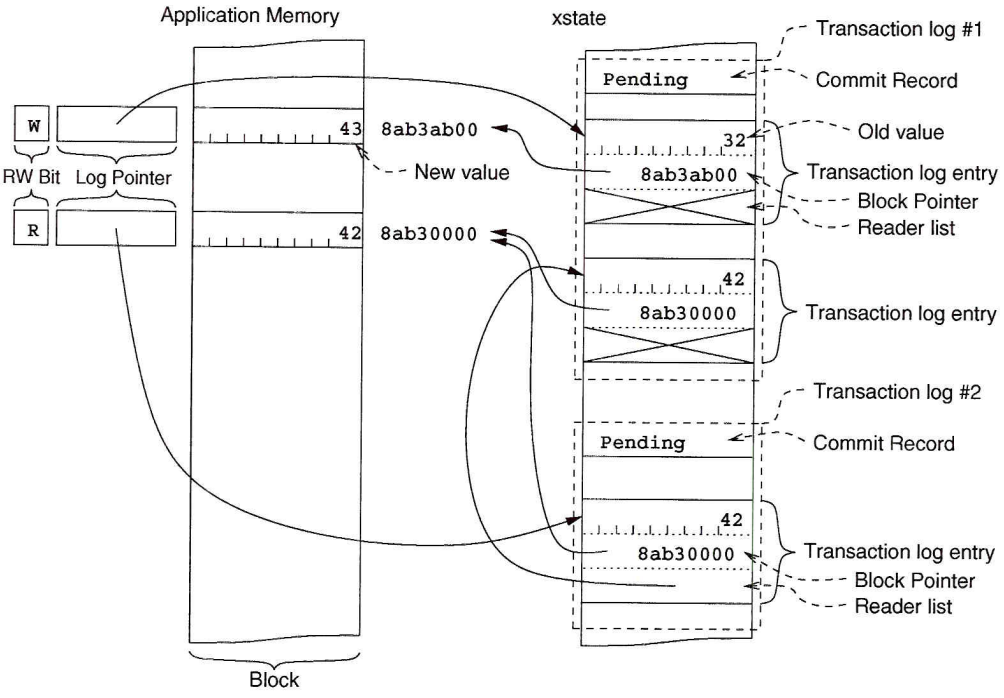

Figure 6.2: The xstate data structure. The transaction log for a transaction contains a commit record and a vector of log entries. The log pointer of a block in memory points to a log entry, which contains the old value of the block and a pointer to the transaction's commit record. Two transaction logs are shown here; generally, the xstate includes the active transaction logs for the entire system.

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 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 [53] 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.

6.1.4 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 [46, 36], 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.

6.1.5 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 [11] provided user-level global virtual addressing using segment-offset pairs as the addresses. The HP Precision Architecture [49] 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.

6.2 The LTM architecture

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 com-

promise 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 [71]. The limited transactions supported by LTM are still powerful enough to serve as the basic for an hybrid system, as we will show in chapter 6.4.

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 [46, 40]. 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

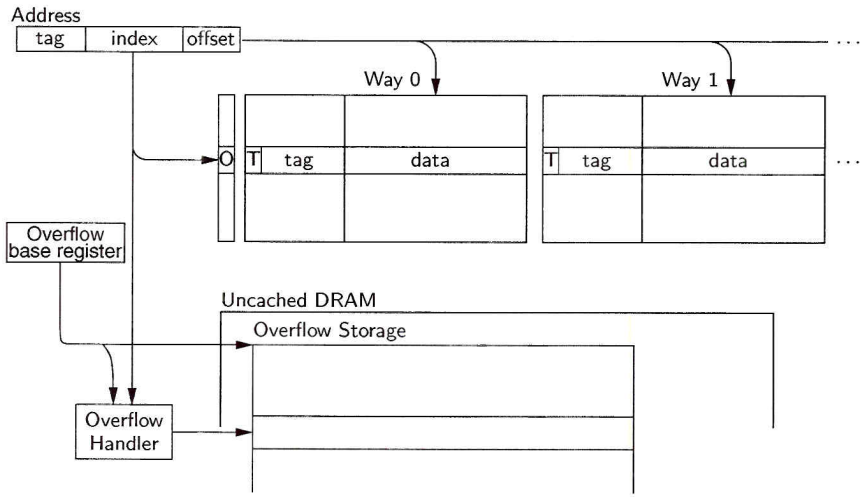

Figure 6.3: LTM cache modifications. The T bit indicates if the line is transactional. The O bit indicates if the set has overflowed. Overflowed data is stored in a data structure in uncached DRAM.

system.

LTM requires only a few small modifications to the cache, as shown in Figure 6.3. 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. As I will show at the end of the chapter, LTM's more practical implementation of quasi-unbounded transactional memory suffices for many real-world concerns, and can be symbiotically paired with our more flexible software transaction system to achieve truely unbounded transactions at minimal hardware cost.

6.3 Evaluation

In this section we will evaluate the UTM and LTM designs, demonstrating low overhead and scalability. We will also consider overflow behavior, providing some motivation for the hybrid system proposed in the next section.

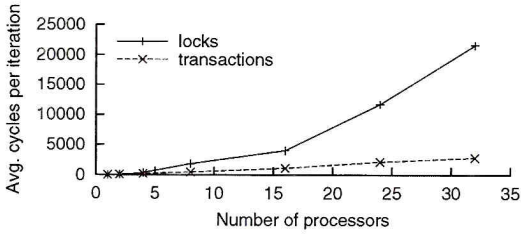

Figure 6.4: `Counter` performance on UVSIM.

6.3.1 Scalability

We used a parallel microbenchmark to examine program behavior for small transactions with high contention. Our results show that the extremely low overhead of small hardware transactions enable them to almost always complete even when contention is high.

The `Counter` microbenchmark has one shared variable that each processor atomically increments repeatedly with no backoff policy; the basic idea is identical to the microbenchmark we used in chapter 3.4. Each transaction is only a few instructions long and every processor attempts to write to the same location repeatedly. Both a locking and a transactional version of `Counter` were run on UVSIM with LTM, and the results are shown in Figure 6.4. In the locking version, there is a global spin-lock that each processor obtains using a load-linked/store-conditional (LLSC) sequence from the SGI synchronization libraries.

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 backoff policy. 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 complete (and perhaps even start and complete the next transaction) before the cache line is taken away. Similar behavior is expected from UTM, and other

transactional-memory systems which use the cache and cache-coherency protocol to store transactional state, since small transactions effectively use the cache the same way.

6.3.2 Overhead

A main goal of LTM and UTM is to run the common case fast. As shown in chapter 3.1, the common case is when transactions are small and fit in the cache. Therefore, by using the cache and cache coherency mechanism to handle small transactions, LTM is able to execute with almost no overhead over serial code in the common case. In this section, we discuss qualitatively how the LTM implementation is optimized for the common case and how similar techniques are used in UTM. The discussion is broken into the following three cases: starting, running, committing a transaction.

Starting a transaction in LTM requires virtually no overhead in the common case since the hardware only needs to record the abort handler address. No communication with the cache or other external hardware is necessary. There is the added overhead of decoding the `XBEGIN` however that overhead is generally insignificant compared to the cost of the transaction. Further, instruction decode overhead is much lower in LTM than with locks. Even schemes where the lock is not actually held such as SLE [57] have higher decode overhead since they have more instructions. LTM's low transaction startup overhead is a very good indicator of the corresponding overhead in UTM since transaction start up in UTM is virtually the same.

Running a transaction in LTM requires no more overhead than running the corresponding non-synchronized code in the common case. In LTM, the T bit is simply set on each transactional cache access. LTM's low overhead in this case unfortunately does not translate directly to UTM since UTM modifies the transaction record on each memory request. However, in the common case the transaction record entry is also in the cache. Thus all operations are local and no external communication is needed. Also, in some cases, the cache can respond to the memory request once the requested data

is found. However, if the request requires data from the transaction record before it can be serviced, an additional cache lookup is necessary. However, the lookup is local and thus can be done relatively quickly. Therefore, the common case overhead of running a transaction can be minimal even in UTM.

Committing a transaction in LTM has virtually no overhead in the common case since it can be done in one clock cycle. LTM transaction commits only requires a simply flash clear of all the transaction bits in the cache. Similarly, UTM transaction commits only require a single change of the cached transaction record to "committed". UTM transaction commit also writes the updated values from the transaction record back to memory. However, this write back can be done lazily in the background. Therefore, since transaction commit requires only a single change in the cache for both LTM and UTM, the overhead is minimal in both cases.

6.3.3 Overflows

Overflows occur only in the uncommon case however our studies show that it is important to have a scalable data structure even though it is used infrequently.

For evaluation, we compiled three versions of the SPECjvm98 benchmark suite to run under UVSIM using FLEX. 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 described in chapter 4.2, our transactional version uses method cloning to flatten transactions; we performed the same cloning on the other compiled versions so that performance improvements due to the specialization would not be improperly attributed to transactionification. The three different benchmark versions were built from a common code-base using method

inlining in gcc[2] to remove or replace all invocations of lock and transaction entry and exit code with appropriate implementations. No garbage collection was performed during these benchmark runs.

Our initial results from chapter 3.1 suggested that overflows ought to be infrequent, thus the efficiency of the overflow data structure would have a negligible effect on overall performance. Therefore, our first LTM implementation used an unsorted array that required a linear search on each miss to an overflowed set. The unsorted array was effective for most of our test cases, as they had less overhead than locks. Using LTM with the unsorted array, however, the transactional version of 213_javac was 14 times slower than the base version. Virtually all of the overhead came from handling overflows, which is not surprising, since the entire application is enclosed in one large transaction. The large transaction touches 13K cache lines with 9K lines overflowed. So, even though only 0.5% of the transactional memory operations miss in the cache, each one incurs a huge search cost. This unexpected slowdown indicated that a naive unsorted array is insufficient as an overflow data structure. Therefore, LTM was redesigned to use a hash table to store overflows.

Since the entire application was enclosed in a transaction, the 213_javac application was clearly not written to be a parallel application. However, it is important that an unbounded transactional memory system be able to support even such applications with reasonable performance. Therefore, we redesigned LTM to use hash table as described in chapter 6.2.

Using LTM with the hash table, the SPECjvm98 application overheads were much more reasonable as shown in Figure 6.5. The hash table data structure decreased the overhead from a 14x slowdown to under 15% in 213_javac. Using the hash table, LTM transactional overhead is less than locking overhead in all cases.

[2]We compiled the files generated by FLEX's "Precise C" backend (chapter 4.1) with -O9 for a -mips4 target using the n64 API to generate fully-static binaries executable by 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	124.0%	101.0%	32.5%	0.0085%
202_jess	75.0M	140.9%	108.0%	59.4%	0.0072%
209_db	11.8M	142.4%	105.2%	54.0%	0%
213_javac	30.7M	169.9%	114.2%	84.2%	10%
222_mpegaudio	99.0M	100.3%	99.6%	0.8%	0%
228_jack	261.4M	175.3%	104.3%	32.1%	0.0056%

Figure 6.5: 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 1. The overflow hash table is 128MB.

6.4 A hybrid transaction implementation

We've seen that UTM and LTM can operate with very little overhead, but hardware schemes encounter difficulties when scaling too very large or long-lived transactions. We have overcome some of the difficulties with an overflow cache (LTM), or by virtualizing transactions and dumping their state to a data structure (UTM). However, it is worth considering whether this extra complexity is worthwhile: why not combine the strengths of our object-based software transaction system (explicit transaction state, unlimited transaction size, flexibility) with the fast small transactions at which a hardware system naturally excels?

Figure 6.6 presents the results of such a combination. In the figure, combining the systems is done in the most simple-minded way: all transactions are begun in LTM, and after any abort the transaction is restarted in the object-based software system. The field flag mechanism described in section 3.2.5 ensures that software transactions properly abort conflicting hardware transactions: when the software scribbles `FLAG` over the original field the hardware will detect the conflict. Hardware transactions must per-

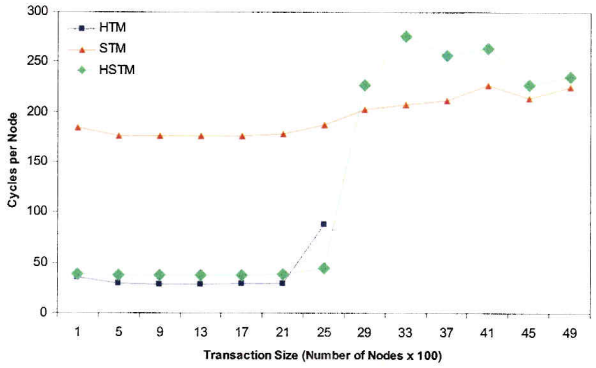

Figure 6.6: Performance (in cycles per node push on a simple queue benchmark) of LTM [5] (HTM), the object-based system presented in this paper (STM) and a hybrid scheme (HSTM).

form the `ReadNT` and `WriteNT` algorithms to ensure they interact properly with concurrent software transactions, although these checks need not be part of the hardware transaction mechanism. In the figure, the checks were done in software, with an implementation similar to that described in chapter 3.4.

The figure shows the performance of a simple queue benchmark as the transaction size increases. The hardware transaction mechanism is fastest, as one would expect, but it performance falters and then fails at a transaction size of around 2500 nodes pushed. At this transaction size the hardware scheme ran out of cache; in a more-realistic system it might also have run out of its timeslice, aborting (LTM) or spilling (UTM) the transaction at the context switch.

Above HTM in the figure is the performance of the software transaction system, which is about 4x slower. This is a pessimistic figure: no special effort was made to tune code or otherwise minimize slowdown, and the processor simulated had limited ability to exploit ILP (2 ALUs and 4-instruction issue width). However, the software scheme is unaffected by

increasing transaction size.

The hybrid scheme successfully combines the best features of both. It is only about 20% slower than the basic hardware scheme, due to the read and write barriers it much implement, but at the point where the hardware stops working well, it smoothly crosses over to track the performance of the software transaction system.

There are many fortuitous synergies in such an approach. Hardware support for small transactions may be used to implement the software transaction implementation's Load Linked/Store Conditional sequences, which may not otherwise be available on a target processor. The small transaction support can also facilitate a functional array solution to the large object problem, as we saw in chapter 5.4. We might further improve performance by adding a bit of hardware support for the `readNT`/`writeNT` barriers [16].

I believe this hybrid approach is the most promising direction for transaction implementations in the near future. It preserves the flexibility to investigate novel solutions to the outstanding challenges of transactional models, which we will review in the next chapter, and solves an important chicken-and-egg problem preventing the development of transactional software and the deployment of transactional hardware. Since the speed of the hardware mechanism is tempered by the cooperation protocol of the software transaction system, high-efficiency software transaction mechanisms, such as the one presented in this thesis, are the key enabler for hybrid systems.

The thing is to remember that the future comes one day at a time.

Dean Acheson

Chapter 7

Challenges

In this section we will review objections that have been raised to straightforward or naïve transaction implementations. Some of these objections do not apply to our implementation; discussion of these may further illuminate our design choices. Others apply to certain situations, and should be kept in mind when creating applications. Some of the problems raised remain unsolved, and are the subject of future work; for these we will attempt to sketch research directions.

7.1 Performance isolation

Zilles and Flint [73] identified *performance isolation* as a potential issue for transaction implementations. In a system with performance isolation, the execution of one task (process, thread, transaction) should complete in an amount of time which is independent of the execution of other tasks (processes, threads, transactions) in the system. For a system with N processors, it is ideal if a task is guaranteed to complete at least as quickly as it would running alone on 1 processor.

It is obvious that most common systems do not provide any guarantee of performance isolation: on a typical multi-user system, the execution of a given task can be made arbitrarily slow by the concurrent execution of

competing tasks.[1] However, a nontransactional system can be constructed with a good deal of performance isolation by appropriately restricting the processes that can be run and the resources they consume.

Zilles and Flint object that many transactional systems are constructed such that a single large transaction may monopolize the atomicity resources such that no other transactions may commit. By opening a transaction, touching a large number of memory locations, and then never closing the transaction, a malicious application may deny service to all concurrent applications in a transaction system.

For this reason, it is important that there are no global resources required to complete a transaction. Our UTM hardware implementation achieves this end, but the LTM design uses a per-processor overflow table. If an LTM design is implemented with a snoopy bus for coherence traffic, overflows on one processor can impact the performance of all other processors on the bus. A directory-based coherence protocol (as we have described in this thesis) eliminates this problem. Hybrid schemes based on LTM also eliminate the problem, because an overflowing transaction can be aborted and retried in software, which requires no global resources.

Concerns about performance isolation are not limited to transaction systems. Transaction systems provide a solution not available to systems with lock-based concurrency, however: the offending transaction can be safely aborted at any point to allow the other transactions to progress.

7.2 Progress guarantees

Aborting troublesome transactions raises another potential pitfall: how do we guarantee that our system will make forward progress? Zilles and Flint [73] note that transaction systems are subject to an "all-or-nothing" problem: it's fine to abort a troublesome large transaction to allow other work to complete, but then we throw away any progress in that transaction.

[1] Grunwald and Ghiasi [28] call this a "microarchitectural denial of service" attack.

The operating system is forced to either allocate a large transaction all the resources it requires, or to refuse to make any progress on the transaction; there is no middle ground.

This criticism applies to the LTM hardware scheme and other unvirtualized transaction implementation. In an LTM system, it is the programmer's responsibility to structure transactions such that the application is likely to complete. The operating system will deny progress when necessary to prevent priority inversion.

The UTM, hybrid, and software-only implementations do not suffer the same problem. UTM and software-only implementations can virtualize the transaction, as all transaction state is in memory. This allows the resources required to be paged in incrementally as needed. The hybrid scheme, even when built on an unvirtualized mechanism such as LTM, can abort and fail-over to a virtualized software system if sufficient resources are not available.

7.3 The semantic gap

In a vein of optimism, transactions are often casually said to be "compatible" with locks: transform your lock acquisition and release statements to begin-transaction and end-transaction, respectively, and your application is transformed. We might even claim that your application will suddenly be faster/more parallel and that some lingering locking bugs and race conditions will be cured by the change as well.

It is the latter part of the claim that draws first scrutiny: if you've "fixed" my race conditions, haven't you altered the semantics of my program? What other behaviors might have changed?

Blundell, Lewis, and Martin [13] describe the "semantic gap" opened between the locking and naïvely-transactified code. They point out that programs with data races – even "benign" ones – may deadlock when converted to use transactions, since the data race will never occur. One may even wrap every access with a location-specific lock to "remove" the race

(for some definitions) without altering the behavior of the locking code or the deadlock for the transactional version.

This concern is valid, and claims of automatic transactification should not be taken too lightly. However, most "best-practices" concurrent code *will* behave as expected, and (unlike timing-dependent code with races) deadlocks make it very obvious where things go wrong. Further, type systems are capable of detecting the deadlocks in transactified code and alerting the programmer of the problem.

Blundell, Lewis, and Martin also point out that some transaction implementations ignore "non-transactional" accesses – even if these are to locations which are currently involved in a transaction. This leads to additional alterations in the semantics of the code. In the implementations described in this thesis, we are careful to ensure that "non-transactional" code still executes as if each statement was its own individual transaction, what Blundell, Lewis, and Martin term ***strong atomicity***.

7.4 I/O Mechanisms

To be useful, computing systems must be connected to the outside world in some fashion. This creates a discontinuity in the transactional model: real world events can not be rolled back in the same way as can changes to program state.

7.4.1 Forbidding I/O

The most straight-forward means to accommodate I/O in the transactional framework is to forbid it: I/O may only happen outside of a transaction. A runtime check or simple type system may be used to enforce this restriction.

A useful programmer technique in this model is to create a separate concurrent thread for I/O. A transaction can interact with a queue to request I/O, and the I/O thread will dequeue requests and enqueue responses. This works best for unidirectional communication. Note that round-trip commu-

nication with the I/O thread can not be accomplished within a single transaction (deadlock will result), so transactions still must be broken between a request and reply: some forms of interaction can not be accomplished atomically.

7.4.2 Mutual exclusion

Another alternative is to integrate mutual exclusion into the transaction model: once we start an I/O activity within a transaction, the transaction becomes ***uninterruptible***: it may no longer be aborted, and must be successfully executed through commit. Only a single uninterruptible transaction may execute at a given time (although other interruptible transactions may be concurrent). Effectively there is a single global mutual exclusion lock for I/O activity; transactions attempting I/O while the lock is already held are either delayed or aborted.

This scheme is reasonable as long as I/O is infrequent in transactions, and is convenient for the programmer. A single debugging print-statement, however, is sufficient to serialize a transaction, and efforts to make the single global I/O lock more fine-grained may ultimately give back the gains in simplicity and concurrency afforded by transactions in the first place.

7.4.3 Postponing I/O

A hybrid approach attempts to anticipate or postpone I/O operations so that they run only at transaction start or end. Only the I/O operations then need to be run serially; the remainder of the transaction may still execute concurrently. Input must be moved to the start of a transaction, and once the input has been consumed, that transaction must still run uninterruptibly; it may be aborted only if a push-back buffer can be constructed for the input, which is not always reasonable. Output is moved to the end of the transaction, but only the actual I/O must be performed uninterruptibly: the transaction can still be aborted at any time prior to commit.

If output and input need to be interleaved, or the input occurs after an output and thus can not be moved to the start of the transaction, uninterruptible transactions are still required. This approach thus works around the disadvantages of mutual exclusion in some cases, but a single misplaced debugging statement can still force serialization.

It is worth noting that modern interface hardware is often designed such that it works well with this approach. For example, a GPU or network card will take commands from or deliver input to a buffer; a single operation is sufficient to hand over a buffer to the card to commence I/O. This single I/O action may be made atomic with the transaction commit.

7.4.4 Integrating do/undo

The most sophisticated integration of I/O with transactions allows the programmer to specify "undo" code for I/O which can not otherwise be rolled back. In the database community, these are referred to as *compensating transactions*.

Again, I/O would be forbidden within "pure" transactions; however, do/undo blocks may be nested within transactions. A do block executes uninterruptibly. If a transaction aborts before it reaches a do block, rollback occurs conventionally. If it aborts after it has executed a do block, then the undo block is executed uninterruptibly as part of transactional rollback. Note that mutual exclusion must still be used in portions of the transaction processing; ideally the critical regions are short and infrequently invoked.

The do/undo behavior allows sophisticated exception processing: an undo block may emit a backspace to the terminal after do emits a character, or it may send an apology email after an email is sent irrevocably in a do block. The do/undo is invisible to clients outside the transaction. Sophisticated libraries can be built up using this mechanism. For example, disk i/o may be made transactional using file versioning and journalling.

The undo blocks may be difficult to program. The most straightforward implementation would prevent the undo from accessing any transactional

state, and the programmer must take special care if she is to maintain a consistent view of program state. A friendlier implementation would present a view of program state such that it appears that the undo block executes immediately after the do block, in the same lexical environment (regardless of what ultimately-aborted transactional code has executed in the interim). The programmer is then able to naturally write code such as the following:

```
String from = ..., to = ..., subject = ...;
do {
  sendEmail(from, to, subject);
} undo {
  sendApology(from, to, "Re: "+subject);
}
/* code here can modify from, to, subject */
/* before transaction commit */
```

Presenting a time-warped view to the undo block can be difficult or impossible, depending on the history mechanism involved. In particular, if the undo reads a new location, previously untouched by the transaction, the value of this location at the (previous) time immediately following the do might be unavailable. Presenting an inconsistent view of memory may be undesirable.

7.5 OS interactions

While transaction-style synchronization has been successfully used to structure interactions within an operating system [52], transactions crossing the boundary between operating system and application present additional challenges. Some operating system requests are either I/O operations or can be handled with the same mechanism used to handle I/O within transactions, as discussed in the previous section. Using memory allocation as an example of an OS request, transactions can be forbidden to allocate memory, required to take a lock, forced to preallocate all required memory,[2] or use a compensation mechanism to deallocate requested memory in case of abort.

[2] A retry mechanism can be used to incrementally increase the preallocation until the transaction can successfully complete.

Since the role of an operating system is to administer shared resources, special care must be taken that transactions involving operating system requests do not "contaminate" other processes. In particular, if data in kernel space is to be included in a transaction:

- If mutual exclusion may used to implement any part of the transaction semantics (as in the various I/O schemes above), then it may be possible to tie up the entire system (including unrelated processes) until a transaction touching kernel structures commits.

- If transaction state if to be tracked in the cache, the kernel address space must be reserved from the application memory map on architectures with virtually-addressed caches.

- It may be desirable to include some loophole mechanism so that kernel data structures may be released from the transaction. Similarly, the operating system may wish to use transactions within itself; it may be desirable for these transactions to be independent from the application's invoking transaction. Motivating examples include fault handlers and the paging mechanism, which ought to be transparent to the application's transaction.

It is possible to handle these challenges simply, for example by forbidding OS calls within a transaction and aborting transactions if necessary on context switches or faults. Such an approach raises hurdles for the application programmer but simplifies the operating system's task considerably. In unvirtualized transaction implementations such as LTM, this approach also limits the maximum duration of a transaction to a single time slice, although the OS may be able to stretch a processes time slice when necessary.

A more sophisticated approach with explicit OS management of transactions may be able to provide better transparency of OS/transaction interactions for the application programmer and improved performance.

7.6 Recommendations for future work

Based on the discussion in this section, a virtualizable transaction mechanism is recommended: if LTM is implemented in hardware, a hybrid scheme should support it in order to provide performance isolation and progress guarantees. A do/undo mechanism for transactions allows better management of critical regions where mutual exclusion must be integrated with the transaction mechanism to support I/O and certain OS interactions. All OS interactions can then be performed in do block so that mutual exclusion need not be extended across the OS boundary.

Need to integrate a discussion of SW-only schemes.