Chapter 4

Implementing LogTM

In the previous chapter, I outline the LogTM system and API without specifying a particular implmentation. Here, I discuss the challenges of implementing LogTM especially LogTM's eager version management and eager conflict detection. Section 4.1 discusses several tradeoffs in implementing LogTM's version managment and presents three solutions that vary in complexity and performance. Section 4.2 presents two concrete implementations of LogTM's eager conflict detection based on directory and broadcast-based coherence. Section 4.3 presents a policy for resolving conflicts in LogTM.

4.1 Implementing LogTM's Eager Version Management

Recall from the previous chapter that the LogTM interface (Section 3.5) specifies that the transaction log must contain an undo record for each block modified by the transaction at the time of an abort. LogTM implementations must also restore the value of processor registers to their pre-transaction values before a transaction can safely be re-executed. Implementations, however, are free to perform these tasks in many ways.

4.1.1 Implementation Trade-Offs

This sub-section describes several design dimensions LogTM system designers must address when implementing the transaction log. The following sub-sections discuss the tradeoffs involved in these choices in the contex of three strategies for implementating logging with varying levels of hardware support: soft-

ware-only logging (no support), hardware-only logging (full support), and hardware/software hybrid (some support).

Hardware vs. Software Register Checkpointing. LogTM designers seeking the highest performing system should consider including a hardware register checkpointing mechansim. Hardware register checkpointing has been proposed both for transactional memory [4, 22, 33, 46] and as a mechansim for implementing speculation in out-of-order processors [3]. Alternatively, to reduce hardware costs, register state could be saved and restored in software. The log provides convenient storage for register values. For small transactions that do not make any procedure calls, a compiler could be modified to generate code to save only the registers that will be overwritten by the transaction.

Implicit vs. Explicit Logging. Perhaps the most important design choice is whether logging will be performed *explicitly*—as specifically directed by software, e.g., a hardware instruction—or *implicitly*—as a side effect of another instruction, e.g., a transactional store. This decision is particularly important because it affects the ISA, which changes much less frequently than the microarchitecture. Explicit logging could be performed using existing instructions, e.g., by copying values from updated memory locations to the log in software, then incrementing the log pointer register directly. Or, explicit logging could be enhanced by special instructions that use dedicated hardware to more efficiently create log entries. Such instructions could also optionally check membership in the log buffer (described below), or W bits (Section 4.2) to eliminate unneccessary duplicate log entries.

Implicit logging will likely be more difficult to implement than explicit logging. A store with implicit logging must at least: (1) read the old values from the target memory block and write them to the cache, or a dedictated buffer, (2) record the target memory block's virtual address, (3) write new values into the cache or store buffer. Such stores are even more complicated without the aid of a log buffer (described below), in which case the processor must additionally translate the address of the log pointer, copy old values and vir-

tual address to the log pointer and increment the log pointer. This difference is especially true for RISC architectures, in which instructions tend to be short, simple operations that can be performed directly by the microarchitecture. Modern CISC architectures that convert CISC instructions into series of micro-ops, however, provide an interesting alternative. Such machines could implement logging that is implicit as seen by application software, but implemented by explicit micro ops. Stores in a transaction might translate to a different series of micro-ops than non-transactional stores.

Implicit logging, however, allows for a cleaner interface for transactional memory programs. No additional instructions are required in the ISA. Furthermore, functions that are called both inside and outside of transactions do not need to written or compiled differently for the two cases. Whereas, if explicit logging is used, any function that is invoked from within a transaction must contain these logging instructions. Runtime support such as *just-in-time* (JIT) compilation, however, could mitigate this effect [1] by allowing for functions to be recompiled with logging on demand.

Buffered vs. Direct Logging. LogTM requires that the transaction log be appropriately filled on abort. Prior to an abort, however, implementations are free to store undo information in any form. An implementation of LogTM could use a small *log buffer* to temporarily store a small number of log entries. The log buffer could be filled by hardware via either explict or implicit logging. For explicit logging, hardware might provide a special log instruction that copies a memory block to the log buffer. Because the size of the buffer is limited, for large transactions, log entries in the log buffer must be spilled to the log itself (in virtual memory). These spills could be performed implicitly in the background, or explicitly by making the contents of the log buffer visible to software. Block-store operations (e.g., SPARC's `stx` instruction [56]) could speed up both log buffer spills and abort processing.

Use of a log buffer has three primary advantages. First, it can reduce bandwidth demand on the memory system. Because transactions often have small write sets, a small log buffer can eliminate the need for copying data to the log for many transactions. Second, storing to a log buffer could be used to reduce the

pressure to translate log addresses. Because the log buffer is not part of virtual memory, address translation can be defered until the buffer is spilled to memory. Finally, using a log buffer offers an opportunity to provide hardware support for logging without defining the log format in hardware. Instead, the log format can be defined by the software used to spill the log buffer to the transaction log in memory.

Logging Granularity. Logically, the log can be stored at any granularity provided that all transactional updates are included in the log, and that isolation is maintained on all memory locations that are logged. That freedom allows system designers to select the granularity that will perform best for their system and workloads. Using larger log records reduces the number of log records generated by each transaction. Using larger records, can potentially reduce the overall size of the log by reducing the number of addresses in the log, provided there is enough spatial locality in transactional stores to fill the larger records. Smaller records, however, can reduce the size of the log if more memory locations are logged than actually updated. Reducing the size of the log may reduce the bandwidth demands on the cache.

Logging Location. LogTM systems are free to store log values—either buffered or in memory—at any level of the memory hierarchy. Caches, in particular those with ECC protection, can be extended to created log entries. Store instructions typically modify a small number of bytes of memory, e.g., 4-8. Cache lines are typically much larger (e.g., 64-128 B). This discrepancy means that store operations typically merge new values—recorded by the given store instruction—with values already in memory. To perform this merge, the cache must hold a valid copy of the memory block. That copy contains exactly the values that must be recorded in the undo record that corresponds to the store. Furthermore, if the cache is protected by ECC, the cache must read the values in the entire ECC word to compute a new ECC code after each such merge. Since the entire memory word is already being read, the calculation of the new ECC code provides a convenient opportunity to copy old values to the log, particularly if the log is maintained at the granularity of an ECC word.

Logging at higher levels of the memory system (e.g., second or third-level caches), although efficient in time, requires additional hardware complexity. First, in order to create the undo record, the processor must pass both the virtual and physical addresses of the target memory location (caches that are physically tagged and physically indexed do not require the virtual address). If the system uses direct logging, the address of the top of the log (the log pointer) must also be passed to the cache, which must be translated to a physical address if, as is typical, the cache is physically tagged. Additionally, because larger caches are typically multi-banked, the log target and store target may be stored in different banks. Therefore, values copied from the store target must be moved from their original bank to the bank that holds the corresponding undo record. Furthermore, although log stores are nearly always cache hits (Section 5.5.2), the cache must be able to handle misses to the log (i.e., the memory block that will hold the undo record is not present). In that case, the cache must have sufficient space to buffer log values, or must be able to stall the processor until the miss has been serviced. However, buffered logging can reduce the complexity of in-cache logging by deffering translation of log addresses until the log buffer is full, and by providing the buffering needed to hold old values while the cache blocks that form the log are fetched from memory.

4.1.2 Compiler-Supported Software Logging

Designers seeking to implement LogTM with minimal hardware support should consider software-implemented logging. The *log base* and *log pointer* registers can be read and modified by user instructions. Because the log is stored in user-addressable virtual memory, ordinary loads and stores can access log values as well. In software logging, user software generates a log entry by first writing the virtual address of the block to the log—the current value of log pointer—then copying each word in the block to subsequent locations in the log. Once the log holds the block address and old values, the software increments the log pointer by the size of a virtual address plus the size of the block. To reduce the burden on the programmer, a modified compiler generates the logging instructions automatically. Because both the logging actions and

the transaction rollback mechanism are under software control, the programmer can choose the logging granularity that is most appropriate for a given application. The expense of logging in software will likely favor logging at a fine granularity—4 or 8 bytes per entry—to reduce unecessary logging.

4.1.3 In-Cache Hardware Logging

To implement LogTM with the best possible performance, designers should consider adding hardware support for in-cache implicit logging. In such a system, the processor translates the target address and log pointer value for each transactional store into physical addresses and sends the physical addresses to the cache along with the new value. The processor uses a single-entry micro-TLB to translate the log pointer. If the log pointer address misses in the micro-TLB, the processor executes a page fault to allocate additional physical memory to the log.

If the L1 cache is write-back, write-allocate and protected with ECC bits, as in the AMD Opteron [27], the L1 cache controller should create log entries at the same granularity at which ECC codes are computed. In that case, the cache performs a lookup of both the target address and the log pointer for each store (lookups are often performed in parallel with address translation using virtual addresses and virtually indexed caches). Figure 4-1 illustrates the data flow, if the translated addresses (log pointer and store target) hit in the cache. The cache reads the value of the target block to a register. The cache then copies the value of that register—the old value of the modified memory block—to the block at the log pointer. Next, the new val-

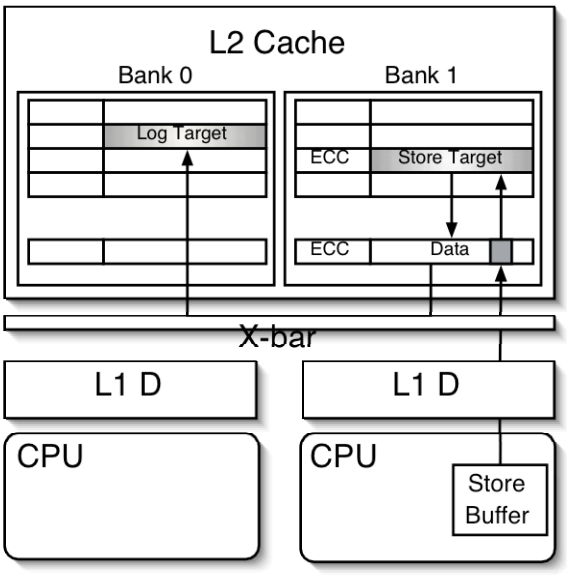

FIGURE 4-1. In-Cache Hardware Logging.

ues from the processor (shown as the shaded section in the Data field in Figure 4-1) are incorporated into the values in the register. One the full block is assembled, the new ECC code is calculated. When the new values and new ECC code are ready, the cache copies them to the target memory block.

If, on the other hand, the L1 cache is write-through, write-no-allocate and not ECC protected, as in the Sun Ultrasparc T1 (Niagara) [29], in-cache logging should be performed at the L2 cache, or the closest write-back cache. Systems like Niagara, already send store values and addresses directly to the L2. In Niagara, even if the target line is present in the L1 cache, the L1 copy of the line is not updated until after the data reaches the L2. Performing logging in the L2 cache will reduce the bandwidth added by logging by keeping old values in the L2 cache. Each thread's log is private so the transaction log may be safely stored in a shared cache. Also, because log entries are only read on transaction aborts, which are rare (Section 5.5), storing them in lower levels of cache (and not in the L1) may improve the effectiveness of the L1 cache for many workloads.

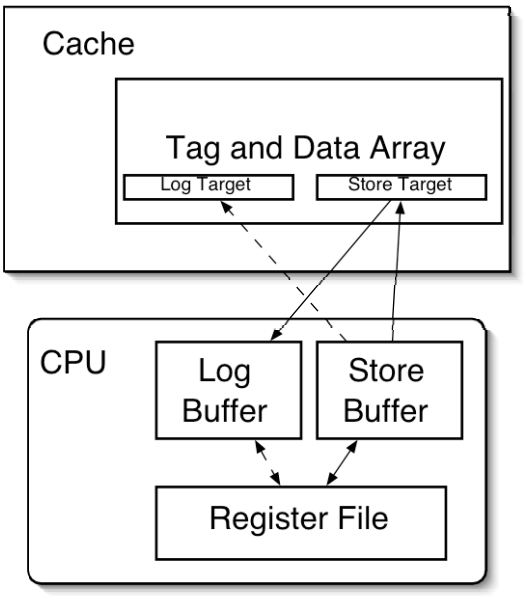

FIGURE 4-2. Hardware/Software Hybrid Buffered Logging.

4.1.4 Hardware/Software Hybrid Logging

An alternative to the complexity of all-harware logging and the overheads of software-only logging is to support software-based logging with a small log buffer, which is filled quickly by hardware and spilled periodically to memory by a software handler. Because the log buffer may be filled without translating the virutal address in the log pointer to the physical address of the log, logging using a buffer does not require additional address translation. Figure 4-2 depicts the log buffer and information flow in buffered logging. Solid arrows represent values sent during transaction execution (buffer fill) and dashed arrows represent values sent during buffer spill.

Characterizations of critical sections in multithreaded programs [11] and transactional memory workloads [48] as well as results presented in Chapter 5, show that a small log buffer could process all logging for many transactions in the benchmarks studied in this dissertation. Log buffers could provide the speed of all-hardware logging for many transactions without requiring the added complexity of (logically) perform-

ing an additional address translation for each transactional store instruction. Using a log buffer, address translation is only required during buffer spill operations. Spilling the log buffer could be implemented as a software trap, much like TLB miss traps in SPARC processors. Because the transaction log itself is updated by software only when the log buffer is spilled, the use of the log buffer does not set the format of the log in hardware.

4.2 Implementing Eager Conflict Detection

In order to isolate transactional stores—which are made in place—LogTM must detect conflicts eagerly, on each load and store. Fortunately, existing coherence mechanisms are well-suited to detect transaction conflicts provided that transactional data reside in the cache. LogTM's interface, however, requires that hardware detect conflicts for all transactions. The primary challenge in implementing LogTM's eager conflict detection is therefore to detect conflicts on data accessed in a transaction, but no longer in the cache. In this section, I first describe conflict detection for cached data then present two strategies for implementing LogTM's eager conflict detection that leverage the coherence mechanism for fast detection, but do not require the caching of transactional data.

In the uncommon event that a cache block containing a memory location in the read or write set of an active transaction is evicted from the executing processor's private cache, LogTM continues to track accesses to the block by (1) ensuring that the executing processor is notified of (and has a chance to nack) all conflicting memory operations and (2) filtering out many false conflicts at the processor. Which of these presents the greater challenge depends on the underlying coherence mechanism. In broadcast systems (e.g., the AMD Hammer [2]), all potentially conflicting memory operations are trivially guaranteed to be sent to the executing processor because all memory operations that miss in a processor's cache are broadcast to all other processors. Because many non-conflicting memory accesses are sent to each processor, however, filtering out false conflicts becomes difficult. In directory systems, on the other hand, novel

coherence extensions are required to ensure that conflicting memory operations are routed to processors that no longer contain the shared block.

As with any block in a LogTM system, transactional conflict detection for out-of-cache blocks follows the pattern of a requesting processor sending a coherence message to every processor whose transaction may conflict with the current memory operation, allowing each processor to detect a potential conflict by checking only local state. For blocks outside the cache, however, the local state cannot be the coherence state of the block. Instead, the local state is a filter that conservatively approximates all of blocks in the read and write set of the current transaction that have been evicted from the processor's cache. The next sections describe two implementations of LogTM, LogTM-Dir and LogTM-Bcast, based on directory and broadcast coherence respectively.

4.2.1 LogTM Directory

LogTM Directory (LogTM-Dir) extends a conventional MESI directory protocol with novel *sticky states* and support for NACK'ed requests to provide transactional conflict detection even when transactional data sets exceed the capacity or associativity of processors' private caches. Extending a conventional directory protocol to support LogTM presents two primary challenges. First, because transaction conflicts result in negative responses (NACKs), the directory cannot assume that all memory requests it forwards will complete successfully. Second, because LogTM must support transactions whose data sets exceed the storage capabilities of processors' private caches, a LogTM directory must continue to forward all potentially conflicting memory requests to a processor running a transaction even for memory blocks that have been evicted from that processor's cache. The rest of this section describes the base protocol, MESI-Dir, and the LogTM extended protocol, LogTM-Dir.

MESI-Dir. MESI-Dir is a cache-coherent non-uniform memory access (ccNUMA) multiprocessor loosely based on the SGI Origin multiprocessor [32]. Like the SGI Origin, MESI-Dir is comprised of sev-

eral nodes, each of which contains a processor (nodes in the SGI Origin had two processors), a block of memory and a directory memory. Also like the SGI Origin, MESI-Dir maintains coherence using a full-bit vector directory and an invalidation-based MESI coherence protocol. Both protocols support the clean exclusive state (E) with silent evictions—i.e., processors may evict lines in the E state without notifying the directory. Unlike in the Origin protocol, however, in MESI-Dir's protocol, the directory does not send speculative data replies for blocks owned by processors (blocks in states E or M). As a result, in MESI-Dir, a processor sends an extra message (the CLEAN message) to the directory in the case that it receives a forwarded get-shared or get-exclusive request for a block not present in its cache (e.g., a block that was formerly present in the E state and replaced silently from the cache). The directory responds to CLEAN messages by sending the data from memory to the original requester. The extra message causes MESI-Dir to be slower in this case, but not sending speculative data responses reduces the use of interconnection bandwidth.

LogTM-Dir. LogTM-Dir first extends MESI-Dir with NACK messages that signal transaction conflicts. Processors respond with a NACK to any coherence message that would otherwise require the violation of LogTM's strict two-phase requirement—i.e. a transaction conflict. When all transactional data are cached, LogTM-dir behaves like the example implementation described in Section 3.2.2. Figure 4-3 illustrates two cases of transaction conflict: Figure 4-3 (a) depicts processor P0 attempting to read a block transactionally modified by processor P1; Figure 4-3 (b) shows P0 attempting to write a block transactionally read by P1. The blocks represent the state of a single memory block in each processor's cache and at the directory/ memory. The cache state includes both the coherence state (MESI) and the transactional R and W bits. In the first case, the directory forwards P0's shared request (GETS) to P1. P1 checks its local cache for the requested block. Finding the block present and the W bit set, P1 responds with a NACK alterting P0 of the

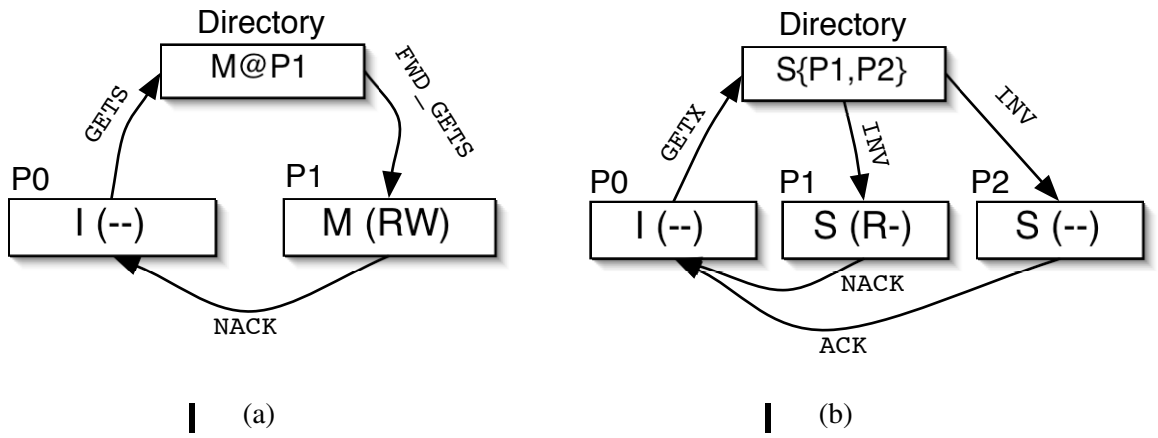

FIGURE 4-3. In-Cache Conflict Detection in LogTM-Dir.

conflict. In the second case, P0 sends a get-exclusive request (GETX) to the directory. The directory responds by sending invalidation messages (INV) to all of the processors on the sharers list (P1 and P2). Both P1 and P2 have shared copies of the block, but only P1 has read it as part of a transaction. P2 responds with an ACK message according to the MESI-Dir protocol, but P1 seeing that the block is present in its cache and that the R bit for the block is set, responds with a NACK signalling the presence of a conflict to P0.

To detect transaction conflicts on data outside the executing processor's private cache, LogTM-Dir adds logical sticky-S and sticky-M states. These states allow the directory to forward all potentially conflicting memory requests to the executing processor. The states are logical in that the behavior of the directory controller is the same for the S and sticky-S state and for the M and sticky-M states. The difference is that in a sticky state, the processor does not posses a valid copy of the data. Each processor maintains a single overflow bit, which is set if it evicts any block from its cache for which the R or W bit is set. When the overflow bit is set, a LogTM processor conservatively assumes that any invalidation messages or forwarded requests from the directory concerning blocks not present in its cache conflict with its current transaction.

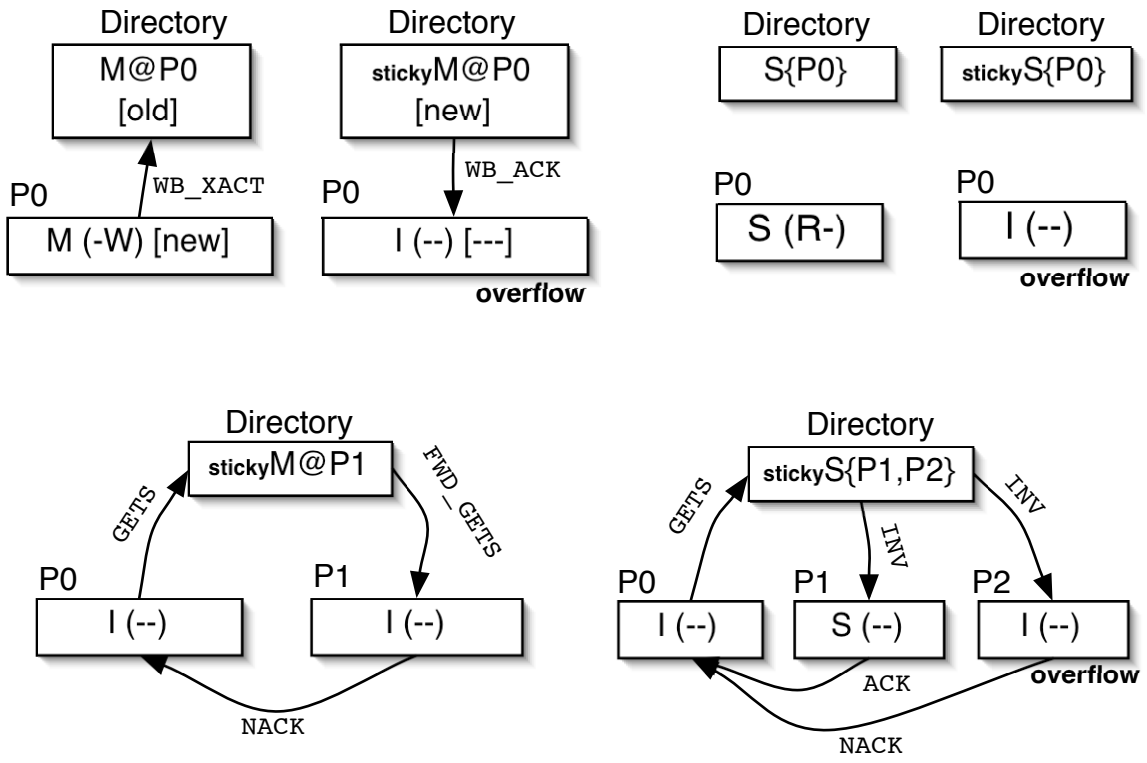

FIGURE 4-4. Conflict Dection on Un-Cached Data in LogTM-Dir Using Sticky States.

A block enters the sticky-S state when a processor executing a transaction evicts a block for which the R bit is set (meaning the block was read as part of the current transaction). As in the base protocol, the eviction is silent and the evicting processor remains on the directory's sharers list. If the R-bit is set a LogTM-Dir processor sets its overflow bit. Because the block was read, but not modified in the transaction (otherwise the processor would have the block in the M state), only a store to that block will cause a conflict. Because the evicting processor is on the directory's sharers list, any store to the evicted block will cause the directory to send an invalidation message to the evicting processor. When it receives the invalidation message, the evicting processor first checks its local cache for the block. Seeing that the block is not present and precise information about the transactional status of that block (i.e., the R/W bits) is not avail-

able, the evicting processor then checks its overflow bit to determine if the invalidation may indicate a possible transaction conflict. If the overflow bit is set, the processor NACK's the request to ensure safety.

A block enters the sticky-M state if a processor evicts a block for which the W bit is set. In that case, the processor sends a special PUTX message along with the data to the directory indicating that the block is part of its current transaction. The directory controller receives the modfied data from the evicting processor (the "new" version) and updates the in-memory copy of the block. But, because the block must still be isolated as part of a transaction, the directory does not change the coherence state of the block. Instead, the block enters the sticky-M state wherein the directory refers all requests for the block to the evicting processor, which no longer possesses a copy of the block. Because the block was modified in the transaction, any request for the block (load or store) will result in a transaction conflict. When the evicting processor receives forwarded GETS or GETX requests from the directory for the evicted block (or any block not present in its cache), it will respond with a NACK.

A sticky-M state is usually cleaned on the first access to the sticky block following the termination (successful or otherwise) of the transaction that caused it to enter the sticky state. When a processor next issues a request for the block (by definition it cannot be valid in any processor's cache), the directory forwards the request to the block's former owner. Because the requested block is not present in its cache, the former owner responds to the forwarded request by checking its overflow bit. Assuming the overflow bit is clear, the former owner can rule out a possible conflict and respond positively to the request. The responding processor cannot supply the data, however, since it is not caching the block. Instead, it sends a CLEAN message to the directory containing the identity of the requesting processor. When the directory receives a CLEAN message, it knows that the data in memory is valid. The directory responds by sending the data from memory to the requesting processor and changing the owner of the block to be the requesting processor. The directory changes the state of the block from sticky-M to M since the requestor will receive an exclusive copy of the data regardless of whether it issued a GETS or GETX request.

A sticky-M state will not be cleared, however, if the processor that is the former owner of the block is currently executing a subsequent overflowed transaction. In that case, the responding processor will not be able to distinguish whether the block in sticky-M was put in that state as a part of its current transaction or a previous one. Therefore, to ensure safety, it must conservatively nack the request. Assuming that overflowed transactions are uncommon, such false conflicts are likely to be rare because they require not one, but at least two transactions to overflow on the same processor.

The rarity of overflowed transactions will prevent false conflicts from degrading performance in most situations, but the sticky states could potentially cause problems when more than one transactional application is running on the same machine. Specifically, if a thread from process p1, which has recently run and committed a large transaction that overflowed the cache and left blocks in sticky states, is preempted and a thread from process p2 is scheduled in its place, then any overflowing transaction executed by the thread from p2 will potentially affect the performance of all threads in p1. Although, both applications will run correctly, the system will not be able to provide *performance isolation* between them [62].

4.2.2 LogTM-Bcast

LogTM-Bcast is based on the AMD Hammer protocol as described in [2, 35]. As in the Hammer protocol, all requests in LogTM-Bcast are sent to memory, which, like a directory, either responds with data, or forwards the request to other processors. Unlike directory-based systems, however, memory does not maintain a list of sharers or a pointer to an owning node. Instead, all forwarded requests are broadcast to all nodes. The "directory" in a Hammer system simply serves as an ordering point for memory requests, but does not reduce coherence bandwidth.

For transactions in which all data remain in the executing processor's private cache, LogTM-Bcast detects conflicts in essentially the same manner as LogTM-Dir. Cache lines are annotated with R and W bits to track the read and write set of the current transaction. Memory requests are sent to the directory, then

broadcast to all nodes. Processors receiving GETS or GETX requests check the R and W bits for the corresponding line (if present in their cache). Each processor responds with an ACK or NACK depending on whether the R and W bits in their cache represent a conflict.

When transactional data overflow the cache, however, the behavior of LogTM-Bcast differs from that of LogTM-Dir. LogTM-Bcast does not need sticky states because the fact that all coherence requests are broadcast to all nodes guarantees that all potentially conflicting memory requests will be sent to any processor executing a transaction regardless of whether or not that processor has a valid copy of the requested block in its cache. Broadcasting, however, also means that many more non-conflicting requests will be sent to each processor. If LogTM-Bcast were to employ the single-bit overflow filtering scheme used in LogTM-Dir, every cache miss on every processor would generate a conflict with any transaction that overflows its processor's cache. Instead, LogTM-Bcast uses a Bloom filter [5] to store a conservative summary of the portion of the read and write sets that have overflowed the local cache.

Bloom filters encode membership in a set allowing *false positives*—reporting membership when the object is not present—but not *false negatives*—reporting absence when the object is present. Similar to a hash-table, an object's location in a Bloom filter is determined by a hash function. When an object is inserted into a Bloom filter, however, multiple indices are selected using multiple hash functions. The record at each location is then updated to indicate the presence of the newly inserted object (e.g., by setting a presence bit or incrementing a counter). To test for the presence of an object in the filter set, the locations corresponding to the result of each hash function are inspected. If any one of them is zero, the object is guaranteed not to be a member of the set.

The filters used to track overflowed cache blocks in LogTM-Bcast are filled during the execution of a large transaction and cleared upon its completion. Because cache blocks are never removed from the read or write set of an active transaction, there is no need to store a count at each filter location. This makes the fil-

ters smaller and removes any concern over saturating the counters. These filters are similar to the Bloom filters employed in JETTY [40] and transaction signatures proposed by Ceze et al. [8].

4.3 Implementing Conflict Resolution

LogTM-Dir and LogTM-Bcast resolve conflicts using the *Requester Stalls* policy [7]. Both systems logically order transactions using TLR's distributed timestamp method [46]. To guarantee forward progress and reduce aborts, Requester Stalls only aborts a transaction that (a) could introduce deadlock and (b) is logically later than the transaction with which it conflicts. LogTM-Dir and LogTM-Bcast detect potential deadlock by recognizing the situation in which one transaction is both waiting for a logically earlier transaction and causing a logically earlier transaction to wait. This is implemented with a per-processor `possible_cycle` flag, which is set if a processor sends a nack to a logically earlier transaction. Under the Requester Stalls policy, a processor aborts its transaction if it receives a nack from a logically earlier transaction while its possible_cycle flag is set.

4.3.1 Write Set Prediction

Because LogTM's eager version management makes aborts more costly, reducing the frequency of aborts can significantly improve performance. Transaction conflicts occur whenever the read or write set of one active transactions overlaps with the write set of another. Many of these conflicts can be resolved by stalling rather than aborting one of the transactions. Aborts are only necessary when a cycle forms between waiting processors. Frequently, this occurs when transactions read a memory location then write to that same location later in the transaction. This type of cycle can be broken by write set prediction—eagerly acquiring exclusive access on the first load to memory locations that will be modified later in the transaction. By setting the W bit early, the system detects the conflict while it is still possible to serialize the transactions.

TABLE 4-1.

Predictor	State	Description
SINGLE-ENTRY	1 address per static transaction	Tracks 1 address per static location, updated on the first store of the transaction.
LOAD-PC	map of load target address to PC, list of upgrade PCs	Tracks the PC of each load by target address in a transaction. If a load target is stored to, the associated PC is added to the upgrade list.
SOFTWARE	None	???

In their comparison of HTM systems, Bobba et al. found that the conflict resolution policy can have a significant impact on the performance of an HTM [7]. They also show that the Requester Stalls policy performs well in comparison to other schemes when combined with write set prediction.