Chapter 3

Log-Based Transactional Memory (LogTM)

LogTM is a strategy for implementing transactional memory a transactional memory system that combines software-based version management (with limited hardware support) and conservative hardware conflict detection to support arbitrary-sized transactions with limited hardware. LogTM adapts a well-known database algorithm for implementing transactions, Strict 2-Phase Locking and Write Ahead Logging. LogTM performs version management using write ahead logging to store old values before new values are written in place. LogTM detects conflicts eagerly, in a manner equivalent to Strict 2-Phase Locking. To balance implementation cost and performance, LogTM divides the work of providing atomicity and isolation in transactions between hardware and software. For speed, hardware detects conflicts; to save complexity, transaction updates are rolled back by software. The following sections describe the requirements for LogTM's version management, conflict detection and conflict resolution. The rest of the chapter describes the LogTM API and discusses the tradeoffs involved in implementing the various components of transactional memory in hardware or software.

3.1 Eager Version Management

A defining feature of LogTM is its use of eager version management, wherein "new" values are stored in place while old values are saved in an alternate location. Specifically, in LogTM, the old values of registers are saved in a register checkpoint and old values of memory are stored in the *transaction log*, a thread-private section of the user program's virtual address space. If the transaction aborts, a software abort handler restores old values from the log and register checkpoint.

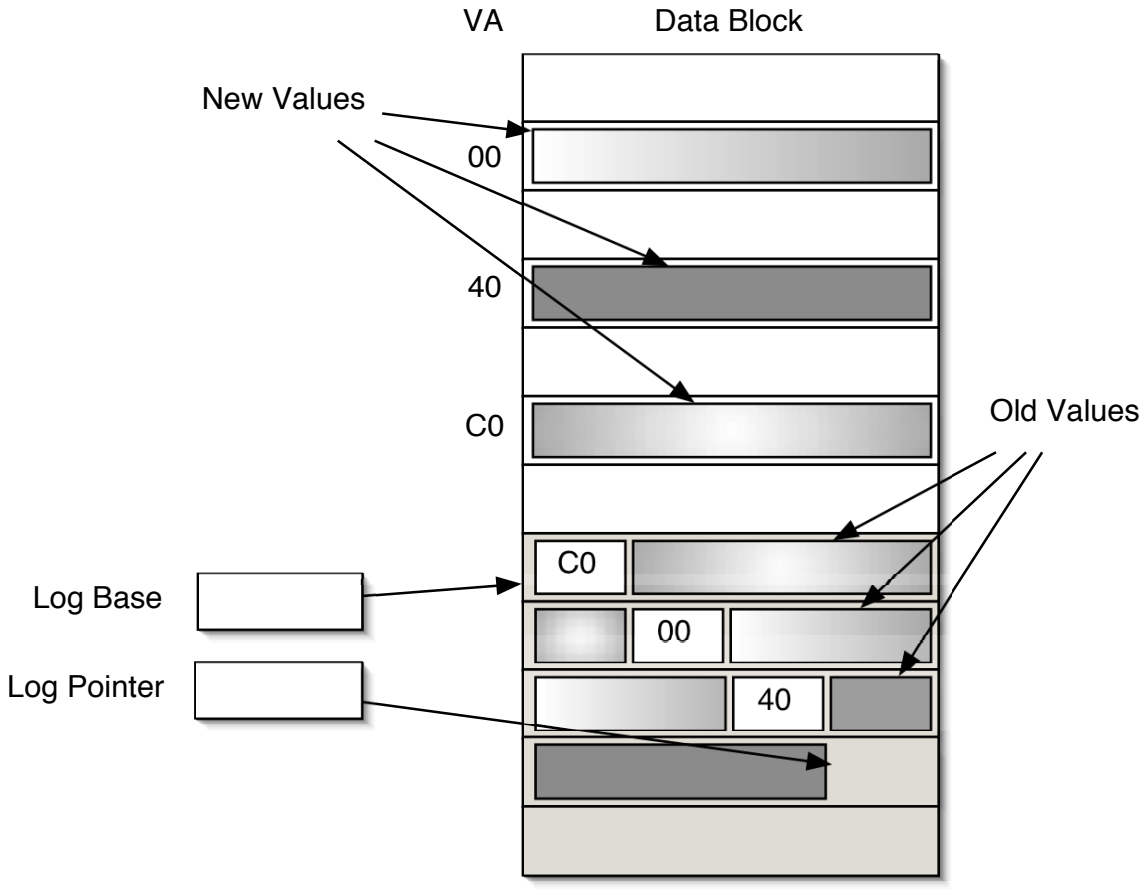

FIGURE 3-1. The Transaction Log.

3.1.1 The Transaction Log

In LogTM, each thread allocates a region of virtual memory, the transaction log, to store a record of the updates made by its most recent transaction. The base and bounds of the log are defined by two processor registers: `Log Base` and `Log Pointer`. The thread sets log base when it allocates space for the log—at thread creation, or when the LogTM system is initialized. The LogTM system updates Log Pointer each time it adds an entry to the log. Figure 3-1 illustrates the layout of the log in virtual memory. The main array is a representation of virtual memory. Virtual addresses are shown on the left. The shaded area (bot-

tom of Figure 3-1) represents the transaction log for the current thread. The unshaded area represents shared memory locations, e.g., part of the heap.

The LogTM system maintains the log at a fixed granularity, logically dividing memory into fixed sized blocks. Figure 3-1 shows an active transaction log after three stores using an 8-byte log granularity. The three pairs of shaded blocks in Figure 3-1 represent the old and new values of three memory blocks. New values are stored in place and old values are stored in the log. The log contains an undo record,which includes the virtual address of the modified block and its old value, for each store executed. Because the purpose of the log is to restore pre-transaction values, a LogTM system need only log the first update to any given memory location.

The log is defined in virtual memory for which physical memory is allocated on demand. If adding a log entry exhausts the physical memory in the log, the current thread takes a page fault and uses the exisiting virtual memory support to allocate an additional page for the log. Because the transaction log is stored in virtual memory, it may grow arbitrarily large. Log pages may be swapped to disk without impacting LogTM's version management.

In database terminology, the transaction log is an undo-only log—i.e., it does not contain sufficient information to re-execute a transaction nor to recover the state of memory in the event of a crash [45]. This is in contrast to the logging used in most database systems. In a typical database, the log is used for recovery as well as transaction rollback. Importantly, because the transaction log is not used for recovery, there is no need to store the log on disk or other stable storeage. Furthermore, since the log is not needed after the commit of a transaction, unlike the log in most database systems, the transaction log in LogTM is thread private and may be discarded after a transaction completes.

3.1.2 Transaction Commit

To commit a transaction, the executing thread simply clears its transaction log by setting the log pointer equal to the log base and discards its register checkpoint. No copying is needed because new values are already in place.

3.1.3 Transaction Abort

To maintain atomicity in the event of a transaction abort, LogTM must undo any updates from the aborting transaction by restoring the old values maintained in the transaction log and register checkpoint to their original locations in memory and processor registers. In order to save hardware complexity, LogTM performs this restoration in software. In LogTM, an abort restores old values by: (1) jumping to the start of the abort handler, (2) executing the handler (a load and store for every word in the log and an additional load for each address—one per log entry—in the log), and (3) restoring the processor registers to their pre-transaction states.

3.2 Eager Conflict Detection

LogTM employs eager conflict detection: the system must detect and resolve any conflict triggered by a memory request before that request completes. In LogTM, this detection is the responsibility of hardware. Implementations of LogTM are expected to leverage the coherence mechanism to implement conflict detection efficiently. LogTM implementations must report all true conflicts between concurrent transactions, but to reduce hardware complexity and cost, they are allowed to report false conflicts.

3.2.1 Requirements

LogTM's eager conflict detection is based on strict two-phase locking. In place of the shared and exclusive locks used in database systems, however, LogTM requires read and write *isolation*. If a block is read iso-

lated, it cannot be written by any thread without generating a conflict. If a block is write isolated, it cannot be read or written by any thread without generating a conflict. Unlike locking, isolation does not prescribe any particular conflict resolution mechanism (e.g., blocking).

Requirement 1: Transactions Must be Well Formed. In order for a thread to read a memory location in a transaction, that thread must obtain read isolation on that location. In order to write a location, a thread must obtain write isolation on that location. If an attempt to acquire read or write isolation fails, or results in a transaction conflict, the system must signal a conflict before the offending memory instruction is retired.

Requirement 2: Isolation Must be Strict Two Phase. This requires that the locking, or isolation in LogTM must be strict two phase. Any memory location that becomes read or write isolated by being read or written in a transaction must remain isolated until the commit or abort of that transaction.

Requirement 3: Isolation Must be Released at Transaction End. Conflicts may prevent one or more tranasctions from making forward progress. In order to ensure forward progress in the system, a thread must release its isolation when it aborts or commits its transaction.

3.2.2 Example Implementation

For small transactions—the expected common case—the entire read and write set of the transaction remains in the private cache of the executing processor. In this case, a standard invalidation-based cache coherence protocol is well suited to detecting transaction conflicts. As discussed in Chapter 2, cache-coherent multiprocessors that use invalidation-based coherence protocols typically enforce the invariant that every block of memory resides in one of three logical states: (I) the block is invalid in all caches, (S) one or more caches hold a valid read-only copy of the block and (M) one cache only has a writable copy of the block. Such systems already require that a processor obtain shared (read only) access to a memory

location before it may be read and exclusive access (read/write permission) before it may be written. As a result, if isolation is provided by the coherence mechanism, these protocols enforce the well formed requirement by default. Additionally, these rules ensure that any memory access that hits in the local (private) cache will not trigger a transaction conflict. Conflict detection in such a system works as follows: each processor tracks the read and write sets of its active transaction with two additional bits per line in its private cache. The read (R) bit indicates that the block has been read during the current transaction. The write (W) bit indicates that the block has been written during the current transaction and potentially contains data values from an uncommitted transaction. When a processor P executes a load or store, it first checks its local cache. If the corresponding block is not present, P issues a request for the block to the memory system. That request is sent to one or more other processors (e.g., via a broadcast or forwarded by a directory). Those processors check their local state (in the cache or memory controller) to detect conflicts with any transactions running there. The presence of a conflict is returned along with the coherence response. That response signals the conflict (if any) to P, which resolves the conflict. The states of the contended block in the responding processors' caches remain the same, as do the R and W bits, until the transactions running on those processors end by committing or aborting. This fulfills the strict two-phase requirement. Finally, at the end of a transaction, the processor flash-clears the R and W bits in its cache, fulfilling the requirement to release all read and write isolation after the transaction commits or aborts.

3.3 Conflict Resolution

When two transactions conflict, any transactional memory implementation must stall (risking deadlock) or abort (risking live-lock) at least one transaction. Recall that when a LogTM processor P makes a coherence request, it may get forwarded to processor Q to *detect* a conflict. Q then responds to P, including the presence or absence of a conflict in the response. If there is a conflict, processor P *resolves* it upon receiving the response. P may resolve the conflict by either stalling or aborting. To reduce the frequency of aborts (which

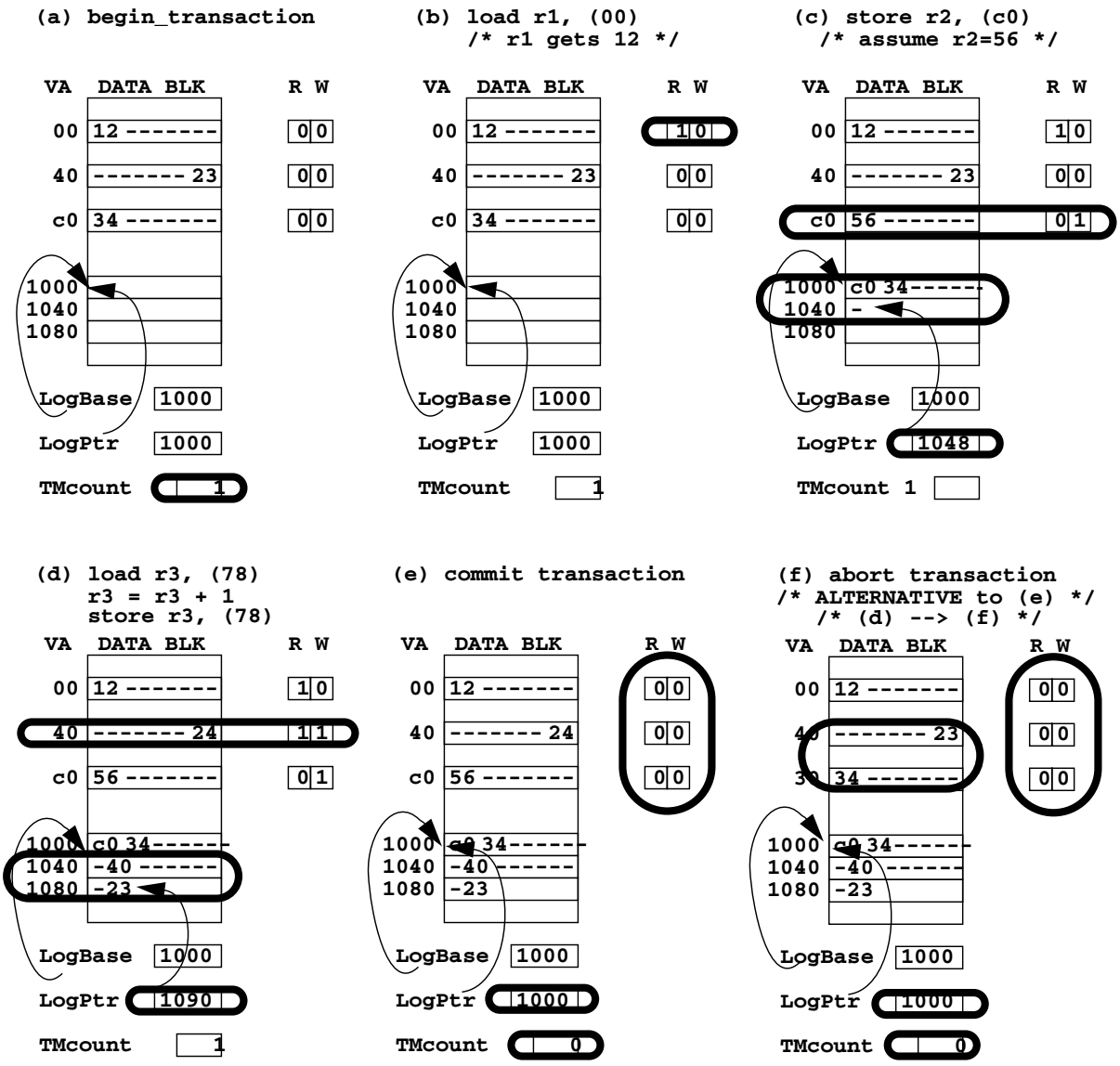

FIGURE 3-2. Execution of a Transaction with Two Alternative Endings

waste work and power), it might be preferable for P to wait for a short time and then re-issue its request to Q in the hope that Q has completed its conflicting transaction. P cannot wait indefinitely for Q, however, without risking deadlock (e.g., if Q is waiting on P).

3.4 Example

Although presented separately in this chapter, conflict detection and version management interact in important ways. To better understand the way LogTM's eager version management and eager conflict detection work together, consider the example depicted in Figure 3-2. Figure 3-2 illustrates the logical execution of a simple transaction. Part (a) displays a logical view of a thread that has just begun a transaction by incrementing its `TMcount`. Assume that the thread's log begins at virtual address (`VA`) 1000 (all numbers in hexadecimal), but is empty (`Log Pointer=Log Base`). In this example, values of data blocks are given as a two-digit word and seven dashes (for the other eight-byte words of a 64-byte block). The conflict detection state for each block is shown to the right. The R and W flags indicate whether a block is read or write isolated by the conflict detection mechanism. Circles indicate changes from the previous snapshot. Part (b) shows a load from virtual address 00 acquiring read isolation on that block. Part (c) depicts a store to virtual address c0 acquiring write isolation on that block and logging the block's virtual address and old data (34 --------). Part (d) shows a read-modify write of address 78 that acquires read and write isolation for the encompassing block and writes the log with the block's virtual address (40) and old data (------- 23). Part (e) shows a transaction commit that resets `TMcount`, `LogPtr`, and releases all read and write isolation. Part (f) shows an alternative where, after part (d), something triggers an abort, which must restore values from the log before resetting the `TMcount`, `Log Pointer`, and releasing read and write isolation.

3.5 LogTM API

Table 3-1 presents LogTM's interface in three levels. The *user interface* (top) allows user threads to *begin*, *commit* and *abort* transactions. Compilers could translate higher level constructs such as an atomic block to LogTM's `begin_transaction` and `commit_transaction` calls like the Java compiler gener-

ates monitor enter and exit calls from statically scoped synchronized blocks [JAVA???]. The system/ library interface (middle) lets thread packages initialize per-thread logs and register an abort handler. Upon an abort, LogTM lets the abort handler "undo" the log via a sequence of calls using the low-level interface (bottom). In the common case, the handler can restart the transaction with user-visible register and memory state restored to their pre-transactions values. Rather than just restart, an abort handler can also complete an abort and run arbitrary user code to manage aborts.

3.6 Discussion

LogTM is designed to provide robust performance when transactions exceed the size or associativity limits of the hardware. To support such transactions, LogTM must therefore provide both conflict detection and version management for data outside processors' private caches and other dedicated hardware structures. Conflict detection and version management pose separate challenges to operating outside the cache. The distinct characteristics of these challenges lead LogTM to handle each differently, extending hardware to provide conflict detection for out-of-cache transactions and deferring version management to software.

Version management in unbounded transactions is difficult because the space required to store the separate versions is unbounded. Version management is the maintenance of the program's data values and must therefore be performed precisely—i.e., values cannot be altered, lost or associated with the wrong version. Because this version information must be maintained precisely, a transactional memory system must provide storage for both old and new versions of each object modified in a transaction, requiring space equal to the write set (in additional to the program's space requirements). Memory provides ample space to maintain both old and new versions of transactional data, but using memory in such a way requires associating separate addresses for each version and implementing a policy to ensure that every memory access reaches the proper version. Fortunately, however, versions are switched rarely—on abort

if eager version management is used, or on commit otherwise. LogTM employs eager version management because aborts should be rarer than commits. LogTM takes advantage of that rarity by passing the responsibility for switching versions to software, in the form of a software abort handler.

Conflict detection for unbounded transactions is challenging because, unlike version management, it must be logically performed on every memory access by every processor. Although caches can filter out conflict checks for accesses that hit in the cache, conflict detection is still performed frequently. Conflict detection is especially challenging to implement in software because it requires remote operations, i.e., the read and write set of an overflowing transaction must be checked on every cache miss from every other processor. Fortunately, unlike version management, conflict detection need not be performed precisely, but merely *conservatively.* A conflict detection mechanism must report all true conflicts between transactions, but reporting a conflict when one does not exist (false conflict) will affect performance, but not correctness. LogTM leverages this property by tracking read and write sets conservatively in hardware. By allowing false conflicts, LogTM can detect all true conflicts between transactions of any size while using finite structures to track read and write sets.

The use of eager version management in LogTM has two primary advantages: (1) commits are fast even for large transactions and (2) loads never need to check local write buffers. Although aborting transaactions is more costly with eager version management, aborts often have little effect on execution time in general because most transactions commit—more than 98% for many workloads (Chapter 5).

Eager conflict detection has two primary advantages: (1) it detects conflicts sooner and (2) it eliminates the need for cascading rollbacks on systems that use eager version management. Detecting conflicts early can reduce wasted work in two important ways. First, detecting a conflict early allows a transactional memory to abort a transaction early, giving it the opportunity to switch execution to a different thread or transaction. Because transactions are atomic, any transaction that does not commit does not contribute to the progress of the program. A transactional memory system that can identify unsuccessful transactions early can abort

those transactions sooner and potentially run other code that will make progress. Second, detecting conflicts early, allows a transactional memory system to resolve many conflicts by stalling one or more transactions. Stalling instead of aborting eliminates wasting work all together.

Strict two-phase locking and write ahead logging has proven to be an effective strategy for implementing transactions in database systesm. Eliminating the requirement for durabilty and leveraging hardware support allows LogTM to adapt this successful algorithm to efficiently implement lightweight memory transactions with little overhead. Eagerly making updates in place allows LogTM to support large transactions without copying on commit and makes processing commits, which should be more common, easier than processing aborts. Detecting conflicts completely in hardware allows for efficient execution even in presence of large transactions. This combination allows LogTM systems to balance performance and implementation cost in transactional memory.