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 [7] 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 [7], 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 (Section 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 [49, 63] 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

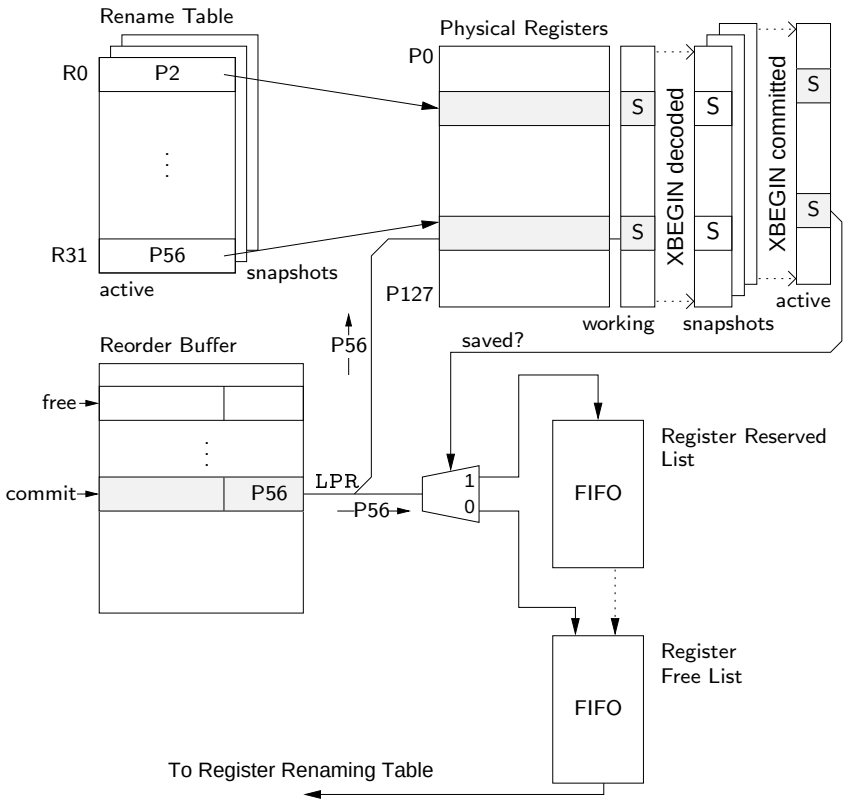

Figure 6.1: UTM processor modifications. The S bit vector tracks the active physical registers. For each rename table snapshot, there is an associated S bit vector snapshot. The Register Reserved List holds the otherwise free physical registers until the transaction commits. The LPR field is the next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

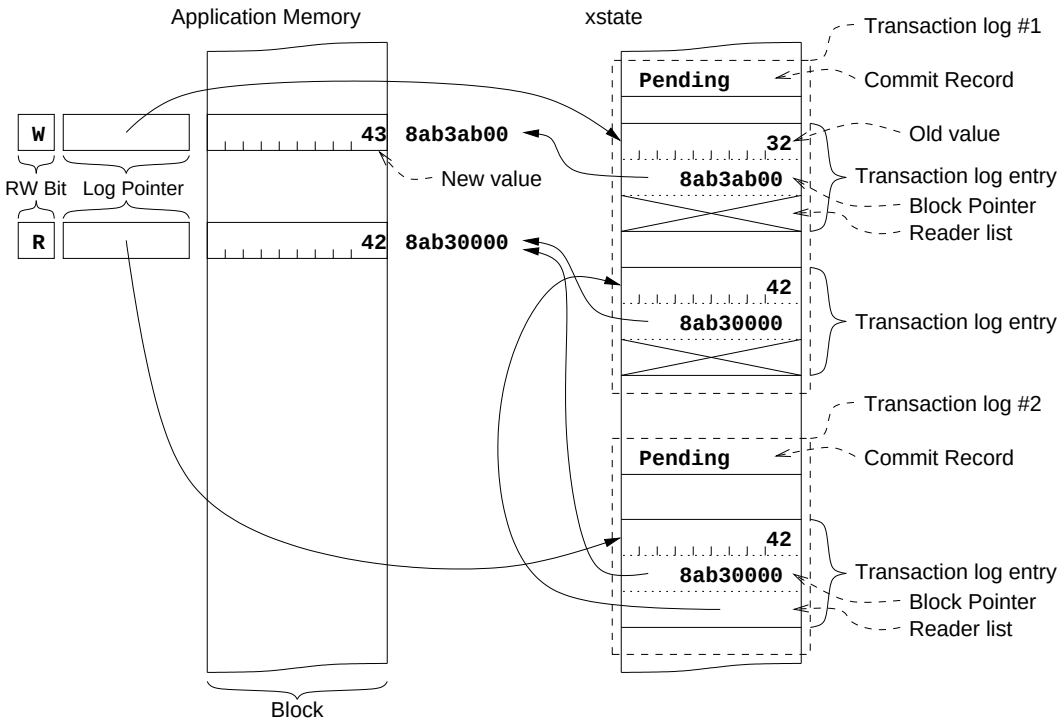

Figure 6.2: The xstate data structure. The transaction log for a transaction contains a commit record and a vector of log entries. The log pointer of a block in memory points to a log entry, which contains the old value of the block and a pointer to the transaction's commit record. Two transaction logs 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, and 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 [70] 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 [49, 63], 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 [16] provided user-level global virtual addressing using segment-offset pairs as the addresses. The HP Precision Architecture [66] 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~~ which requires significant changes to both the processor and the memory subsystem of a ~~current~~ contemporary 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 [96]. The limited transactions supported by LTM are still powerful enough to serve as the basic for an hybrid system, as we will show in Section 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 [55, 63]. 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

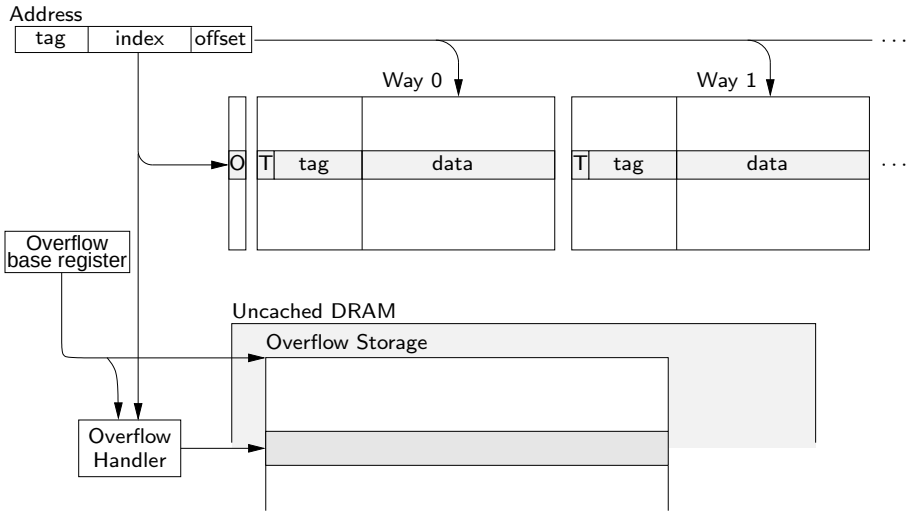

Figure 6.3: LTM cache modifications. The T bit indicates if the line is transactional. The O bit indicates if the set has overflowed. Overflowed data is stored in a data structure in uncached DRAM.

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 truly 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.

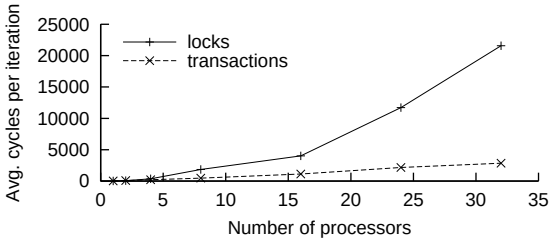

Figure 6.4: `Counter` performance on UVSIM.

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 Section 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 that 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 Section 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 [75] 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 simple 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 Section 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 Section 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 Section 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 (Section 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 to 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-

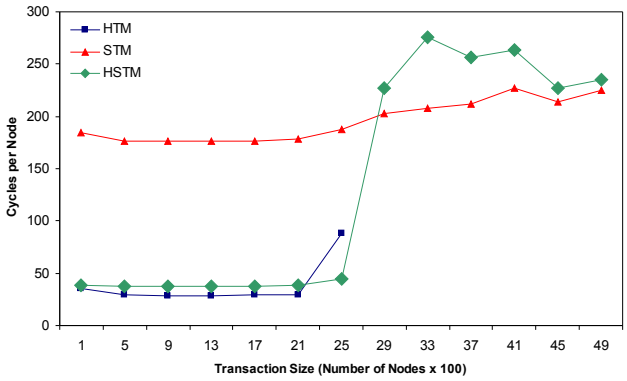

Figure 6.6: Performance (in cycles per node push on a simple queue benchmark) of LTM [7] (HTM), the object-based system presented in this paper (STM) and a hybrid scheme (HSTM).

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 Section 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 its 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 must 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 Section 5.4. We might further improve performance by adding a bit of hardware support for the `readNT`/`writeNT` barriers [22].

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 [98] 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 [98] 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 [42] 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 [18] 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 straightforward 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 [69], 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, the following challenges arise:

- 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 can be released from the transaction. Similarly, if 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.

Everything in the universe relates to [transactions], one way or another, given enough ingenuity on the part of the interpreter.

Principia Discordia (amended)

Chapter 8

Related work

A number of researchers have been investigating transactional memory systems. This thesis is the first to present a hybrid hardware/software model-checked non-blocking object-oriented system that allows co-existence of non-transactional and transactional accesses to a dynamic set of object fields.

8.1 Non-blocking synchronization

Lamport presented the first alternative to synchronization via mutual exclusion in [65], for a limited situation involving a single writer and multiple readers. Lamport's technique relies on reading guard elements in an order opposite to that in which they are written, guaranteeing that a consistent data snapshot can be recognized. The writer always completes its part of the algorithm in a constant number of steps; readers are guaranteed to complete only in the absence of concurrent writes.

Herlihy formalized *wait-free* implementations of concurrent data objects in [54]. A wait-free implementation guarantees that any process can complete any operation in a finite number of steps, regardless of the activities of other processes. Lamport's algorithm is not wait-free because readers can be delayed indefinitely.

Massalin and Pu introduced the term *lock-free* to describe algorithms

with weaker progress guarantees. A lock-free implementation guarantees only that *some* process will complete in a finite number of steps [69]. Unlike a wait-free implementation, lock-freedom allows starvation. Since other simple techniques can be layered to prevent starvation (for example, exponential backoff), simple lock-free implementations are usually seen as worthwhile practical alternatives to more complex wait-free implementations.

An even weaker criterion, *obstruction-freedom*, was introduced by Herlihy, Luchangco, and Moir in [56]. Obstruction-freedom only guarantees progress for threads executing in isolation; that is, although other threads may have partially completed operations, no other thread may take a step until the isolated thread completes. Obstruction-freedom not only allows starvation of a particular thread, it allows contention among threads to halt all progress in all threads indefinitely. External mechanisms are used to reduce contention (thus, achieve progress) including backoff, queueing, or timestamping.

We will use the term *non-blocking* to describe generally any synchronization mechanism that doesn't rely on mutual exclusion or locking, including wait-free, lock-free, and obstruction-free implementations. We will be concerned mainly with lock-free algorithms.[1]

8.2 Efficiency

Herlihy presented the first *universal* method for wait-free concurrent implementation of an arbitrary sequential object [47, 54]. This original method was based on a *fetch-and-cons* primitive, which atomically places an item on the head of a list and returns the list of items following it; all concurrent

[1] Note that some authors use "non-blocking" and "lock-free" as synonyms, usually meaning what we here call *lock-free*. Others exchange our definitions for "lock-free" and "non-blocking", using lock-free as a generic term and non-blocking to describe a specific class of implementations. As there is variation in the field, we choose to use the parallel construction *wait-free*, *lock-free*, and *obstruction-free* for our three specific progress criteria, and the dissimilar *non-blocking* for the general class.

primitives capable of solving the n-process consensus problem—*universal* primitives—were shown powerful enough to implement *fetch-and-cons*. In Herlihy's method, every sequential operation is translated into two steps. In the first, *fetch-and-cons* is used to place the name and arguments of the operation to be performed at the head of a list, returning the other operations on the list. Since the state of a deterministic object is completely determined by the history of operations performed on it, applying the operations returned in order from last to first is sufficient to locally reconstruct the object state prior to our operation. We then use the prior state to compute the result of our operation without requiring further synchronization with the other processes.

This first universal method was not very practical, a shortcoming which Herlihy soon addressed [48]. In addition, his revised universal method can be made lock-free, rather than wait-free, resulting in improved performance. In the lock-free version of this method, objects contain a shared variable holding a pointer to their current state. Processes begin by loading the current state pointer and then copying the referenced state to a local copy. The sequential operation is performed on the copy, and then if the object's shared state pointer is unchanged from its initial load, it is atomically swung to point at the updated state.

Herlihy called this the "small object protocol" because the object copying overhead is prohibitive unless the object is small enough to be copied efficiently (in, say, $O(1)$ time). He also presented a "large object protocol" which requires the programmer to manually break the object into small blocks, after which the small object protocol can be employed. (This trouble with large objects is common to many non-blocking implementations; our solution is presented in Chapter 5.)

Barnes provided the first universal non-blocking implementation method that avoids object copying [13]. He eliminates the need to store "old" object state in case of operation failure by having all threads cooperate to apply operations. For example, if the first processor begins an operation and then

halts, another processor will complete the operation of the first before applying its own. Barnes proposes to accomplish the cooperation by creating a parallel state machine for each operation, so that each thread can independently try to advance the machine from state to state and thus advance incomplete operations.[2] Although this avoids copying state, the lock-step cooperative process is extremely cumbersome and does not appear to have ever been implemented. Furthermore, it does not protect against errors in the implementation of the operations, which could cause *every* thread to fail in turn as one by one they attempt to execute a buggy operation.

Alemany and Felten [2] identified two factors hindering the performance of non-blocking algorithms to date: resources wasted by operations that fail, and the cost of data copying. Unfortunately, they proceeded to "solve" these problems by ignoring short delays and failures and using operating system support to handle delays caused by context switches, page faults, and I/O operations. This works in some situations, but obviously suffers from a bootstrapping problem as the means to implement an operating system.

Although lock-free implementations are usually assumed to be more efficient that wait-free implementations, LaMarca [64] showed experimental evidence that Herlihy's simple wait-free protocol scales ~~very~~ well on parallel machines. When more than about twenty threads are involved, the wait-free protocol becomes faster than Herlihy's lock-free small-object protocol, three OS-aided protocols of LaMarca and Alemany and Felten, and a *test-and-Compare&Swap* spin-lock.

8.3 Transactional Memory systems

Transactions are described in the database context by Gray [39], and [40] contains a thorough treatment of database issues. Hardware Transactional

[2] It is interesting to note that Barnes' cooperative method for non-blocking situation plays out in a real-time system very similarly to priority inheritance for locking synchronization.

Memory (HTM) was first proposed by Knight [63], and Herlihy and Moss coined the term "transactional memory" and proposed HTM in the context of lock-free data structures [49, 55]. The BBN Pluribus [83, 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 [84].

Transactional memory is sometimes described as an extension of Load-Linked/Store-Conditional [59] and other atomic instruction sequences. In fact, some CISC machines, such as the VAX, had complex atomic instructions such as enqueue and dequeue [27].

Of particular relevance are ***Speculative Lock Elision*** (SLE) [75] and ***Transactional Lock Removal*** (TLR) [76], 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. My research thrust differs from theirs in that I hope to free programmers from the protocol complexities of locking, not just optimize existing practice. The quantitative results presented in this thesis confirm their finding that transactional hardware can be more efficient than locks.

Martinez and Torrellas proposed ***Speculative Synchronization*** [68], 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.

The recent work on ***Transactional memory Coherence and Consistency*** (TCC) [43] 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 the UTM and LTM proposals in Chapter 6 can employ scalable cache-consistency protocols to implement transactions. TCC affirms the conclusion we draw from our own Figure 3.3: 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.

Several software transaction systems have been proposed. Some constrain the programmer and make transactions difficult to use. All have relatively high overheads, which make transactions unattractive for uniprocessor and small SMP systems. Once the number of processors is large enough, the increased parallelism that can be provided by optimistic transactions may cancel out the performance penalty of their use.

The first proposal for software transactional memory was proposed by Shavit and Touitou [82]; their system requires that all input and output locations touched by a transaction be known in advance, which limits its application. It performs at least 10 fetches and 4 stores per location accessed (not counting the loads and stores directly required by the computation). The benchmarks presented were run on between 10 and 64 processors.

Rudys and Wallach [79] proposed a copying-based transaction system to allow rollback of hostile codelets. They show an order of magnitude slowdown for field and array accesses, and 6x to 23x slowdown on their benchmarks.

Herlihy, Luchango, Moss, and Scherer's scheme [50] allows transactions to touch a dynamic set of memory locations; however the user still has to explicitly *open* every object touched before it can be used in a transaction. This implementation is based on object copying, and so has poor performance for large objects and arrays. Not including work necessary to copy objects involved in writes, they require $O(R(R+W))$ work to open R ob-

jects for reading and W objects for writing, which may be quadratic in the number of objects involved in the transaction. A list insertion benchmark that they present shows 9x slowdown over a locking scheme, although they beat the locking implementation when more than 5-10 processors are active. They present benchmark data with up to 576 threads on 72 processors.

Harris and Fraser built a software transaction system on a flat word-oriented transactional memory abstraction [44], roughly similar to simulating Herlihy's original hardware transactional memory proposal in software. This avoids problems with large objects. Performing m memory operations touching l distinct locations costs at least $m + l$ extra reads and $l + 1$ CAS operations, in addition to the reads and writes required by the computation. They appear to execute about twice as slowly as a locking implementation on some microbenchmarks. They benchmark on a 4-processor as well as a 106-processor machine; their crossover point (at which the blocking overhead of locks matches the software transaction overhead) is around 4 processors. Note that Harris and Fraser do not address the problem of concurrent non-transactional operations on locations involved in a transaction. Java synchronization allows such concurrent operations, with semantics given by the Java memory model [67]. We support these operations safely using the mechanisms presented in Chapter 3.

Programmers will be reluctant to use transactions to synchronize their code when it results in their code running more slowly on the uniprocessor and small-SMP systems that are most common today.

Herlihy and Moss [49] 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. In Section 6.4 we presented just such a system.

```
const myDirectory == object oneEntryDirectory
  export Store, Lookup
  monitor
    var name : String
    var AnObject : Any

    operation Store [ n : String, o : Any ]
      name ← n
      AnObject ← o
    end Store
    function Lookup [ n : String ] → [ o : Any ]
      if n = name
        then o ← AnObject
        else o ← nil
      end if
    end Lookup

    initially
      name ← nil
      AnObject ← nil
    end initially

  end monitor
end oneEntryDirectory
```

Figure 8.1: A directory object in Emerald, from [17], illustrating the use of monitor synchronization.

```
class Account {

  int balance = 0;

  atomic int deposit(int amt) {
    int t = this.balance;
    t = t + amt;
    this.balance = t;
    return t;
  }

  atomic int readBalance() {
    return this.balance;
  }

  atomic int withdraw(int amt) {
    int t = this.balance;
    t = t - amt;
    this.balance = t;
    return t;
  }

}
```

Figure 8.2: A simple bank account object, adapted from [29], illustrating the use of the `atomic` modifier.

8.4 Language-level approaches to synchronization

Our work on integrating transactions into the Java programming language is related to prior work on integrating synchronization mechanisms for multiprogramming, and in particular, to prior work on synchronization in an object-oriented framework.

The Emerald system [17, 61] introduced *monitored objects* for synchronization. Emerald code to implement a simple directory object is shown in Figure 8.1. Each object is associated with Hoare-style monitor, which provides mutual exclusion and process signaling. Each Emerald object is divided into a monitored part and a non-monitored part. Variables declared in the monitored part are shared, and access to them from methods in the non-monitored part is prohibited—although non-monitored methods may call monitored methods to effect the access. Methods in the monitored part acquire the monitor lock associated with the receiver object before entry and release it on exit, providing for mutual exclusion and safe update of the shared variables. Monitored objects naturally integrate synchronization into the object model.

Unlike Emerald monitored objects, where methods can only acquire the monitor of their receiver and where restricted access to shared variables is enforced by the compiler, Java implements a loose variant where any monitor may be explicitly acquired and no shared variable protection exists. As a default, however, Java methods declared with the `synchronized` keyword behave like Emerald monitored methods, ensuring that the monitor lock of their receiver is held during execution.

Java's synchronization primitives arguably allow for more efficient concurrent code than Emerald's—for example, Java objects can use multiple locks to protect disjoint sets of fields, and coarse-grain locks can be used which protect multiple objects—but Java is also more prone to programmer error. However, even Emerald's restrictive monitored objects are not sufficient to prevent data races. As a simple example, imagine that an object

provided two monitored methods `read` and `write` which accessed a shared variable. Non-monitored code can call `read`, increment the value returned, and then call `write`, creating a classic race condition scenario. The atomicity of the parts is not sufficient to guarantee atomicity of the whole [29].

This suggests that a better model for synchronization in object-oriented systems is *atomicity*. Figure 8.2 shows Java extended with an `atomic` keyword to implement an object representing a bank account. Rather than explicitly synchronizing on locks, I simply require that the methods marked `atomic` execute atomically with respect to other threads in the system; that is, that every execution of the program computes the same result as some execution where all atomic methods were run *in isolation* at a certain point in time, called the ***linearization point***, between their invocation and return. Note that atomic methods invoked directly or indirectly from an atomic method are subsumed by it: if the outermost method appears atomic, then by definition all inner method invocations will also appear atomic. Flanagan and Qadeer provide a more formal semantics in [29]. Atomic methods can be analyzed using sequential reasoning techniques, which significantly simplifies reasoning about program correctness.

Atomic methods can be implemented using locks. A simple if inefficient implementation would simply acquire a single global lock during the execution of every atomic method. Flanagan and Qadeer [29] present a more sophisticated technique that proves that a given implementation using standard Java monitors correctly guarantees method atomicity.

The transaction implementations presented in this thesis will use non-blocking synchronization to implement atomic methods.

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."

Lewis Carroll, *Alice's Adventures in Wonderland*

Chapter 9

Conclusion

In this thesis I have shown that it is possible to implement an efficient strongly-atomic software transaction system, and that non-blocking transactions can be used in applications beyond synchronization, including fault tolerance and backtracking search. I have presented implementation details to address the practical problems of building such a system. I have argued the transactions should not be subject to limits on size or duration, and presented both software and hardware implementations free of such restrictions. Finally, the low overhead of my software system allows it to be profitably combined with a hardware transaction system; I have shown how this hybrid yields fast execution of short and small transactions while allowing fallback to software for large or complicated transactions.

There is no escape: parallel systems are in our future. However, programming them does not have to be as fraught as it is presently. I believe that transactions provide a programmer-friendly model of concurrency which eliminates many potential pitfalls of our current locking-based methodologies.

In this thesis I have presented several designs for efficient transaction systems that will enable us to take advantage of the transactional programming model. The software-only system runs on current hardware, and LTM and UTM indicate possible directions for future hardware. However, there

are challenges and design decisions remaining; how should I/O be handled? What are the proper semantics for nested transactions? What loophole mechanisms are necessary to allow information about a transaction's progress to escape?

I believe hybrid systems are the best answer to these challenges; they combine the inherent speed of hardware systems with the flexibility of software, allowing novel solutions to be attempted without requiring that design decisions be cast in silicon. The flag-based software transaction system described in this thesis imposes very low overhead, allowing transactional programming to get off the ground without hardware support in the near term, while later supporting the development of new transactional models and methodologies as part of a hybrid system.

Designing correct transaction systems is not easy, however. In the appendix you will find a Promela model of my software transaction system. Automated verification was essential when designing and debugging the system, uncovering via exhaustive enumeration race conditions much too subtle for me to discover by other means. I believe any credible transaction system must be buttressed by formal verification.

Essentially, all models are
wrong, but some are useful.

George E. P. Box, *Empirical Model-Building and Response Surfaces*

Appendix A

Model-checking the software implementation

My work on both software and hardware transaction systems has reiterated the difficulty of creating correct implementations of concurrent and fault-tolerant constructs. Automatic model checking is a prerequisite to achieving confidence in the design and implementation. Versions of the software transaction system have been modeled in Promela using SPIN 4.1.0 and verified on an SGI 64-processor MIPS machine with 16G of main memory.

Sequences of transactional and non-transactional load and store operations were checked using two concurrent processes, and all possible interleavings were found to produce results consistent with the semantic atomicity of the transactions. Several test scripts were run against the model using separate processors of the verification machine (SPIN cannot otherwise exploit SMP). Some representative costs include:

- testing two concurrent `writeT` operations (including "false flag" conditions) against a single object required 3.8×10^6 states and 170M memory;

- testing sequences of transactional and non-transactional reads and writes against two objects (checking that all views of the two objects

were consistent) required 4.6×10^6 states and 207M memory; and

- testing a pair of concurrent increments at values bracketing the FLAG value to 99.8% coverage of the state space required 7.6×10^7 states and 4.3G memory. Simultaneously model-checking a range of values caused the state space explosion in this case.

SPIN's unreachable code reporting was used to ensure that our test cases exercised every code path, although this doesn't guarantee that every interesting interaction is checked.

In the process one bug in SPIN was discovered[1] and a number of subtle race conditions in the model were discovered and corrected. These included a number of modeling artifacts. In particular, we were extremely aggressive about reference-counting and deallocating objects in order to control the size of the state space, and this proved difficult to do correctly. We also discovered some subtle-but-legitimate race conditions in the transactions algorithm. For example:

- A race allowed conflicting readers to be created while a writer was inside `ensureWriter` creating a new version object.

- Allowing already-committed version objects to be mutated when `writeT` or `writeNT` was asked to store a "false flag" produced races between `ensureWriter` and `copyBackField`. The code that was expected to manage these races had unexpected corner cases.

- Using a bitmask to provide per-field granularity to the list of readers proved unmanageable as there were three-way races between the bitmask, the readers list, and the version tree.

In addition, the model-in-progress proved a valuable design tool, as portions of the algorithm could readily be mechanically checked to validate (or dis-

[1] Breadth-first search of atomic regions was performed incorrectly in SPIN 4.0.7; this was fixed in SPIN 4.1.0.

credit) the designer's reasoning about the concurrent system. Humans do not excel at exhaustive state-space exploration.

SPIN is not particularly suited to checking models with dynamic allocation and deallocation. In particular, it considers the location of objects as part of the state space, and allocating object A before object B produces a different state than if object B were allocated first. This artificially enlarges the state space. A great deal of effort was expended tweaking the model to approach a canonical allocation ordering. A better solution to this problem would allow larger model instances to be checked.

A.1 Promela primer

A concise Promela reference is available at http://spinroot.com/spin/Man/Quick.html; we will here attempt to summarize just enough of the language to allow the model we've presented in this thesis to be understood.

Promela syntax is C-like, with the same lexical and commenting conventions. Statements are separated by either a semicolon, or, equivalently, an arrow. The arrow is typically used to separate a guard expression from the statements it is guarding.

The program counter moves past a statement only if the statement is *enabled*. Most statements, including assignments, are always enabled. A statement consisting only of an expression is enabled iff the expression is true (non-zero). Our model uses three basic Promela statements: selection, repetition, and `atomic`.

The selection statement

```
if
:: guard -> statements
...
:: guard -> statements
fi
```

selects one among its options and executes it. An option can be selected iff its first statement (the guard) is enabled. The special guard `else` is enabled

iff all other guards are not.

The repetition statement

```
do
:: statements
...
:: statements
fi
```

is similar: one among its enabled statements is selected and executed, and then the process is repeated (with a different statement possibly being selected each time) until control is explicitly transferred out of the loop with a break or goto.

Finally,

```
atomic { statements }
```

executes the given statements in one indivisible step. For the purposes of this model, a d_step block is functionally identical. Outside atomic or d_step blocks, Promela allows interleaving before and after every statement, but statements are indivisible.

Functions as specified in this model are similar to C macros: every parameter is potentially both an input *and* an output. Calls to functions with names starting with move are simple assignments; they've been turned into macros so that reference counting can be performed.

A.2 Spin model for software transaction system

The complete Spin 4.1.0 model for the FLEX software transaction system is presented here. It may also be downloaded from http://flex-master.csail.mit.edu/Harpoon/swx.pml.

```
/*************************************************************************
 * Very detailed model of software transaction code.
 * Checking for safety and correctness properties.  Not too worried about
 * liveness at the moment.
 *
 * (C) 2006 C. Scott Ananian <cananian@alumni.princeton.edu>
```

```
 ************************************************************************/

/* CURRENT ISSUES:
 *  none known.
 */
/* MINOR ISSUES:
 * 1) use smaller values for FLAG and NIL to save state space?
 */

/* Should use Spin 4.1.0, for correct nested-atomic behavior. */

#define REFCOUNT

#define NUM_OBJS 2
#define NUM_VERSIONS 6 /* each obj: committed and waiting version, plus nonce
                        * plus addition nonce for NT copyback in test3 */
#define NUM_READERS 4 /* both 'read' trans reading both objs */
#define NUM_TRANS 5 /* two 'real' TIDs, plus 2 outstanding TIDs for
     * writeNT(FLAG) [test3], plus perma-aborted TID. */
#define NUM_FIELDS 2

#define NIL 255 /* special value to represent 'alloc impossible', etc. */
#define FLAG 202 /* special value to represent 'not here' */

typedef Object {
  byte version;
  byte readerList; /* we do LL and CAS operations on this field */
  pid fieldLock[NUM_FIELDS]; /* we do LL operations on fields */
  byte field[NUM_FIELDS];
};
typedef VersiOn { /* 'Version' misspelled because spin #define's it. */
  byte owner;
  byte next;
  byte field[NUM_FIELDS];
#ifdef REFCOUNT
  byte ref; /* reference count */
#endif /* REFCOUNT */
};
typedef ReaderList {
  byte transid;
  byte next;
#ifdef REFCOUNT
  byte ref; /* reference count */
#endif /* REFCOUNT */
};
mtype = { waiting, committed, aborted };
typedef TransID {
  mtype status;
#ifdef REFCOUNT
  byte ref; /* reference count */
#endif /* REFCOUNT */
};
```

```
Object object[NUM_OBJS];
VersiOn version[NUM_VERSIONS];
ReaderList readerlist[NUM_READERS];
TransID transid[NUM_TRANS];
byte aborted_tid; /* global variable; 'perma-aborted' */

/* -------------------------- alloc.pml ---------------------- */
mtype = { request, return };

inline manager(NUM_ITEMS, allocchan) {
  chan pool = [NUM_ITEMS] of { byte };
  chan client;
  byte nodenum;
  /* fill up the pool with node identifiers */
  d_step {
    i=0;
    do
    :: i<NUM_ITEMS -> pool!!i; i++
    :: else -> break
    od;
  }
end:
  do
  :: allocchan?request(client,_) ->
     if
     :: empty(pool) -> assert(0); client!NIL /* deny */
     :: nempty(pool) ->
pool?nodenum;
client!nodenum;
nodenum=0
     fi
  :: allocchan?return(client,nodenum) ->
     pool!!nodenum; /* sorted, to reduce state space */
     nodenum=0
  od
}

chan allocObjectChan = [0] of { mtype, chan, byte };
active proctype ObjectManager() {
  atomic {
    byte i;
    manager(NUM_OBJS, allocObjectChan)
  }
}
chan allocVersionChan = [0] of { mtype, chan, byte };
active proctype VersionManager() {
  atomic {
    byte i=0;
    d_step {
      do
      :: i<NUM_VERSIONS ->
 version[i].owner=NIL; version[i].next=NIL;
```

```
 version[i].field[0]=FLAG; version[i].field[1]=FLAG;
 assert(NUM_FIELDS==2);
 i++
      :: else -> break
      od;
    }
    manager(NUM_VERSIONS, allocVersionChan)
  }
}
chan allocReaderListChan = [0] of { mtype, chan, byte };
active proctype ReaderListManager() {
  atomic {
    byte i=0;
    d_step {
      do
      :: i<NUM_READERS ->
 readerlist[i].transid=NIL; readerlist[i].next=NIL;
 i++
      :: else -> break
      od;
    }
    manager(NUM_READERS, allocReaderListChan)
  }
}
chan allocTransIDChan = [0] of { mtype, chan, byte };
active proctype TransIDManager() {
  atomic {
    byte i=0;
    d_step {
      do
      :: i<NUM_TRANS -> transid[i].status=waiting; i++
      :: else -> break
      od;
    }
    manager(NUM_TRANS, allocTransIDChan)
  }
}

inline alloc(allocchan, retval, result) {
  result = NIL;
  do
  :: result != NIL -> break
  :: else -> allocchan!request(retval,0) ; retval ? result
  od;
  skip /* target of break. */
}
inline free(allocchan, retval, result) {
  allocchan!return(retval,result)
}
inline allocObject(retval, result) {
  atomic {
    alloc(allocObjectChan, retval, result);
```

```
    d_step {
      object[result].version = NIL;
      object[result].readerList = NIL;
      object[result].field[0] = 0;
      object[result].field[1] = 0;
      object[result].fieldLock[0] = _thread_id;
      object[result].fieldLock[1] = _thread_id;
      assert(NUM_FIELDS==2); /* else ought to initialize more fields */
    }
  }
}
inline allocTransID(retval, result) {
  atomic {
    alloc(allocTransIDChan, retval, result);
    d_step {
      transid[result].status = waiting;
#ifdef REFCOUNT
      transid[result].ref = 1;
#endif /* REFCOUNT */
    }
  }
}
inline moveTransID(dst, src) {
  atomic {
#ifdef REFCOUNT
    _free = NIL;
    if
    :: (src!=NIL) ->
       transid[src].ref++
    :: else
    fi;
    if
    :: (dst!=NIL) ->
       transid[dst].ref--;
       if
       :: (transid[dst].ref==0) -> _free=dst
       :: else
       fi
    :: else
    fi;
#endif /* REFCOUNT */
    dst = src;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeTransID(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}
proctype freeTransID(byte result; chan retval) {
```

```
  chan _retval = [0] of { byte };
  atomic {
#ifdef REFCOUNT
    assert(transid[result].ref==0);
#endif /* REFCOUNT */
    transid[result].status = waiting;
    free(allocTransIDChan, _retval, result)
    retval!0; /* done */
  }
}
inline allocVersion(retval, result, a_transid, tail) {
  atomic {
    alloc(allocVersionChan, retval, result);
    d_step {
#ifdef REFCOUNT
      if
      :: (a_transid!=NIL) -> transid[a_transid].ref++;
      :: else
      fi;
      if
      :: (tail!=NIL) -> version[tail].ref++;
      :: else
      fi;
      version[result].ref = 1;
#endif /* REFCOUNT */
      version[result].owner = a_transid;
      version[result].next = tail;
      version[result].field[0] = FLAG;
      version[result].field[1] = FLAG;
      assert(NUM_FIELDS==2); /* else ought to initialize more fields */
    }
  }
}
inline moveVersion(dst, src) {
  atomic {
#ifdef REFCOUNT
    _free = NIL;
    if
    :: (src!=NIL) ->
       version[src].ref++
    :: else
    fi;
    if
    :: (dst!=NIL) ->
       version[dst].ref--;
       if
       :: (version[dst].ref==0) -> _free=dst
       :: else
       fi
    :: else
    fi;
#endif /* REFCOUNT */
```

```
    dst = src;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeVersion(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}
proctype freeVersion(byte result; chan retval) {
  chan _retval = [0] of { byte };
  byte _free;
  atomic { /* zero out version structure */
#ifdef REFCOUNT
    assert(version[result].ref==0);
#endif /* REFCOUNT */
    moveTransID(version[result].owner, NIL);
    moveVersion(version[result].next, NIL);
    version[result].field[0] = FLAG;
    version[result].field[1] = FLAG;
    assert(NUM_FIELDS==2);
    free(allocVersionChan, _retval, result)
    retval!0; /* done */
  }
}

inline allocReaderList(retval, result, head, tail) {
  atomic {
    assert(head!=NIL);
    alloc(allocReaderListChan, retval, result);
    d_step {
#ifdef REFCOUNT
      readerlist[result].ref = 1;
      transid[head].ref++;
      if
      :: (tail!=NIL) -> readerlist[tail].ref++
      :: else
      fi;
#endif /* REFCOUNT */
      readerlist[result].transid = head;
      readerlist[result].next = tail;
    }
  }
}
inline moveReaderList(dst, src) {
  atomic {
#ifdef REFCOUNT
    _free = NIL;
    if
    :: (src!=NIL) ->
       readerlist[src].ref++
```

```
    :: else
    fi;
    if
    :: (dst!=NIL) ->
       readerlist[dst].ref--;
       if
       :: (readerlist[dst].ref==0) -> _free=dst
       :: else
       fi
    :: else
    fi;
#endif /* REFCOUNT */
    dst = src;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeReaderList(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif
  }
}
proctype freeReaderList(byte result; chan retval) {
  chan _retval = [0] of { byte };
  byte _free;
  atomic {
#ifdef REFCOUNT
    assert(readerlist[result].ref==0);
#endif /* REFCOUNT */
    moveTransID(readerlist[result].transid, NIL);
    moveReaderList(readerlist[result].next, NIL);
    free(allocReaderListChan, _retval, result)
    retval!0; /* done */
  }
}

/* -------------------------- atomic.pml ---------------------- */
inline DCAS(loc1, oval1, nval1, loc2, oval2, nval2, st) {
  d_step {
    if
    :: (loc1==oval1) && (loc2==oval2) ->
       loc1=nval1;
       loc2=nval2;
       st=true
    :: else ->
       st=false
    fi
  }
}
inline CAS(loc1, oval1, nval1, st) {
  d_step {
    if
```

```
    :: (loc1==oval1) ->
       loc1=nval1;
       st=true
    :: else ->
       st=false
    fi
  }
}
inline CAS_Version(loc1, oval1, nval1, st) {
  atomic {
    _free = NIL;
    if
    :: (loc1==oval1) ->
#ifdef REFCOUNT
       if
       :: (nval1!=NIL) -> version[nval1].ref++;
       :: else
       fi;
       if
       :: (oval1!=NIL) -> version[oval1].ref--;
  if
  :: (version[oval1].ref==0) -> _free = oval1
  :: else
  fi
       :: else
       fi;
#endif /* REFCOUNT */
       loc1=nval1;
       st=true
    :: else ->
       st=false
    fi;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeVersion(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}
inline CAS_Reader(loc1, oval1, nval1, st) {
  atomic {
    /* save oval1, as it could change as soon as we leave the d_step */
    _free = NIL;
    if
    :: (loc1==oval1) ->
#ifdef REFCOUNT
       if
       :: (nval1!=NIL) -> readerlist[nval1].ref++;
       :: else
       fi;
```

```
       if
       :: (oval1!=NIL) -> readerlist[oval1].ref--;
  if
  :: (readerlist[oval1].ref==0) -> _free = oval1
  :: else
  fi
       :: else
       fi;
#endif /* REFCOUNT */
       loc1=nval1;
       st=true
    :: else ->
       st=false
    fi;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeReaderList(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}

/* ---------------- end atomic.pml ------------------ */

mtype = { kill_writers, kill_all };
mtype = { success, saw_race, saw_race_cleanup, false_flag };

inline tryToAbort(t) {
  assert(t!=NIL);
  CAS(transid[t].status, waiting, aborted, _);
  assert(transid[t].status==aborted || transid[t].status==committed)
}
inline tryToCommit(t) {
  assert(t!=NIL);
  CAS(transid[t].status, waiting, committed, _);
  assert(transid[t].status==aborted || transid[t].status==committed)
}
inline copyBackField(o, f, mode, st) {
  _nonceV=NIL; _ver = NIL; _r = NIL; st = success;
  /* try to abort each version.  when abort fails, we've got a
   * committed version. */
  do
  :: moveVersion(_ver, object[o].version);
     if
     :: (_ver==NIL) ->
st = saw_race; break /* someone's done the copyback for us */
     :: else
     fi;
      /* move owner to local var to avoid races (owner set to NIL behind
       * our back) */
```

```
      _tmp_tid=NIL;
      moveTransID(_tmp_tid, version[_ver].owner);
      if
      :: (_tmp_tid==NIL) ->
break; /* found a committed version */
      :: else
      fi;
      tryToAbort(_tmp_tid);
      if
      :: (transid[_tmp_tid].status==committed) ->
moveTransID(_tmp_tid, NIL);
moveTransID(version[_ver].owner, NIL); /* opportunistic free */
moveVersion(version[_ver].next, NIL); /* opportunistic free */
break /* found a committed version */
      :: else
      fi;
      /* link out an aborted version */
      assert(transid[_tmp_tid].status==aborted);
      CAS_Version(object[o].version, _ver, version[_ver].next, _);
      moveTransID(_tmp_tid, NIL);
  od;
  /* okay, link in our nonce.  this will prevent others from doing the
   * copyback. */
  if
  :: (st==success) ->
     assert (_ver!=NIL);
     allocVersion(_retval, _nonceV, aborted_tid, _ver);
     CAS_Version(object[o].version, _ver, _nonceV, _cas_stat);
     if
     :: (!_cas_stat) ->
st = saw_race_cleanup
     :: else
     fi
  :: else
  fi;
  /* check that no one's beaten us to the copy back */
  if
  :: (st==success) ->
     if
     :: (object[o].field[f]==FLAG) ->
_val = version[_ver].field[f];
if
:: (_val==FLAG) -> /* false flag... */
   st = false_flag /* ...no copy back needed */
:: else -> /* not a false flag */
   d_step { /* could be DCAS */
     if
     :: (object[o].version == _nonceV) ->
object[o].fieldLock[f] = _thread_id;
object[o].field[f] = _val;
     :: else /* hmm, fail.  Must retry. */
st = saw_race_cleanup /* need to clean up nonce */
```

```
     fi
   }
fi
     :: else /* may arrive here because of readT, which doesn't set _val=FLAG*/
st = saw_race_cleanup /* need to clean up nonce */
     fi
  :: else /* !success */
  fi;

  /* always kill readers, whether successful or not.  This ensures that we
   * make progress if called from writeNT after a readNT sets readerList
   * non-null without changing FLAG to _val (see immediately above; st will
   * equal saw_race_cleanup in this scenario). */
  if
  :: (mode == kill_all) ->
     do /* kill all readers */
     :: moveReaderList(_r, object[o].readerList);
if
:: (_r==NIL) -> break
:: else
fi;
tryToAbort(readerlist[_r].transid);
/* link out this reader */
CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _);
     od;
  :: else /* no more killing needed. */
  fi;

  /* finally, clean up our mess. */
  moveVersion(_ver, NIL);
  if
  :: (st == saw_race_cleanup || st == success || st == false_flag) ->
     CAS_Version(object[o].version, _nonceV, version[_nonceV].next, _);
     moveVersion(_nonceV, NIL);
     if
     :: (st==saw_race_cleanup) -> st=saw_race
     :: else
     fi
  :: else
  fi;
  /* done */
  assert(_nonceV==NIL);
}

inline readNT(o, f, v) {
  do
  :: v = object[o].field[f];
     if
     :: (v!=FLAG) -> break /* done! */
     :: else
     fi;
     copyBackField(o, f, kill_writers, _st);
```

```
     if
     :: (_st==false_flag) ->
v = FLAG;
break
     :: else
     fi
  od
}
inline writeNT(o, f, nval) {
  if
  :: (nval != FLAG) ->
     do
     ::
atomic {
  if /* this is a LL(readerList)/SC(field) */
  :: (object[o].readerList == NIL) ->
     object[o].fieldLock[f] = _thread_id;
     object[o].field[f] = nval;
     break /* success! */
  :: else
  fi
}
/* unsuccessful SC */
copyBackField(o, f, kill_all, _st)
/* ignore return status */
     od
  :: else -> /* create false flag */
     /* implement this as a short *transactional* write.  this may be slow,
      * but it greatly reduces the race conditions we have to think about. */
     do
     :: allocTransID(_retval, _writeTID);
ensureWriter(_writeTID, o, _tmp_ver);
checkWriteField(o, f);
writeT(_tmp_ver, f, nval);
tryToCommit(_writeTID);
moveVersion(_tmp_ver, NIL);
if
:: (transid[_writeTID].status==committed) ->
   moveTransID(_writeTID, NIL);
   break /* success! */
:: else ->/* try again */
   moveTransID(_writeTID, NIL)
fi
     od
  fi;
}
inline readT(tid, o, f, ver, result) {
  do
  ::
     /* we should always either be on the readerlist or aborted here */
     atomic { /* complicated assertion; evaluate atomically */
       if
```

```
       :: (transid[tid].status == aborted) -> skip /* okay then */
       :: else ->
  assert (transid[tid].status == waiting);
  _r = object[o].readerList;
  do
  :: (_r==NIL || readerlist[_r].transid==tid) -> break
  :: else -> _r = readerlist[_r].next
  od;
  assert (_r!=NIL); /* we're on the list */
  _r = NIL /* back to normal */
       fi
     }
     /* okay, sanity checking done -- now let's get to work! */
     result = object[o].field[f];
     if
     :: (result==FLAG) ->
if
:: (ver!=NIL) ->
   result = version[ver].field[f];
   break /* done! */
:: else ->
   findVersion(tid, o, ver);
   if
   :: (ver==NIL) -> /* use value from committed version */
      assert (_r!=NIL);
      result = version[_r].field[f]; /* false flag? */
      moveVersion(_r, NIL);
      break /* done */
   :: else /* try, try, again */
   fi
fi
     :: else -> break /* done! */
     fi
  od
}
inline writeT(ver, f, nval) {
  /* easy enough: */
  version[ver].field[f] = nval;
}

/* make sure 'tid' is on reader list. */
inline ensureReaderList(tid, o) {
  /* add yourself to readers list. */
  _rr = NIL; _r = NIL;
  do
  :: moveReaderList(_rr, object[o].readerList); /* first_reader */
     moveReaderList(_r, _rr);
     do
     :: (_r==NIL) ->
break /* not on the list */
     :: (_r!=NIL && readerlist[_r].transid==tid) ->
break /* on the list */
```

```
     :: else ->
/* opportunistic free? */
if
:: (_r==_rr && transid[readerlist[_r].transid].status != waiting) ->
   CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _)
   if
   :: (_cas_stat) -> moveReaderList(_rr, readerlist[_r].next)
   :: else
   fi
:: else
fi;
/* keep looking */
moveReaderList(_r, readerlist[_r].next)
     od;
     if
     :: (_r!=NIL) ->
break /* on the list; we're done! */
     :: else ->
/* try to put ourselves on the list. */
assert(tid!=NIL && _r==NIL);
allocReaderList(_retval, _r, tid, _rr);
CAS_Reader(object[o].readerList, _rr, _r, _cas_stat);
if
:: (_cas_stat) ->
   break /* we're on the list */
:: else
fi
/* failed to put ourselves on the list, retry. */
     fi
  od;
  moveReaderList(_rr, NIL);
  moveReaderList(_r, NIL);
  /* done. */
}

/* look for a version read/writable by 'tid'.  Returns:
 *   ver!=NIL -- ver is the 'waiting' version for 'tid'. _r == NIL.
 *   ver==NIL, _r != NIL -- _r is the first committed version in the chain.
 *   ver==NIL, _r == NIL -- there are no commited versions for this object
 *                          (i.e. object[o].version==NIL)
 */
inline findVersion(tid, o, ver) {
  assert(tid!=NIL);
  _r = NIL; ver = NIL; _tmp_tid=NIL;
  do
  :: moveVersion(_r, object[o].version);
     if
     :: (_r==NIL) -> break /* no versions. */
     :: else
     fi;
     moveTransID(_tmp_tid, version[_r].owner);/*use local copy to avoid races*/
     if
```

```
     :: (_tmp_tid==tid) ->
ver = _r; /* found a version: ourself! */
_r = NIL; /* transfer owner of the reference to ver, without ++/-- */
break
     :: (_tmp_tid==NIL) ->
/* perma-committed version.  Return in _r. */
moveVersion(version[_r].next, NIL); /* opportunistic free */
break
     :: else -> /* strange version.  try to kill it. */
/* ! could double-check that our own transid is not aborted here. */
tryToAbort(_tmp_tid);
if
:: (transid[_tmp_tid].status==committed) ->
   /* committed version.  Return this in _r. */
   moveTransID(version[_r].owner, NIL); /* opportunistic free */
   moveVersion(version[_r].next, NIL); /* opportunistic free */
   break /* no need to look further. */
:: else ->
   assert (transid[_tmp_tid].status==aborted);
   /* unlink this useless version */
   CAS_Version(object[o].version, _r, version[_r].next, _)
   /* repeat */
fi
     fi
  od;
  moveTransID(_tmp_tid, NIL); /* free tmp transid copy */
  assert (ver!=NIL -> _r == NIL : 1)
}

inline ensureReader(tid, o, ver) {
  assert(tid!=NIL);
  /* make sure we're on the readerlist */
  ensureReaderList(tid, o)
  /* now kill any transactions associated with uncommitted versions, unless
   * the transaction is ourself! */
  findVersion(tid, o, ver);
  /* don't care about which committed version to use, at the moment. */
  moveVersion(_r, NIL);
}

/* per-object, before write. */
/* returns NIL in ver to indicate suicide. */
inline ensureWriter(tid, o, ver) {
  assert(tid!=NIL);
  /* Same beginning as ensureReader */
  ver = NIL; _r = NIL; _rr = NIL;
  do
  :: assert (ver==NIL);
     findVersion(tid, o, ver);
     if
     :: (ver!=NIL) -> break /* found a writable version for us */
     :: (ver==NIL && _r==NIL) ->
```

```
/* create and link a fully-committed root version, then
 * use this as our base. */
allocVersion(_retval, _r, NIL, NIL);
CAS_Version(object[o].version, NIL, _r, _cas_stat)
     :: else ->
_cas_stat = true
     fi;
     if
     :: (_cas_stat) ->
/* so far, so good. */
assert (_r!=NIL);
assert (version[_r].owner==NIL ||
transid[version[_r].owner].status==committed);
/* okay, make new version for this transaction. */
assert (ver==NIL);
allocVersion(_retval, ver, tid, _r);
/* want copy of committed version _r.  Race here because _r can be
 * written to under peculiar circumstances, namely: _r has
 * non-flag value, non-flag value is copied back to parent,
 * flag_value is written to parent -- this forces flag_value to
 * be written to committed version. */
/* IF WRITES ARE ALLOWED TO COMMITTED VERSIONS, THERE IS A RACE HERE.
 * But our implementation of false_flag writes at the moment does
 * not permit *any* writes to committed versions. */
version[ver].field[0] = version[_r].field[0];
version[ver].field[1] = version[_r].field[1];
assert(NUM_FIELDS==2); /* else ought to initialize more fields */
CAS_Version(object[o].version, _r, ver, _cas_stat);
moveVersion(_r, NIL); /* free _r */
if
:: (_cas_stat) ->
   /* kill all readers (except ourself) */
   /* note that all changes have to be made from the front of the
    * list, so we unlink ourself and then re-add us. */
   do
   :: moveReaderList(_r, object[o].readerList);
      if
      :: (_r==NIL) -> break
      :: (_r!=NIL && readerlist[_r].transid!=tid)->
 tryToAbort(readerlist[_r].transid)
      :: else
      fi;
      /* link out this reader */
      CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _)
   od;
   /* okay, all pre-existing readers dead & gone. */
   assert(_r==NIL);
   /* link us back in. */
   ensureReaderList(tid, o);
   break
:: else
fi;
```

```
/* try again */
     :: else
     fi;
     /* try again from the top */
     moveVersion(ver, NIL)
  od;
  /* done! */
  assert (_r==NIL);
}
/* per-field, before read. */
inline checkReadField(o, f) {
  /* do nothing: no per-field read stats are kept. */
  skip
}
/* per-field, before write. */
inline checkWriteField(o, f) {
  _r = NIL; _rr = NIL;
  do
  ::
     /* set write flag, if not already set */
     _val = object[o].field[f];
     if
     :: (_val==FLAG) ->
break; /* done! */
     :: else
     fi;
     /* okay, need to set write flag. */
     moveVersion(_rr, object[o].version);
     moveVersion(_r, _rr);
     assert (_r!=NIL);
     do
     :: (_r==NIL) -> break /* done */
     :: else ->
object[o].fieldLock[f] = _thread_id;
if
/* this next check ensures that concurrent copythroughs don't stomp
 * on each other's versions, because the field will become FLAG
 * before any other version will be written. */
:: (object[o].field[f]==_val) ->
   if
   :: (object[o].version==_rr) ->
      atomic {
if
:: (object[o].fieldLock[f]==_thread_id) ->
   version[_r].field[f] = _val;
:: else -> break /* abort */
fi
      }
   :: else -> break /* abort */
   fi
:: else -> break /* abort */
fi;
```

```
moveVersion(_r, version[_r].next) /* on to next */
     od;
     if
     :: (_r==NIL) ->
/* field has been successfully copied to all versions */
atomic {
  if
  :: (object[o].version==_rr) ->
     assert(object[o].field[f]==_val ||
    /* we can race with another copythrough and that's okay;
     * the locking strategy above ensures that we're all
     * writing the same values to all the versions and not
     * overwriting anything. */
    object[o].field[f]==FLAG);
     object[o].fieldLock[f]=_thread_id;
     object[o].field[f] = FLAG;
     break; /* success!  done! */
  :: else
  fi
}
     :: else
     fi
     /* retry */
  od;
  /* clean up */
  moveVersion(_r, NIL);
  moveVersion(_rr, NIL);
}
```

Bibliography

Each citation is followed by a bracketed list of the pages on which it appears. DOI names can be resolved at http://doi.org.

[1] San Jose, California, October 5–9 2002. ACM Press. ISBN 1-58113-574-2. doi: 10.1145/605397.605400. [on pp. 185 and 186.]

[2] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC '92)*, pages 125–134, Vancouver, British Columbia, Canada, August 1992. ACM Press. ISBN 0-89791-495-3. doi: 10.1145/135419.135446. [on p. 144.]

[3] C. Scott Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1999. MIT Technical Report MIT-LCS-TR-801. [on pp. 63 and 69.]

[4] C. Scott Ananian. The FLEX compiler project. http://flex-compiler.csail.mit.edu/, 2007. [on pp. 33 and 63.]

[5] C. Scott Ananian and Martin Rinard. Efficient object-based software transactions. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, CA, October 2005. [on p. 63.]

[6] C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, pages 59–68, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-647-1. doi: 10.1145/780732.780741. [on p. 63.]

[7] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 316–327, San Francisco, California, February 2005. IEEE Computer Society. doi: 10.1109/HPCA.2005.41. [on pp. 22, 28, 38, 63, 109, and 128.]

[8] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. *IEEE Micro Special Issue: Top Picks from Computer Architecture Conferences*, January/February 2006. doi: 10.1109/MM.2006.26. [on p. 63.]

[9] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software–Practice and Experience*, 19(2):171–183, February 1989. doi: 10.1002/spe.4380190206. [on p. 89.]

[10] Andrew W. Appel and Guy J. Jacobson. The world's fastest Scrabble program. *Communications of the ACM*, 31(5):572–578, May 1988. doi: 10.1145/42411.42420. [on p. 26.]

[11] Henry G. Baker. Shallow binding makes functional arrays fast. *ACM SIGPLAN Notices*, 26(8):145–147, August 1991. ISSN 0362-1340. doi: 10.1145/122598.122614. [on p. 102.]

[12] Henry G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359566. [on p. 102.]

[13] Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 261–270. ACM Press, June 1993. ISBN 0-89791-599-2. doi: 10.1145/165231.165265. [on p. 143.]

[14] William S. Beebee and Martin Rinard. An implementation of scoped memory for Real-Time Java. In *Proceedings of Embedded Software: First International Workshop (EMSOFT 2001)*, volume 2211/2001 of *LNCS*, Tahoe City, CA, October 2001. [on p. 63.]

[15] William S. Beebee, Jr. Region-based memory management for Real-Time Java. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, September 2001. [on p. 63.]

[16] A. Bensoussan, C.T. Clingen, and R.C. Daley. The Multics virtual memory: Concepts and design. *CACM*, 15(5):308–318, May 1972. doi: 10.1145/355602.361306. [on p. 119.]

[17] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 78–86. ACM Press, September 1986. ISBN 0-89791-204-7. doi: 10.1145/28697.28706. [on pp. 29, 148, and 150.]

[18] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, Madison, Wisconsin, June 2005. URL http://www.ece.wisc.edu/~wddd/2005/papers/WDDD05_blundell.pdf. [on pp. 17 and 133.]

[19] Hans Boehm, Alan Demers, and Mark Weiser. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/, 1991. [on p. 70.]

[20] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr, and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-662-5. doi: 10.1145/781131.781168. [on p. 63.]

[21] Tyng-Ruey Chuang. A randomized implementation of multiple functional arrays. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming (LFP)*, pages 173–184. ACM Press, June 1994. ISBN 0-89791-643-3. doi: 10.1145/182409.156779. [on pp. 91, 101, and 102.]

[22] Cliff Click, Gil Tene, and Michael Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference*

on Virtual Execution Environments, pages 46–56, Chicago, Illinois, June 2005. doi: 10.1145/1064979.1064988. [on p. 129.]

[23] Shimon Cohen. Multi-version structures in Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 265–274, November 1984. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/CSD-84-178.pdf. [on p. 100.]

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2nd edition, 2001. [on p. 27.]

[25] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with adabu. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-400-6. doi: 10.1145/1138912.1138918. [on p. 63.]

[26] Dave Dice, Ori Shalev, and Nir Shavir. Transactional locking II. In *International Symposium on Distributed Computing (DISC 2006*, Stockholm, Sweden, September 2006. Swedish Institute of Computer Science. URL http://research.sun.com/scalable/pubs/DISC2006.pdf. [on p. 17.]

[27] *VAX MACRO and Instruction Set Reference Manual*. Digital Equipment Corporation, Maynard, Massachusetts, November 1996. [on p. 145.]

[28] Shahrooz Feizabadi, William Beebee, Jr, Binoy Ravindran, Peng Li, and Martin Rinard. Utility accrual scheduling with Real-Time Java. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, volume 2889/2003 of *LNCS*, pages 550–563, 2003. ISBN 3-540-20494-6. doi: 10.1007/b94345. [on p. 63.]

[29] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 338–349, Montreal, Quebec, Canada, June 1998. doi: 10.1145/781131.781169. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998. [on pp. 29, 149, and 151.]

[30] Catalin A. Francu. Real-time scheduling for Java. Master's thesis, MIT Department of Electrical Engineering and Computer Science, June 2002. [on p. 63.]

[31] Keir Fraser and Tim Harris. Concurrent programming without locks. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/, July 2004. [on p. 17.]

[32] *MPC7450 RISC Microprocessor Family Reference Manual, Rev. 5.* Freescale Semiconductor, Chandler, Arizona, January 2005. MPC7450UM. [on p. 42.]

[33] Jeffery E. F. Friedl. *Mastering Regular Expressions.* O'Reilly & Associates, 2nd edition, July 2002. [on p. 26.]

[34] Ovidiu Gheorghioiu. Statically determining memory consumption of Real-Time Java threads. Master's thesis, MIT Department of Electrical Engineering and Computer Science, June 2002. [on p. 63.]

[35] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatibility analysis for static object preallocation. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-628-5. doi: 10.1145/604131.604154. [on p. 63.]

[36] GNU Classpath. The GNU Classpath project. http://classpath.org/, 2004. [on p. 63.]

[37] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Prentice Hall PTR, 3rd edition, June 2005. [on p. 78.]

[38] Jim Gray. The transaction concept: Virtues and limitations. Technical Report 81.3, Tandem Computers, June 1981. URL http://www.hpl.hp.com/techreports/tandem/TR-81.3.html. [on p. 181.]

[39] Jim Gray. The transaction concept: Virtues and limitations. In *Seventh International Conference of Very Large Data Bases*, pages 144–154, September 1981. Also published as [38]. [on p. 144.]

[40] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. [on p. 144.]

[41] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 123–136. ACM Press, October 1996. ISBN 1-880446-82-0. doi: 10.1145/238721.238767. [on pp. 15 and 28.]

[42] Dirk Grunwald and Soraya Ghiasi. Microarchitectural denial of service: insuring [sic] microarchitectural fairness. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 409–418, Istanbul, Turkey, November 2002. doi: 10.1109/MICRO.2002.1176268. [on p. 132.]

[43] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA 31*, pages 102–113, München, Germany, June 2004. ISBN 0-7695-2143-6. doi: 10.1109/ISCA.2004.1310767. [on p. 145.]

[44] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, Anaheim, California, October 2003. ACM Press. ISBN 1-58113-712-5. doi: 10.1145/949305.949340. [on pp. 17, 28, 39, 82, and 147.]

[45] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-080-9. doi: 10.1145/1065944.1065952. [on pp. 17 and 23.]

[46] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996. ISBN 1-55860-329-8. [on pp. 41, 57, 60, and 61.]

[47] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1): 124–149, January 1991. ISSN 0164-0925. doi: 10.1145/114005.102808. [on p. 142.]

[48] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM TOPLAS*, 15(5):745–770, November 1993. ISSN 0164-0925. doi: 10.1145/161468.161469. [on pp. 95 and 143.]

[49] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Conference on Computer Architecture (ISCA). (Also published as ACM SIGARCH Computer Architecture News, Volume 21, Issue 2, May 1993.)*, pages 289–300, San Diego, California, May 1993. ACM Press. ISBN 0-8186-3810-9. doi: 10.1145/165123.165164. [on pp. 15, 16, 28, 36, 39, 107, 112, 117, 145, and 147.]

[50] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC '03)*, pages 92–101, Boston, Massachusetts, July 2003. ACM Press. ISBN 1-58113-708-7. doi: 10.1145/872035.872048. [on pp. 15, 16, 17, and 146.]

[51] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167495. [on p. 17.]

[52] Maurice P. Herlihy. The transactional manifesto: Software engineering and non-blocking synchronization. Invited talk at PLDI, June 2005. http://research.ihost.com/pldi2005/manifesto.pldi.ppt. [on pp. 21 and 23.]

[53] Maurice P. Herlihy. SXM1.1: Software transactional memory package for C#. http://www.cs.brown.edu/people/mph/, May 2005. [on p. 17.]

[54] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 276–290, Toronto, Ontario, Canada, August 1988. ACM Press. ISBN 0-89791-277-2. doi: 10.1145/62546.62593. [on pp. 15, 141, and 142.]

[55] Maurice P. Herlihy and J. Eliot B. Moss. Transactional support for lock-free data structures. Technical Report 92/07, Digital Cambridge Research Lab, One Kendall Square, Cambridge, MA 02139, December 1992. [on pp. 120 and 145.]

[56] Maurice P. Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, Providence, Rhode Island, May 2003. doi: 10.1109/ICDCS.2003.1203503. [on pp. 15 and 142.]

[57] Gerard J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003. ISBN 0-321-22862-6. [on p. 43.]

[58] *PowerPC Virtual Environment Architecture, Book II, Version 2.02*. IBM, Austin, Texas, January 2005. [on p. 42.]

[59] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, LLNL, Livermore, California, November 1987. [on p. 145.]

[60] Mike Jones. What really happened on Mars? http://research.microsoft.com/~mbj/Mars_Pathfinder/, December 1997. [on p. 15.]

[61] Eric Jul and Bjarne Steensgaard. Implementation of distributed objects in Emerald. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 130–132, Palo Alto, CA, October 1991. IEEE. doi: 10.1109/IWOOOS.1991.183037. [on pp. 29 and 150.]

[62] Felix S. Klock, II. Architecture independent register allocation. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, 2001. [on p. 63.]

[63] Tom Knight. An architecture for mostly functional languages. In *LFP*, pages 105–112. ACM Press, 1986. ISBN 0-89791-200-4. doi: 10.1145/319838.319854. [on pp. 15, 112, 117, 120, and 145.]

[64] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC '94)*, pages 130–140, Los Angeles, CA, August 1994. ACM Press. ISBN 0-89791-654-9. doi: 10.1145/197917.197975. [on p. 144.]

[65] Leslie Lamport. Concurrent reading and writing. *CACM*, 20(11):806–811, November 1977. ISSN 0001-0782. doi: 10.1145/359863.359878. [on pp. 15 and 141.]

[66] Ruby B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, January 1989. doi: 10.1109/2.19825. [on p. 119.]

[67] Jeremy Manson and William Pugh. Semantics of multithreaded Java. Technical Report UCMP-CS-4215, Department of Computer Science, University of Maryland, College Park, January 2002. [on pp. 34 and 147.]

[68] J. F. Martinez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)* ASP [1], pages 18–29. ISBN 1-58113-574-2. doi: 10.1145/605397.605400. [on p. 145.]

[69] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, New York, NY 10027, June 1991. [on pp. 15, 28, 137, and 142.]

[70] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *CACM*, 19(7):395–404, July 1976. doi: 10.1145/360248.360253. [on p. 117.]

[71] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of*

Distributed Computing (PODC '96), pages 267–276, Philadelphia, PA, 23–26 May 1996. ISBN 0-89791-710-3. doi: 10.1145/248052.248106. [on p. 23.]

[72] Sun Microsystems. Java Native Interface. http://java.sun.com/j2se/1.5.0/docs/guide/jni/, 2004. [on pp. 69, 71, and 72.]

[73] Chris Okasaki. Purely functional random-access lists. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 86–95. ACM Press, June 1995. ISBN 0-89791-719-7. doi: 10.1145/224164.224187. [on p. 100.]

[74] Melissa E. O'Neill and F. Warren Burton. A new method for functional arrays. *J. Func. Prog.*, 7(5):487–514, September 1997. doi: 10.1017/S0956796897002852. [on pp. 91, 100, and 102.]

[75] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, Austin, Texas, December 2001. doi: 10.1109/MICRO.2001.991127. [on pp. 124 and 145.]

[76] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In Rajwar ASP [1], pages 5–17. ISBN 1-58113-574-2. doi: 10.1145/605397.605399. [on pp. 15 and 145.]

[77] Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-855-5. doi: 10.1145/1029894.1029917. [on p. 63.]

[78] Martin C. Rinard. Implicitly synchronized abstract data types: Data structures for modular parallel programming. *Journal of Programming Languages*, 6:1–35, 1998. URL http://www.cag.lcs.mit.edu/~rinard/paper/jpl98.pdf. [on p. 22.]

[79] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 2002 International Conference*

on Dependable Systems and Networks (DSN), pages 439–448. IEEE Computer Society, June 2002. ISBN 0-7695-1597-5. doi: 10.1109/DSN.2002.1028929. [on pp. 17 and 146.]

[80] Daniel J. Scales and Kourosh Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 157–169. ACM Press, October 1997. ISBN 0-89791-916-5. doi: 10.1145/268998.266673. [on p. 42.]

[81] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *CACM*, 10(8): 501–506, August 1967. ISSN 0001-0782. doi: 10.1145/363534.363554. [on p. 24.]

[82] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 204–213, Ottawa, Ontario, Canada, August 1995. ACM Press. ISBN 0-89791-710-3. doi: 10.1145/224964.224987. [on pp. 15, 17, 28, and 146.]

[83] D.P. Siewiorek, Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, 1982. [on p. 145.]

[84] Janice M. Stone, Harold S. Stone, Philip Heidelberg, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993. doi: 10.1109/88.260295. [on pp. 15 and 145.]

[85] Alexandru Sălcianu. Pointer analysis and its applications for Java programs. Master's thesis, Massachusetts Institute of Technology, September 2001. [on p. 63.]

[86] Alexandru Sălcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 12–23, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-346-4. doi: 10.1145/379539.379553. [on p. 63.]

[87] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, number 3385 in LNCS, pages 199–215, 2005. doi: 10.1007/b105073. [on p. 63.]

[88] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992. [on p. 14.]

[89] Frédéric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-414-2. doi: 10.1145/378795.378804. [on p. 63.]

[90] John Whaley. Partial method compilation using dynamic profile information. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–179, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-335-9. doi: 10.1145/504282.504295. [on p. 63.]

[91] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA '99*, pages 187–206, Denver, November 1999. doi: 10.1145/320384.320400. [on pp. 63 and 89.]

[92] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanović. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, Austin, Texas, December 2001. doi: 10.1109/MICRO.2001.991111. [on pp. 43, 63, and 77.]

[93] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 304–316, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-574-2. doi: http://doi.acm.org/10.1145/605397.605429. [on p. 63.]

[94] Sharon Zakhour, Scott Hommel, Jacob Royal, Isaac Rabinovitch, Tom Risser, and Mark Hoeber. *The Java Tutorial: A Short Course on the Basics*. Prentice Hall PTR, 4th edition, September 2006. [on p. 81.]

[95] Karen Zee and Martin Rinard. Write barrier removal by static analysis. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 191–210, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-471-1. doi: 10.1145/582419.582439. [on p. 63.]

[96] Lixin Zhang. UVSIM reference manual. Technical Report UUCS-03-011, University of Utah, March 2003. [on p. 120.]

[97] Yoav Zibin, Alex Potanin, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. Technical Report MIT-CSAIL-TR-2007-018, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, March 16, 2007. URL http://hdl.handle.net/1721.1/36850. [on p. 89.]

[98] Craig Zilles and David H. Flint. Challenges to providing performance isolation in transactional memories. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, Madison, Wisconsin, June 2005. URL http://www.ece.wisc.edu/~wddd/2005/papers/WDDD05_zilles.pdf. [on pp. 131 and 132.]