Flexible Decoupled Transactional Memory Support*

Arrvindh Shriraman Sandhya Dwarkadas Michael L. Scott
Department of Computer Science, University of Rochester
{ashriram,sandhya,scott}@cs.rochester.edu

Abstract

A high-concurrency transactional memory (TM) implementation needs to track concurrent accesses, buffer speculative updates, and manage conflicts. We present a system, FlexTM *(FLEXible Transactional Memory), that coordinates four decoupled hardware mechanisms: read and write* signatures*, which summarize per-thread access sets; per-thread* conflict summary tables (CSTs)*, which identify the threads with which conflicts have occurred;* Programmable Data Isolation*, which maintains speculative updates in the local cache and employs a thread-private buffer (in virtual memory) in the rare event of overflow; and* Alert-On-Update*, which selectively notifies threads about coherence events. All mechanisms are software-accessible, to enable virtualization and to support transactions of arbitrary length. FlexTM allows software to determine when to manage conflicts (either eagerly or lazily), and to employ a variety of conflict management and commit protocols. We describe an STM-inspired protocol that uses CSTs to manage conflicts in a distributed manner (no global arbitration) and allows parallel commits.*

In experiments with a prototype on Simics/GEMS, FlexTM exhibits ∼5× speedup over high-quality software TM, with no loss in policy flexibility. Its distributed commit protocol is also more efficient than a central hardware manager. Our results highlight the importance of flexibility in determining when to manage conflicts: lazy maximizes concurrency and helps to ensure forward progress while eager provides better overall utilization in a multi-programmed system.

1 Introduction

Transactional Memory (TM) addresses one of the key challenges of programming multi-core systems: the complexity of lock-based synchronization. At a high level, the programmer or compiler labels sections of the code in a single thread as *atomic*. The underlying system is expected to execute this code atomically, consistently, and in isolation from other transactions, while exploiting as much concurrency as possible.

Most TM systems execute transactions speculatively, and must thus be prepared for *data conflicts*, when concurrent transactions access the same location and at least one of the accesses is a write. *Conflict detection* refers to the mechanism by which such conflicts are identified. *Conflict management* is responsible for arbitrating between conflicting transactions and deciding which should abort. Pessimistic (eager) systems perform both conflict detection and conflict management as soon as possible. Optimistic (lazy) systems delay conflict management until commit time (though they may *detect* conflicts earlier). TM systems must also perform *version management*, either buffering new values in private locations (a *redo* log) and making them visible at commit time, or buffering old values (an *undo* log) and restoring them on aborts. In the taxonomy of Moore et al. [23], undo logs are considered an orthogonal form of eagerness (they put updates in the "right" location optimistically); redo logs are considered lazy.

The mechanisms required for conflict detection, conflict management, and version management can be implemented in hardware (HTM) [1, 12, 14, 23, 24], software (STM) [9, 10, 13, 19, 25], or some hybrid of the two (HyTM) [8, 16, 22, 30]. Full hardware systems are typically inflexible in policy, with fixed choices for eagerness of conflict management, strategies for conflict arbitration and back-off, and eagerness of versioning. Software-only systems are typically slow by comparison, at least in the common case.

Several systems [5,30,36] have advocated *decoupling* of the hardware components of TM, giving each a well-defined API that allows them to be implemented and invoked independently. Hill et al. [15] argue that decoupling makes it easier to refine an architecture incrementally. Shriraman et al. [30] argue that decoupling helps to separate policy from mechanism, allowing software to choose a policy dynamically. Both groups suggest that decoupling may allow TM components to be used for non-transactional purposes [15] [30, TR version].

Several papers have identified performance pathologies with certain policy choices (eagerness of conflict management; arbitration and back-off strategy) in certain applications [4, 28, 30, 32]. RTM [30] promotes policy flexibility by decoupling version management from conflict detection and management—specifically, by separating data and metadata, and performing conflict detection only on the latter. While RTM hardware provides a single mechanism for both conflict detection and management, software can choose (by controlling the timing of metadata inspection and updates) when conflicts are detected. Unfortunately, metadata management imposes significant software costs.

In this paper, we propose more fully decoupled hardware, allowing us to maintain the separation between version management and conflict management without the need for

*This work was supported in part by NSF grants CCF-0702505, CCR-0204344, CNS-0411127, CNS-0615139, and CNS-0509270; NIH grant 1 R21 GM079259-01A1; an IBM Faculty Partnership Award; equipment support from Sun Microsystems Laboratories; and financial support from Intel and Microsoft.

software-managed metadata. Specifically, our FlexTM (FLEXible Transactional Memory) system deploys four decoupled hardware primitives (1) Bloom filter *signatures* (as in Bulk [5] and LogTM-SE [36]) to track and summarize a transaction's read and write sets; (2) *conflict summary tables (CSTs)* to concisely capture conflicts between transactions; (3) the versioning system of RTM (*programmable data isolation—PDI*), adapted to directory-based coherence and augmented with an *Overflow Table (OT)* in virtual memory filled by hardware; and (4) RTM's *Alert-On-Update* mechanism to help transactions ensure consistency without checking on every memory access.

The hardware primitives are fully visible in software, and can be read and written under software control. This allows us to virtualize these structures when context switching and paging, and to use them for non-TM purposes. FlexTM separates conflict detection from management, and leaves software in charge of policy. Simply put, hardware always detects conflicts, and records them in the CSTs, but software chooses when to notice, and what to do about it. Additionally, FlexTM employs a commit protocol that arbitrates between transactions in a distributed fashion and allows parallel commits. This enables lazy conflict management without commit tokens [12], broadcast of write sets [5,12], or ticket-based serialization [7]. FlexTM is, to our knowledge, the first hardware TM in which the decision to commit or abort can be an entirely local operation, even when performed lazily by multiple threads in parallel.

We have developed a 16-core FlexTM CMP prototype on the Simics/GEMS simulation framework, and have investigated performance using benchmarks that stress the various hardware components and software policies. Our results suggest that FlexTM's performance is comparable to that of fixed policy HTMs, and 2× and 5× better than that of hardware accelerated STMs and plain STMs, respectively. Experiments with globally-arbitrated commit mechanisms support the claim that CSTs avoid significant latency and serialization penalties. Finally, experiments indicate that lazy conflict management (for which FlexTM is ideally suited) serves to maximize concurrency and encourages forward progress. Conversely, eager management may maximize overall system utilization in a multi-programmed environment. These results underscore the importance of letting software take policy decisions.

2 Related Work

Transactional memory is a very active area. Larus and Rajwar [17] provide an excellent summary as of fall 2006. We discuss only the most relevant proposals here.

The Bulk system of Ceze et al. [5] decouples conflict detection from cache tags by summarizing read/write sets in Bloom filter signatures [2]. To commit, a transaction broadcasts its write signatures, which other transactions compare to their own read and write signatures to detect conflicts. Conflict management (arbitration) is first-come-first-served, and requires global synchronization in hardware to order commit operations.

LogTM-SE [36] integrates the cache-transparent eager versioning mechanism of LogTM [23] with Bulk style signatures. It supports efficient virtualization (i.e., context switches and paging), but this is closely tied to eager versioning (undo logs), which in turn requires eager conflict detection and management to avoid inconsistent reads. Since LogTM does not allow transactions to abort one another, it is possible for running transactions to "convoy" behind a suspended transaction.

UTM [1] and VTM [24] both perform lazy versioning using virtual memory. On a cache miss (local or forwarded request) in UTM, a hardware controller walks an uncacheable in-memory data structure that specifies access permissions. VTM employs tables maintained in software and uses software routines to walk the table only on cache misses that hit in a locally cached lookaside filter. Like LogTM, both VTM and UTM require eager conflict management.

Hybrid TMs [8,16] allow hardware to handle common-case bounded transactions, and fall back to software for transactions that overflow time and space resources. Hybrid TMs must maintain metadata compatible with the fallback STM and use policies compatible with the underlying HTM. SigTM [22] employs hardware signatures for conflict detection but uses an (always on) TL2 [9] style software redo-log for versioning. Like hybrid systems, it suffers from per-access metadata bookkeeping overheads. It restricts conflict management policy (specifically, only self aborts) and requires expensive commit time arbitration on every speculatively written location.

RTM [30] explored hardware acceleration of STM. Specifically, it introduced (1) Alert-On-Update (AOU), which triggers a software handler when pre-specified lines are modified remotely, and (2) Programmable Data Isolation (PDI), which buffers speculative writes in (potentially incoherent) local caches. Unfortunately, to decouple version management from conflict detection and management, RTM software had to segregate data and metadata, retaining much of the bookkeeping cost of all-software TM systems.

3 FlexTM Architecture

FlexTM provides hardware mechanisms for access tracking, conflict tracking, versioning, and explicit aborts. We describe these separately, then discuss how they work together.

3.1 Access Tracking: Signatures

Like Bulk [5], LogTM-SE [36], and SigTM [22], FlexTM uses Bloom filter *signatures* [2] to summarize the read and write sets of transactions in a concise but conservative fashion (i.e., false positives but no false negatives). Signatures decouple conflict detection from critical L1 tag arrays and enable remote requests to test for conflicts using local processor state without walking in-memory structures, as might be required in [1, 24] in the case of overflow. Every FlexTM processor maintains a *read signature* (R_{sig}) and a write signature (W_{sig}) for the current transaction. The signatures are updated by the processor on transactional loads and stores. They allow the controller to detect conflicts when it receives a remote coherence request.

3.2 Conflict Tracking: CSTs

Existing proposals for both eager [1,23] and lazy [5,12,22] conflict detection track information on a cache-line-by-cache-line basis. FlexTM, by contrast, tracks conflicts on a processor-by-processor basis (virtualized to thread-by-thread). Specifically, each processor has three *Conflict Summary Tables (CSTs)*, each of which contains one bit for every other processor in the system. Named *R-W*, *W-R*, and *W-W*, the CSTs indicate that a local read (R) or write (W) has conflicted with a read or write (as suggested by the name) on the corresponding remote processor. On each coherence request, the controller reads the local W_{sig} and R_{sig}, sets the local CSTs accordingly, and includes information in its response that allows the requestor to set its own CSTs to match.

3.3 Versioning Support: PDI

RTM [30] proposed a lazy versioning mechanism (*programmable data isolation (PDI)*) that allowed software to exploit incoherence (when desired) by utilizing the inherent buffering capabilities of private caches. Programs use explicit *TLoad* and *TStore* instructions to inform the hardware of transactional memory operations: *TStore* requests isolation of a speculative write, whose value will not propagate to other processors until commit time. *TLoad* allows local caching of (previous values of) remotely *TStore*d lines. When speculatively modified state fits in the private cache, PDI avoids the latency and bandwidth penalties of logging.

FlexTM adapts PDI to a directory protocol and simplifies the management of speculative reads, adding only two new stable states to the base MESI protocol, rather than the five employed in RTM. Details appear in Figure 1.

FlexTM's base protocol for private L1s and a shared L2 is an adaptation of the SGI ORIGIN 2000 [18] directory-based MESI, with the directory maintained at the L2 (Figure 2). Local L1 controllers respond to both the requestor and the directory (to indicate whether the cache line has been dropped or retained). Requestors issue a *GETS* on a read (Load/*TLoad*) miss in order to get a copy of the data, a *GETX* on a normal write (Store) miss/upgrade in order to gain exclusive access and an updated copy (in case of a miss), and a *TGETX* on a transactional store (*TStore*) miss/upgrade.

A *TStore* results in a transition to the *TMI* state in the L1 cache (encoded by setting both the T bit and the MESI dirty bit—Figure 2). A *TMI* line reverts to *M* on commit (propagating the speculative modifications) and to *I* on abort (discarding speculative values). On the first *TStore* to a line in *M*, TMESI writes back the modified line to L2 to ensure subsequent Loads get the latest non-speculative version. To the directory, the local *TMI* state is analogous to the conventional *E* state. The directory realizes that the processor can transition to *M* (silent upgrade) or *I* (silent eviction), and any data request needs to be forwarded to the processor to detect the latest state. The only modification required at the directory is the ability to support multiple owners. We do this by extending the existing support for multiple sharers and use the modified bit to distinguish between multiple readers and multiple writers. We add requestors to the sharer list when they issue a *TGETX* request and ping all of them on other requests. On remote requests for a *TMI* line, the L1 controller sends a *Threatened* response, analogous to the *Shared* response to a *GETS* request on an *S* or *E* line.

In addition to transitioning the cache line to *TMI*, a *TStore* also updates the W_{sig}. *TLoad* likewise updates the R_{sig}. *TLoad*s when threatened move to the *TI* state, encoded by setting the T bit when in the I (invalid) state. (Note that a *TLoad* from *E* or *S* can never be threatened; the remote transition to *TMI* would have moved the line to *I*.) *TI* lines must revert to *I* on commit or abort, because if a remote processor commits its speculative *TMI* block, the local copy could go stale. The *TI* state appears as a conventional sharer to the directory.

FlexTM enforces the single-writer or multiple-reader invariant for non-transactional lines. For transactional lines, it enforces (1) *TStore*s can only update lines in *TMI* state, and (2) *TLoad*s that are threatened can cache the block only in *TI* state. Software is expected to ensure that at most one of the conflicting transactions commits. It can restore coherence to the system by triggering an *Abort* on the remote transaction's cache, without having to re-acquire exclusive access to store sets. Previous lazy protocols [5,12] forward invalidation messages to the sharers of the store-set and enforce coherence invariants at commit time. In contrast, TMESI forwards invalidation messages at the time of the individual *TStore*s, and arranges for concurrent transactional readers (writers) to use the *TI* (*TMI*) state; software can then control when (and if) invalidation happens.

Transaction commit is requested with a special variant of the CAS (compare-and-swap) instruction. Like a normal CAS, *CAS-Commit* fails if it does not find an expected value in memory. It also fails if the caller's *W-W* or *W-R* CST is nonzero. As a side effect of success, it simultaneously reverts all local *TMI* and *TI* lines to *M* and *I*, respectively (achieved by flash clearing the T bits). On failure, *CAS-Commit* leaves transactional state intact in cache. Software can clean up transactional state by issuing an *ABORT* to the controller that reverts all *TMI* and *TI* lines to *I* (achieved by conditionally clearing the M bits based on the T bits and then flash clearing the T bits).

Conflict Detection On forwarded L1 requests from the directory, the local cache controller tests the signatures and appends an appropriate message type to its response, as shown in the table in Figure 1. *Threatened* indicates a write conflict (hit in the W_{sig}), *Exposed-Read* indicates a read conflict (hit in the R_{sig}), and *Shared* or *Invalidated* indicate no conflict. On a miss in the W_{sig}, the result from testing the R_{sig} is used; on a miss in both, the L1 cache responds as in normal MESI. The local controller also piggybacks a data response if the block is currently in *M* state. When it sends a *Threatened* or *Exposed-Read* message, a responder sets the bit corresponding to the requestor in its *R-W*, *W-W*, or *W-R* CSTs, as appropriate. The requestor likewise sets the bit corresponding to the responder in its own CSTs, as appropriate, when it receives the response.

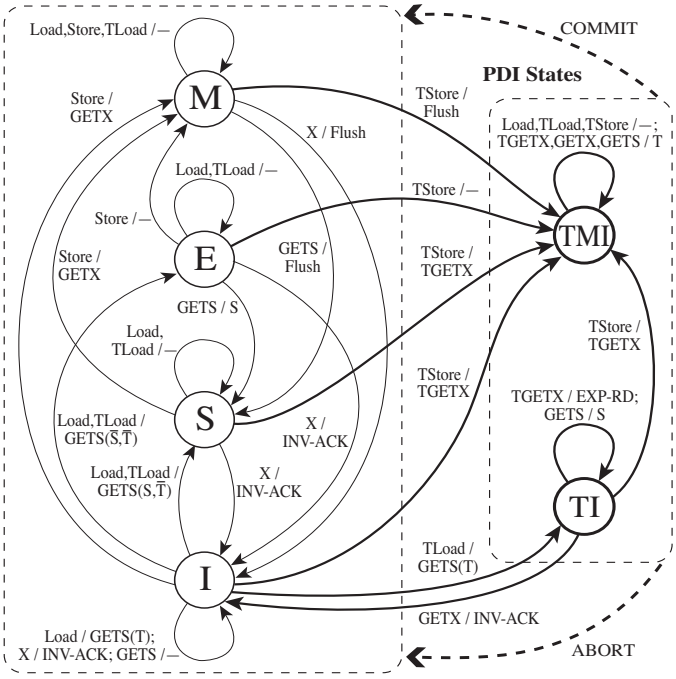


Figure 1: Dashed boxes enclose the MESI and PDI subsets of the state space. Notation on transitions is conventional: the part before the slash is the triggering message; after is the ancillary action ('–' means none). GETS indicates a request for a valid sharable copy; GETX for an exclusive copy; TGETX for a copy that can be speculatively updated with *TStore*. X stands for the set {GETX, TGETX}. "Flush" indicates a data block response to the requestor and directory. S indicates a *Shared* message; T a *Threatened* message. Plain, they indicate a response by the local processor to the remote requestor; parenthesized, they indicate the message that accompanies the response to a request. An overbar means logically "not signaled".

State Encoding

	MESI		
	M bit	V bit	T bit
M	1	0	0
E	1	1	0
S	0	1	0
I	0	0	0
TMI	1	0	1
TI	0	0	1

Responses to requests that hit in W_{sig} or R_{sig}

Request Msg	Hit in W_{sig}	Hit in R_{sig}
GETX	Threatened	Invalidated
TGETX	Threatened	Exposed-Read
GETS	Threatened	Shared

Requestor CST set on coherence message

Local op.	Response Message	
	Threatened	Exposed-Read
TLoad	*R-W*	—
TStore	*W-W*	*W-R*

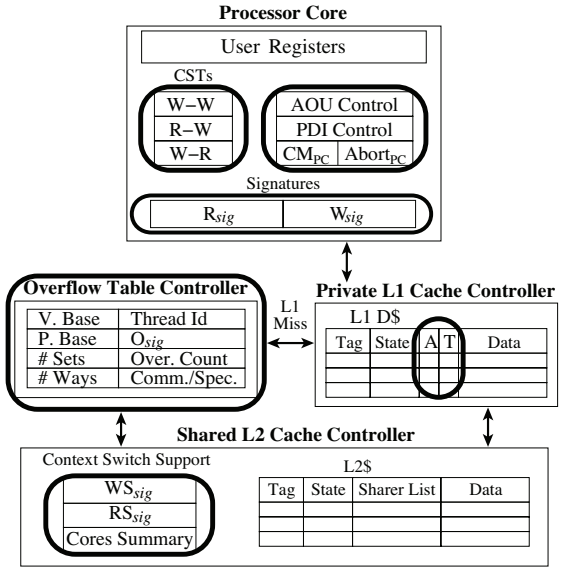


Figure 2: FlexTM Architecture Overview (dark lines surround FlexTM-specific state).

3.4 Explicit Aborts: AOU

The *Alert-On-Update (AOU)* mechanism, borrowed from RTM [30], supports synchronous notification of conflicts. To use AOU, a program marks (*ALoad*s) one or more cache lines, and the cache controller effects a subroutine call to a user-specified handler if the marked line is invalidated. Alert traps require simple additions to the processor pipeline. Modern processors already include trap signals between the Load-Store-Unit (LSU) and Trap-Logic-Unit (TLU) [35]. AOU adds an extra message to this interface and an extra mark bit, 'A', to each line in the L1 cache. (An overview of the FlexTM hardware required in the processor core, the L1 controller, and the L2 controller appears in Figure 2.) RTM used AOU to detect software-induced changes to (a) transaction status words (indicating an abort) and (b) the metadata associated with objects accessed in a transaction (indicating conflicts). FlexTM uses AOU for abort detection only; it performs conflict detection using signatures and CSTs instead of metadata. Hence, FlexTM requires AOU support for only one cache line (i.e., the transaction status word; see Section 3.6) and can therefore use the simplified hardware mechanism (avoiding the bit per cache tag) as proposed in [33]. More general AOU support might still be useful for non-transactional purposes.

3.5 Programming Model

A FlexTM transaction is delimited by `BEGIN_TRANSACTION` and `END_TRANSACTION` macros. The first of these establishes conflict and abort handlers for the transaction, checkpoints the processor registers, configures per-transaction metadata, sets the transaction status word (TSW) to `active`, and *ALoad*s that word (for notification of aborts). The second macro aborts conflicting transactions and tries to atomically update the status word from `active` to `committed` using *CAS-Commit*. In this paper, we assume a subsumption model for nesting, with support for transactional pause [37].

Within a transaction, a processor issues *TLoad*s and *TStore*s when it expects transactional semantics, and conventional loads and stores when it wishes to bypass those semantics. While one could imagine requiring the compiler to generate instructions as appropriate, our prototype implementation follows typical HTM practice and interprets ordinary loads and stores as *TLoad*s and *TStore*s when they occur within a transaction. This convention facilitates code sharing between transactional and nontransactional program fragments. Ordinary loads and stores

can be requested within a transaction by issuing special instructions; while not needed in our experiments, these could be used to implement open nesting, update software metadata, or reduce the cost of thread-private updates in transactions that overflow cache resources (Section 4).

As implied in Figure 1, transactional and ordinary loads and stores to the same location can occur concurrently. While we are disinclined to require strong isolation [3] as part of the user programming model (it's hard to implement on legacy hardware, and is of questionable value to the programmer [34]), it can be supported at essentially no cost in HTM systems (FlexTM among them), and we see no harm in providing it. If the GETX request resulting from a nontransactional write miss hits in the responder's R_{sig} or W_{sig}, it aborts the responder's transaction, so the nontransactional write appears to serialize before the (retried) transaction. A nontransactional read, likewise, serializes before any concurrent transactions, because transactional writes remain invisible to remote processors until commit time (in order to enforce coherence, the corresponding cache line, which is threatened in the response, is uncached).

3.6 Bounded Transactions

In this section, we focus on transactions that fit in the L1 cache and complete within an OS quantum.

Name	Description
TSW	`active` / `committed` / `aborted`
State	`running` / `suspended`
R_{sig}, W_{sig}	Signatures
R-W, *W-R*, *W-W*	Conflict Summary Tables
OT	Pointer to Overflow Table descriptor
$Abort_{PC}$	Handler address for AOU on TSW
CM_{PC}	Handler address for *Eager* conflicts
E/L	*Eager*(1)/*Lazy*(0) conflict detection.

Table 1: Transaction Descriptor contents. All fields except TSW and State are cached in hardware registers for transactions running.

Every FlexTM transaction is represented by a software *descriptor* (Table 1). Transactions of a given application can operate in either *Eager* or *Lazy* conflict detection mode. In *Eager* mode, when conflicts appear through response messages (i.e., *Threatened* and *Exposed-Read*), the processor effects a subroutine call to the handler specified by CM_{PC}. The conflict manager either stalls the requesting transaction or aborts one of the conflicting transactions. The remote transaction can be aborted by atomically updating its TSW from `active` to `aborted`, thereby triggering an alert (since the TSW is always *ALoad*ed). FlexTM supports a wide variety of conflict management policies (even policies that desire the ability to synchronously abort a remote transaction). When an *Eager* transaction reaches its commit point, its CSTs will be empty, since all prior conflicts will have been resolved. It attempts to commit by executing a *CAS-Commit* on its TSW. If the *CAS-Commit* succeeds (replacing `active` with `committed`), the hardware flash-commits all locally buffered (*TMI*) state. The *CAS-Commit* will fail leaving the buffered state intact if the CAS does not find the expected value (a remote transaction managed to abort the committing transaction before the *CAS-Commit* could complete).

In *Lazy* mode, transactions are not alerted into the conflict manager. The hardware simply updates requestor and responder CSTs. To ensure serialization, a *Lazy* transaction must, prior to committing, abort every concurrent transaction that conflicts with its write-set. It does so using the *Commit()* routine shown in Figure 3.

```
Commit()   /* Non-blocking, pre-emptible */
1.  copy-and-clear W-R and W-Wregisters
2.  foreach i set in W-R or W-W
3.      abort_id = manage_conflict(my_id, i)
4.      if (abort_id ≠ NULL)        // not resolved by waiting
5.          CAS(TSW[abort_id], active, aborted)
6.  CAS-Commit(TSW[my_id], active, committed)
7.  if (TSW[my_id] == active)       // failed due to nonzero CST
8.      goto 1
```

Figure 3: Simplified Commit Routine for *Lazy* transactions.

All of the work for the *Commit()* routine occurs in software, with no need for global arbitration [5, 7, 12], blocking of other transactions [12], or special hardware states. The routine begins by using a copy and clear instruction (e.g., `clruw` on the SPARC) to atomically access its own *W-R* and *W-W*. In lines 2–5 of Figure 3, for each of the bits that was set, transaction T aborts the corresponding transaction R by atomically changing R's TSW from `active` to `aborted`. Transaction R, of course, could try to *CAS-Commit* its TSW and race with T, but since both operations occur on R's TSW, conventional cache coherence guarantees serialization. After T has successfully aborted all conflicting peers, it performs a *CAS-Commit* on its own status word. If the *CAS-Commit* fails and the failure can be attributed to a non-zero *W-R* or *W-W* (i.e., new conflicts), the *Commit()* routine is restarted. To avoid subsequent spurious aborts, T may also clean itself out of X's *W-R*, where X is the transaction in T's *R-W* (not shown).

4 Unbounded Space Support

For common case transactions that do not overflow the cache, signatures, CSTs, and PDI avoid the need for logging or other per-access software overhead. To provide the illusion of unbounded space, however, FlexTM must support transactions in the presence of (1) L1 cache overflows and (2) physical memory virtualization (i.e., paging).

4.1 Cache Evictions

Cache evictions must be handled carefully in FlexTM. First, signatures rely on forwarded requests from the directory to trigger lookups and provide conservative conflict hints (*Threatened* and *Exposed-Read* messages). Second, *TMI* lines holding speculative values need to be buffered and cannot be merged into the shared level of the cache.

Conventional MESI performs silent eviction of *E* and *S* lines to avoid the bandwidth overhead of notifying the directory. In

FlexTM, silent evictions of *E*, *S*, and *TI* lines also serve to ensure that a processor continues to receive the coherence requests it needs to detect conflicts. (Directory information is updated only in the wake of L1 responses to L2 requests, at which point any conflict is sure to have been noticed.) When evicting a cache block in *M*, FlexTM updates the L2 copy but does not remove the processor from the sharer list. Processor sharer information can, however, be lost due to L2 evictions. To preserve the access conflict tracking mechanism, L2 misses result in querying all L1 signatures in order to recreate the sharer list. This scheme is much like the *sticky bits* used in LogTM [23].

FlexTM employs a per-thread *overflow table (OT)* to buffer evicted *TMI* lines. The OT is organized as a hash table in virtual memory. It is accessed both by software and by an OT controller that sits on the L1 miss path. The latter implements (1) fast lookups on cache misses, allowing software to be oblivious to the overflowed status of a cache line, and (2) fast cleanup and atomic commit of overflowed state.

The controller registers required for OT support appear in Figure 2. They include a thread identifier, a signature (O_{sig}) for the overflowed cache lines, a number count of such lines, a committed/speculative flag, and parameters (virtual and physical base address, number of sets and ways) used to index into the table.

On the first overflow of a *TMI* cache line, the processor traps to a software handler, which allocates an OT, fills the registers in the OT controller, and returns control to the transaction. To minimize the state required for lookups, the current OT controller design requires the OS to ensure that OTs of active transactions lie in physically contiguous memory. If an active transaction's OT is swapped out, then the OS invalidates the Base-Address register in the controller. If subsequent activity requires the OT, the hardware traps to a software routine that re-establishes a mapping. The hardware needs to ensure that new *TMI* lines aren't evicted during OT set-up; the L1 cache controller could easily support this routine by ensuring at least one entry in the set is free for non-*TMI* lines. On a subsequent *TMI* eviction, the OT controller calculates the set index using the physical address of the line, accesses the set tags of the OT region to find an empty way, and writes the data block back to the OT instead of the L2. The controller tags the line with both its physical address (used for associative lookup) and its virtual address (used to accommodate page-in at commit time; see below). The controller also adds the physical address to the *overflow signature* (O_{sig}) and increments the *overflow count*.

The O_{sig} provides quick lookaside checks for entries in the OT. Reads and writes that miss in the L1 are checked against the signature. Signature hits trigger the L1-to-L2 request and the OT lookup in parallel. On OT hits, the line is fetched from the OT, the corresponding OT tag is invalidated, and the L2 response is squashed. This scheme is analogous to the speculative memory request issued by the home memory controller before snoop responses are all collected. When a remote request hits in the O_{sig} of a committed transaction, the controller could perform lookup in the OT, much as it does for local requests, or it could NACK the request until copy-back completes. Our current implementation does the latter.

In addition to functions previously described, the *CAS-Commit* operation sets the Committed bit in the controller's OT state. This indicates that the OT content should be visible, activating NACKs or lookups. At the same time, the controller initiates a microcoded copy-back operation. There are no constraints on the order in which lines from the OT are copied back to their natural locations. This stands in contrast to time-based undo logs [23], which must proceed in reverse order of insertion. Remote requests need to check only committed OTs (since speculative lines are private) and for only a brief span of time (during OT copy-back). On aborts, the OT is returned to the operating system. The next overflowed transaction allocates a new OT. When an OT overflows a way, the hardware generates a trap to the OS, which expands the OT appropriately.

With the addition of the OT controller, software is involved only for the allocation and deallocation of the OT structure. Indirection to the OT on misses, while unavoidable, is performed in hardware rather than in software, thereby reducing the resulting overheads. Furthermore, FlexTM's copyback is performed by the controller and occurs in parallel with other useful work on the processor.

Virtual Memory Paging Though presumably infrequent, page faults may nonetheless occur in the middle of a transaction. To accommodate paging of the original locations, OT tags include the virtual addresses of cache blocks. These addresses are used during copy-back, to ensure automatic page-in of any nonresident pages. Though for simplicity we currently require that OTs be physically contiguous, they can themselves be paged, albeit as a single unit. In particular, it makes sense for the OS to swap out the OTs of descheduled threads. A more ambitious FlexTM design could allow physically non-contiguous OTs, with controller access mediated by more complex mapping information.

The two challenges left to consider are (1) when a page is swapped out and its frame is reused for a different page in the application, and (2) when a page is re-mapped to a different frame. Since signatures are built using physical addresses, (1) can lead to false positives, which can cause spurious aborts but not correctness issues. In a more ambitious design, we could solve this problem with virtual address-based conflict detection for non-resident pages.

For (2) we adapt a solution first proposed in LogTM-SE [36]. At the time of the unmap, active transactions are interrupted both for TLB entry shootdown (already required) and to flush TMI lines to the OT. When the page is assigned to a new frame, the OS interrupts all the threads that mapped the page and tests each thread's R_{sig}, W_{sig}, and O_{sig} for the old address of each block. If the block is present, the new address is inserted into the signatures. The corresponding tags of the OT entries are also updated with the new physical address.

5 Context Switch Support

STMs provide effective virtualization support because they maintain conflict detection and versioning state in virtualizable locations and use software routines to manipulate them. For common case transactions, FlexTM uses scalable hardware support to bookkeep the state associated with access permissions, conflicts, and versioning while controlling policy in software. In the presence of context switches, FlexTM detaches the transactional state of suspended threads from the hardware and manages it using software routines. This enables support for transactions to extend across context switches (i.e., to be unbounded in time [1]).

Ideally, only threads whose actions overlap with the read and write set of suspended transactions should bear the software routine overhead. To track the accesses of descheduled threads, FlexTM maintains two summary signatures, RS_{sig} and WS_{sig}, at the directory of the system. When suspending a thread in the middle of a transaction, the OS unions (i.e., ORs) the signatures (R_{sig} and W_{sig}) of the suspended thread into the current RS_{sig} and WS_{sig} installed at the directory.[1]

Once the RS_{sig} and WS_{sig} are up to date, the OS invokes hardware routines to merge the current transaction's hardware state into virtual memory. This hardware state consists of (1) the *TMI* lines in the local cache, (2) the OT registers, (3) the current R_{sig} and W_{sig}, and (4) the CSTs. After saving this state (in the order shown), the OS issues an *abort* instruction, causing the cache controller to revert all *TMI* and *TI* lines to *I*, and to clear the signatures, CSTs, and OT registers. This ensures that any subsequent conflicting access will miss in the private cache and generate a directory request. In other words, *for any given location, the first conflict between the running thread and a local descheduled thread always results in an L1 miss*. The L2 controller consults the summary signatures on each such miss, and traps to software when a conflict is detected.[2]

On summary hits a software handler mimics hardware operations on a per-thread basis, testing signature membership and updating the CSTs of suspended transactions. No special instructions are required, since the CSTs and signatures of descheduled threads are all visible in virtual memory. Nevertheless, updates need to be performed atomically to ensure consistency when multiple active transactions conflict with a common descheduled transaction and update the CSTs concurrently. The OS helps the handler distinguish among transactions running on different processors. It maintains a global *conflict management table (CMT)*, indexed by processor id, with the following invariant: *if transaction T is active, and has executed on processor P, irrespective of the state of the thread (suspended/running), the transaction descriptor will be included in P's portion of the CMT.* The handler uses the processor ids in its CST to index into the CMT and to iterate through transaction descriptors, testing the saved signatures for conflicts, updating the saved CSTs (if running in lazy mode), or invoking conflict management (if running in eager mode). Similar perusal of the CMT occurs at commit time if running in lazy mode. As always, we abort a transaction by writing its TSW. If the remote transaction is running, an alert is triggered since it would have previously *ALoad*ed its TSW. Otherwise, the OS virtualizes the AOU operation by causing the transaction to wake up in a software handler that checks and re-*ALoad*s the TSW.

The directory needs to ensure that sticky bits are retained when a transaction is suspended. Along with RS_{sig} and WS_{sig}, the directory maintains a bitmap indicating the processors on which transactions are currently descheduled (the "Cores Summary" register in Figure 2). When the directory would normally remove a processor from the sharers list (because a response to a coherence request indicates that the line is no longer cached), the directory refrains from doing so if the processor is in the Cores Summary list and the line hits in RS_{sig} or WS_{sig}. This ensures that the L1 continues to receive coherence messages for lines accessed by descheduled transactions. It will need these messages if the thread is swapped back in, even if it never reloads the line.

When re-scheduling a thread, if the thread is being scheduled back to the same processor from which it was swapped out, the thread's R_{sig}, W_{sig}, CST, and OT registers are restored on the processor. The OS then re-calculates the summary signatures for the currently swapped out threads with active transactions and re-installs them at the directory. Thread migration is a little more complex, since FlexTM performs write buffering and does not re-acquire ownership of previously written cache lines. To avoid the inherent complexity, FlexTM adopts a simple policy for migration: abort and restart.

Unlike LogTM-SE [36], FlexTM is able to place the summary signature at the directory rather than on the path of every L1 access. This scheme does not require inter-processor interrupts to install summary signatures. Since speculative state is flushed from the local cache when descheduling a transaction, the first access to a conflicting line after re-scheduling is guaranteed to miss, and the conflict will be caught by the summary signature at the directory. Because it is able to abort remote transactions using AOU, FlexTM also avoids the problem of potential convoying behind suspended transactions.

6 Area Analysis

In this section, we briefly summarize the area overheads of FlexTM. Further details can be found in a technical report [31]. Area estimates appear in Table 2. We consider processors from a uniform (65nm) technology generation to better understand microarchitectural tradeoffs. Processor component sizes were

[1] FlexTM updates RS_{sig} and WS_{sig} using a *Sig* message that uses the L1 coherence request network and carries the processor's R_{sig} and W_{sig}. The directory updates the summary signatures and returns an ACK on the forwarding network. This avoids races between the ACK and remote requests that were forwarded to the suspending thread/processor before the summary signatures were updated.

[2] *TStore* to an *M* line generates a write-back (see Figure 1) that also tests the RS_{sig} and WS_{sig} for conflicts. This resolves the corner case in which a suspended transaction *TLoad*ed a line in *M* and a new transaction on the same processor *TStore*s it.

estimated using published die images. FlexTM component areas were estimated using CACTI 6.

Only for the 8-way multithreaded Niagara-2 do the R_{sig} and W_{sig} have a noticeable area impact: 2.2%; on Merom and Power6 they add only ∼0.1%. CACTI indicates that the signatures should be readable and writable in less than the L1 access latency. These results appear to be consistent with those of Sanchez et al. [27]. The CSTs for their part are full-map bit-vector registers (as wide as the number of processors), and we need only three per hardware context. Also, we do not expect the extra state bits in the L1 to affect the access latency because (a) they have minimal impact on the cache area and (b) the state array is typically accessed in parallel with the higher latency data array. Finally, the OT controller adds less than 0.5% to core area. Moreover, its state machine is similar to Niagara-2's TSB walker [35]. Overall, FlexTM's add-ons have noticeable area impact (∼2.6%) only in the case of high core multithreading. The overheads imposed on out-of-order CMP cores (Merom and Power6) are well under 1%.

Processor	Merom [26]	Power6 [11]	Niagara-2 [35]
Actual Die			
SMT (threads)	1	2	8
Core (mm^2)	31.5	53	11.7
L1 D (mm^2)	1.8	2.6	0.4
CACTI Prediction			
R_{sig} + W_{sig} (mm^2)	.033	.066	0.26
RS_{sig} + WS_{sig} (mm^2)	.033	.033	0.033
CSTs (registers)	3	6	24
OT controller (mm^2)	0.16	0.24	0.035
Extra state bits	2(T,A)	3(T,A,ID)	5(T,A,ID)
% Core increase	0.6%	0.59%	2.6%
% L1 Dcache increase	0.35%	0.29%	3.9%

Table 2: Area Estimation. "ID" = SMT context of TMI line.

7 FlexTM Evaluation

7.1 Evaluation Framework

We evaluate FlexTM through full system simulation of a 16-way chip multiprocessor (CMP) with private L1 caches and a shared L2 (see Table 3(a)), on the GEMS/Simics infrastructure [21]. We added support for the FlexTM instructions using the standard Simics "magic instruction" interface. Our base protocol is an adaptation of the SGI ORIGIN 2000 [18] for a CMP, extended to support FlexTM's requirements: signatures, CSTs, PDI, and AOU. Software routines (`setjmp`) were used to checkpoint registers.

Simics allows us to run an unmodified Solaris 9 kernel. Simics also provides a "user-mode-change" and "exception-handler" interface, which we use to trap user-kernel mode crossings. On crossings, we suspend the current transaction mode and allow the OS to handle TLB misses, register-window overflow, and other kernel activities required by an active user context in the midst of a transaction. On transfer back from the kernel, we deliver any alert signals received during the kernel routine, triggering the alert handler if needed.

7.2 Runtime Systems

We evaluate FlexTM using the seven benchmarks described in Table 3(b). In the data-structure tests, we execute a fixed number of transactions in a single thread to warm up the structure, then fork off threads to perform the timed transactions. Workload set 1 (WS1) interfaces with three TM systems: (1) FlexTM; (2) RTM-F [30], a hardware accelerated STM system; and (3) RSTM [20], a non-blocking STM for legacy hardware (configured to use invisible readers, with self validation for conflict detection). Workload set 2 (WS2), which uses a different API, interfaces with (1) FlexTM and (2) TL2, a blocking STM for legacy hardware [9]. We use the "Polka" conflict manager [28] across all systems. While all runtime systems execute on our simulated hardware, RSTM and TL2 make no use of FlexTM extensions. RTM-F uses only PDI and AOU. FlexTM uses all the presented mechanisms.

7.3 Throughput and Scalability

Result 1: *Separable hardware support for conflict detection, conflict management, and versioning can provide significant acceleration for software controlled TMs; eliminating software bookkeeping from the common case critical path is essential to realize the full benefits of hardware acceleration.*

Figure 4 shows normalized throughput (transactions/sec.) across our applications and systems. FlexTM, RTM-F, and RSTM have all been set up to perform eager conflict management (TL2 is inherently lazy). Throughput is normalized to that of single-thread coarse-grain locks (CGL), which is very close to sequential thread performance. To illustrate the usefulness of CSTs (see the table in Figure 4), we also report the number of conflicts encountered and resolved by an average transaction—the number of bits set in the *W-R* and *W-W* CST registers.

STM performance suffers from the bookkeeping required to track data versions (copying), detect conflicts, and guarantee a consistent view of memory (validation). RTM-F exploits AOU and PDI to eliminate validation and copying but still incurs bookkeeping overhead accounting for 40–50% of execution time. FlexTM's hardware tracks conflicts, buffers speculative state, and ensures consistency in a manner transparent to software, resulting in single thread performance close to that of CGL. FlexTM's main overhead, register checkpointing, involves spilling of local registers into the stack and is nearly constant across thread levels. Eliminating per-access software overheads (metadata tracking, validation, and copying) allows FlexTM to realize the full potential of hardware acceleration, with an average speedup of 2× over RTM-F, 5.5× over RSTM, and 4.5× over TL2.

HashTable and RBTree both scale well. In RSTM, validation and copying account for 22% of execution time in HashTable and 50% in RBTree; metadata management accounts for 40% and 30%, respectively. Tree rebalancing in RBTree is non-trivial: insertion proceeds bottom-up while searching moves top-down. At higher thread levels, eager conflict management precludes read-write sharing and increases the likelihood of

(a) Target System Parameters

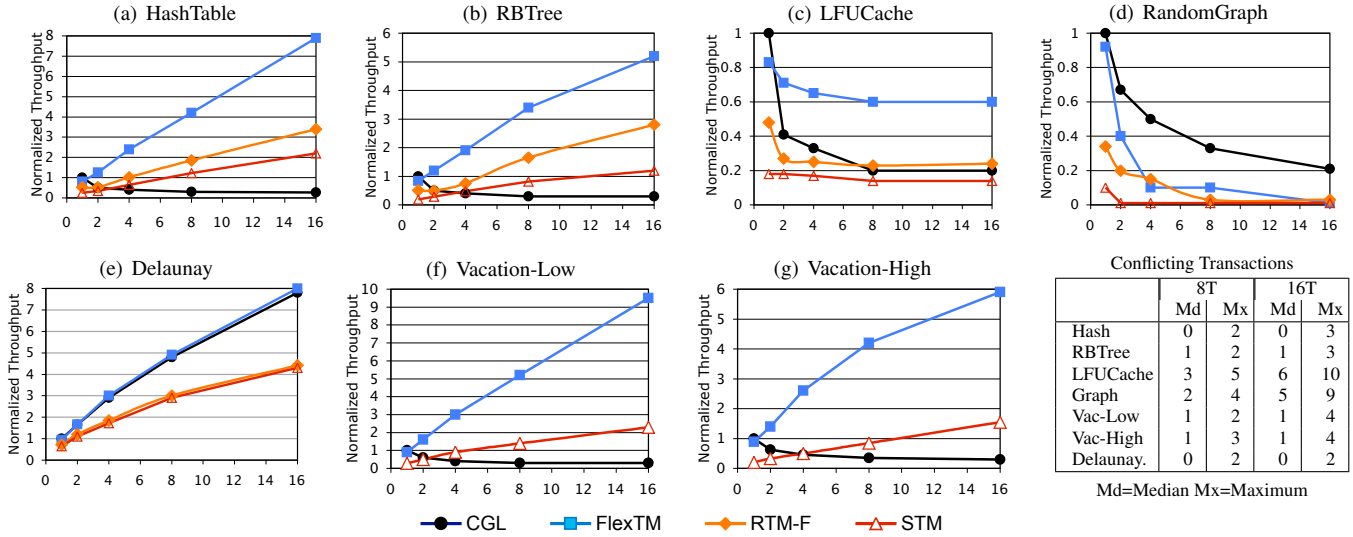
16-way CMP, Private L1, Shared L2	
Processor Cores	16 1.2GHz in-order, single issue; non-memory IPC=1
L1 Cache	32KB 2-way split, 64-byte blocks, 1 cycle, 32 entry victim buffer, 2Kbit signature [5, S14]
L2 Cache	8MB, 8-way, 4 banks, 64-byte blocks, 20 cycle
Memory	2GB, 250 cycle latency
Interconnect	4-ary tree, 1 cycle, 64-byte links,
Central Arbiter (Section 7.4)	
Arbiter Lat. Commit Msg. Lat.	30 cycles [6] 16 cycles/link

Commit messages also use the 4-ary tree.

(b) Workload Description

Workload-Set 1
HashTable: Transactions attempt to lookup, insert, or delete (33% each) a value (range 0 . . . 255) with equal probability into a hash table with 256 buckets and overflow chains.
RBTree: Transactions attempt to insert, remove, or delete (33% each) values in the range 0 . . . 4095 with equal probability. At steady state there are about 2048 objects, with 50% of the values in leaves. Node size is 256 bytes.
LFUCache: Simulates a web cache using a large (2048) array based index and a smaller (255 entry) priority queue to track the page access frequency. Pages to be accessed are randomly chosen using a Zipf distribution: $p(i) \propto \Sigma_{0<j\leq i} j^{-2}$.
RandomGraph Transactions insert or delete vertices (50% each) in an undirected graph represented with adjacency lists. Edges are chosen at random, with each new vertex initially having up to 4 randomly selected neighbors.
Delaunay [29] Solves the original triangulation problem. Sorts the points into geometric regions, employs sequential solvers in parallel to triangulate the regions, then uses transactions to "stitch" together the seams.
Workload-Set 2
Vacation [22]: Implements a travel reservations system. Client threads interact with an in-memory database in which tables are implemented as a Red-Black tree. This workload is similar in design to SPECjbb2000. Two contention levels: Low – 90% of relations queried, read-only tasks dominate; High – 10% of relations queried, 50-50 mix of read-only and read-write tasks.

Table 3: Experimental Set-Up



Conflicting Transactions

	8T		16T	
	Md	Mx	Md	Mx
Hash	0	2	0	3
RBTree	1	2	1	3
LFUCache	3	5	6	10
Graph	2	4	5	9
Vac-Low	1	2	1	4
Vac-High	1	3	1	4
Delaunay.	0	2	0	2

Md=Median Mx=Maximum

Figure 4: Throughput (transactions/10^6 cycles), normalized to 1-thread CGL. X-axis specifies the number of threads. In plots (a)-(e) STM represents RSTM [20]; in (f)-(g) it represents TL2 [9].

aborts, though the back-off strategy of the Polka conflict manager limits aborts to about 10% of total transactions committed.

LFUCache and RandomGraph do not scale. Conflict for popular keys in the Zipf distribution forces transactions in LFUCache to serialize. Stalled writers lead to extra aborts with larger numbers of threads, but performance eventually stabilizes for all TM systems. In RandomGraph, larger numbers of more random conflicts cause all TM systems to livelock at higher thread levels, due to eager contention management. The average RandomGraph transaction reads ∼80 cache lines and writes ∼15. In RSTM, read-set validation accounts for 80% of execution time. RTM-F eliminates this overhead, after which per-access bookkeeping accounts for 60% of execution time. FlexTM eliminates this overhead as well, to achieve 2.7× the performance of RTM-F at 1 thread. At higher thread levels, all TM systems livelock due to eager conflict management. In the language of Bobba et al. [4], RandomGraph suffers from the FriendlyFire, FutileStall, and DuellingUpgrade pathologies.

Delaunay [29] is fundamentally data parallel (less than 5% of execution time is spent in transactions) and memory bandwidth limited. FlexTM and CGL track closely out to 16 threads. RSTM and RTM-F also track closely, but at half the throughput, because the extra indirection required for metadata bookkeeping induces a ∼ 2× increase in the number of cache misses.

Vacation is incompatible with the object-based API of RSTM and RTM-F. Therefore, we evaluate its performance on CGL, word-based TL2, and Flex-TM. Transactions read ∼100 entries from a database and stream them through an RBTree. TL2 suffers from the bookkeeping required prior to the first read (i.e., for checking write sets), post-read and commit time validation [9]. FlexTM avoids this bookkeeping, yielding 4× the performance of TL2 at 1 thread. Vacation-Low displays good scalability (Figure 4f): 10× CGL's performance at 16 threads. Vacation-High displays less (Figure 4g): 6× at 16 threads. Multiple threads introduce (1) a mix of read-only (e.g., ticket lookup) and read-write (e.g., ticket reservation) tasks and (2) sets of dueling transactions that try to rotate common subtree nodes. These increase the level of conflicts and aborts.

Idealized Overflow In the event of an overflow, FlexTM buffers new values in a redo log and needs to perform copy-back at commit time. Almost all of our benchmarks use the overflow mechanism sparingly, with a maximum of 5 cache lines overflowed in RandomGraph. Because our benchmarks have small write sets, cache set conflicts account for all cases of overflow. In separate experiments, we extended the L1 with an unbounded victim buffer. In applications with overflows, we found that redo-logging reduced performance by an average of 7% and a maximum of 13% (in RandomGraph) compared to the ideal case, mainly because re-starting transactions have their accesses queued behind the committed transaction's copy-back phase. As expected, benchmarks that don't overflow the cache (e.g., HashTable) experience no slowdown.

7.4 FlexTM vs. Central-Arbiter Lazy HTM

Result 2: *CSTs are useful: transactions don't conflict and even when they do the number of conflicts per transaction is less than the total active transactions. FlexTM's distributed commit demonstrates better scalability than a centralized arbiter.*

As shown in the table at the end of Figure 4, the number of conflicts encountered by a transaction is small compared to the total number of transactions in the system. Even in workloads that have a large number of conflicts (LFUCache and RandomGraph) a typical transaction encounters only 30% of total transactions as conflicts. Scalable workloads (e.g., HashTable and Vacation) encounter essentially no conflict. This clearly suggests that global arbitration and serialized commits will not only waste bandwidth but also restrict concurrency. CSTs enable local arbitration and the distributed commit protocol allows parallel commits thereby unlocking the full concurrency potential of the application.

In this set of experiments, we compare FlexTM's distributed commit against two schemes with centralized hardware arbiters: *Central-Serial* and *Central-Parallel*. In both schemes, instead of using CSTs and requiring each transaction to *ALoad* its TSW, transactions forward their R_{sig} and W_{sig} to a central hardware arbiter at commit time. The arbiter orders each commit request, and broadcasts the W_{sig} to other processors. Every recipient uses the forwarded W_{sig} to check for conflicts and abort its active transaction; it also sends an ACK as a response to the arbiter. The arbiter collects all the ACKs and then allows the committing processor to complete. This process adds 97 cycles to a transaction, assuming unloaded links (latencies are listed in Table 3(a)) and arbiter. The *Serial* version services only one commit request at a time (queuing up any others), while *Parallel* services all non-conflicting transactions in parallel (assumes infinite buffers in the arbiter). *Central* arbiters are similar in spirit to BulkSC [6], but serve only to order commits; they do not interact with the L2 directory.

We present results (see Figure 5) for HashTable, Vacation-Low, LFUCache, and RandomGraph (we eliminate Delaunay since the transaction phases have negligible impact on overall throughput and RBTree since it demonstrates a similar pattern to Vacation). We enumerate the general trends below:

- Arbitration latency for the *Central* commit scheme is on the critical path of transactions. This gives rise to noticeable overhead in the case of short transactions (e.g., HashTable and LFUCache) at all thread levels. CSTs simplify the commit process, in the absence of conflicts, commit requires only a single memory operation on a transaction's cached TSW.
- At higher thread levels, the benchmarks that are inherently parallel (HashTable and Vacation-Low) suffer from serialization of commits in *Central-Serial*, as transactions wait for predecessors in the queue. *Central-Parallel* removes the serialization overhead, but still suffers from commit arbitration latency at all thread levels.
- In benchmarks with high conflicts (e.g., LFUCache and RandomGraph) that don't inherently scale, *Central*'s conflict management strategy avoids performance degradation. The transaction being serviced by the arbiter always commits successfully, ensuring progress and livelock freedom. The current distributed protocol allows the possibility of livelock. However, the CSTs streamline the commit process, narrow the vulnerability window (to essentially the inter-processor message latency) and eliminate the problem as effectively as *Central*.

At low conflict levels, a CST-based commit requires mostly local operations and its performance should be comparable to an ideal *Central-Parallel* (i.e., zero message and arbitration latency). At high conflict levels, the penalties of *Central* are lower compared to the overhead of aborts and workload inherent serialization. Finally, the influence of commit latency on performance is dependent on transaction latency (e.g., reducing commit latency helps *Central-Parallel* approach FlexTM's throughput in HashTable but has negligible impact on RandomGraph's throughput).

7.5 Conflict Management Timing

Result 3a: *When applications get to use the machine in isolation, lazy conflict management exploits available resources to maximize concurrency and encourage forward progress.*

Result 3b: *With multiprogramming, lazy management allows doomed but executing transactions to occupy valuable resources Eager management may free up resources for other useful work.*

Figures 6(a)-(d) show the potential benefit of lazy conflict management in FlexTM—specifically, the ability to eliminate the performance pathologies observed in RBTree, Vacation-High, LFUCache, and RandomGraph. In applications with very few conflicts (e.g., HashTable and Vacation-Low), eager and lazy management yield almost identical results.

RBTree and Vacation-High embody similar tradeoffs in conflict management. At low contention levels, *Eager* and *Lazy* yield similar performance. Beyond 4 threads *Lazy* scales better than *Eager*. *Lazy* permits reader-writer concurrency, which pays off when the readers commit first. At 16 threads, *Lazy*'s advantage is 16% in RBTree and 27% in Vacation-High.

LFUCache admits no concurrency, since transactions conflict on the same cache line with high probability. On conflict,

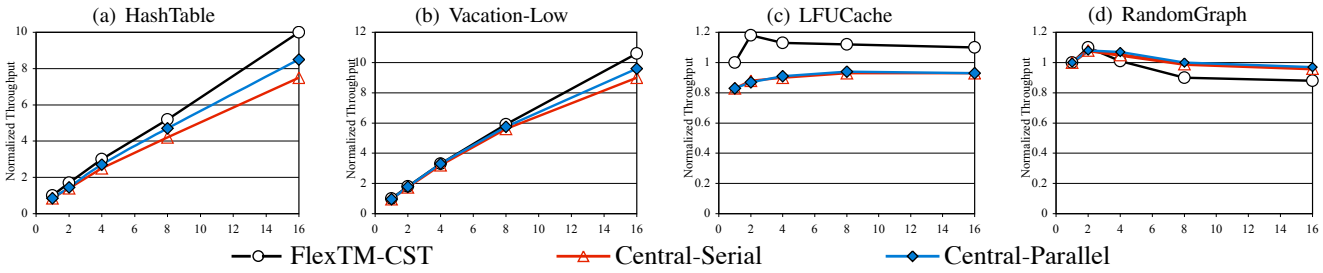
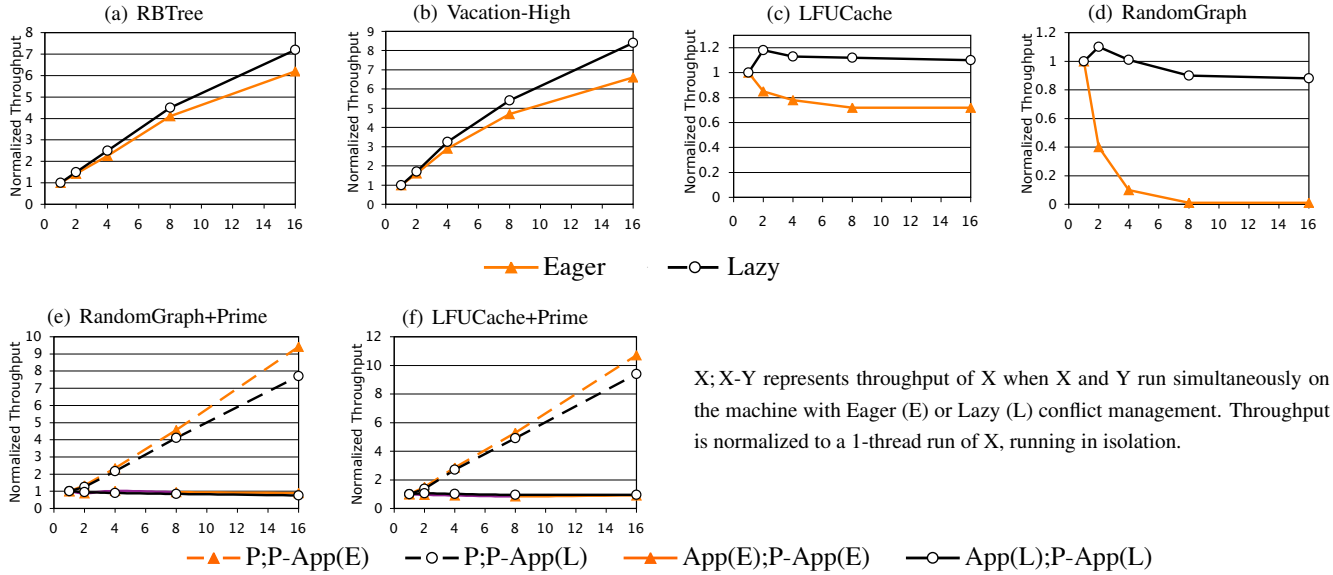


Figure 5: FlexTM vs. Centralized hardware arbiters



X; X-Y represents throughput of X when X and Y run simultaneously on the machine with Eager (E) or Lazy (L) conflict management. Throughput is normalized to a 1-thread run of X, running in isolation.

Figure 6: Eager vs. lazy conflict management in FlexTM. In plots (a)-(d), Throughput is normalized to FlexTM Eager, 1 thread. In plots (e)-(f), a prime factoring program (P) is mixed with RandomGraph and LFUCache respectively.

the conflict manager performs back-off within the active transaction. With high levels of contention, *Eager* causes a cascade of futile stalls in doomed transactions. It also reduces concurrency by creating a large window of conflict vulnerability (first-access to commit-time). In lazy mode, transactions abort enemies at commit-time, by which time the likelihood of committing is very high. Hence, with *Lazy*, throughput improves by 38% at 16 threads when compared to *Eager* for which performance degrades.

RandomGraph transactions livelock with *Eager* at higher thread levels, because it is highly likely that transactions will conflict on a highly contended object, giving rise to multi-transaction duelling aborts. With *Lazy*, once a transaction aborts an enemy at commit time, the remaining window of vulnerability is very small, the transaction is quite likely to commit, and performance remains flat with increasing thread count.

In a second set of experiments (Figure 6(e) and 6(f)), we analyze the impact of conflict management on background applications. We experimented with both transactional and non-transactional workloads; for brevity we present only the latter here: a CPU intensive application (Prime Factorization) sharing the machine with a non-scalable transactional workload (LFUCache or RandomGraph). We minimized extraneous overheads by controlling workload schedules at the user level: on transaction abort the thread yields to compute-intensive work.

We found that Prime scales better when running with eager mode transactions (~20% better than lazy in RandomGraph), because *Eager* detects doomed transactions earlier and immediately yields the CPU to useful work. *Lazy* is optimistic and takes longer to detect impending aborts. It also delays the restart of commit-worthy transactions. Significantly, yielding to the background application did not negatively impact the throughput of the transactional application, since LFUCache and RandomGraph have little concurrency anyway. By effectively serializing transactions, yielding also avoids the livelock encountered by eager RandomGraph.

8 Conclusions and Future Work

FlexTM introduces *Conflict Summary Tables*; combines them with Bloom filter signatures, alert-on-update, and programmable data isolation; and virtualizes the combination across context switches, overflow, and page-swaps. It (1) decouples conflict detection from conflict management and allows software to control detection time (i.e., eager or lazy); (2) supports

a variety of commit protocols by tracking conflicts on a thread-by-thread basis, rather than a location-by-location basis; (3) enables software to dictate policy without the overhead of separate metadata; and (4) permits TM components to be used for non-transactional purposes. To the best of our knowledge, it is the first hardware TM to admit an STM-like distributed commit protocol, allowing lazy transactions to arbitrate and commit in parallel (it also supports eager transactions).

On a variety of benchmarks, FlexTM outperformed both pure and hardware-accelerated STM systems. It imposed minimal overheads at lower thread levels (single thread latency comparable to CGL) and attained $\sim 5\times$ more throughput than RSTM and TL2 at all thread levels. Experiments with commit schemes indicate that FlexTM's distributed protocol is free from the arbitration and serialization overheads of central hardware managers. Finally, our experiments confirm that the choice between eager and lazy conflict management is workload dependent, highlighting the value of policy flexibility.

In the future, we hope to enrich our semantics with hardware support for nesting, and to study the interplay of conflict management timing and policies. We have begun to experiment with non-TM uses of our decoupled hardware [30, TR version] [31]; we expect to extend this work by developing more general interfaces and exploring their applications.

9 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Onur Mutlu, for suggestions and feedback that helped to improve this paper. Our thanks as well to Virtutech AB for their support of Simics, and to the Wisconsin Multifacet group for their support of GEMS.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. *11th Intl. Symp. on High Performance Computer Architecture*, Feb. 2005.

[2] B. H. Bloom. Space/Time Trade-Off in Hash Coding with Allowable Errors. *Comm. of the ACM*, 13(7), July 1970.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), Nov. 2006.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. *34th Intl. Symp. on Computer Architecture*, June 2007.

[5] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. *33rd Intl. Symp. on Computer Architecture*, June 2006.

[6] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. *34th Intl. Symp. on Computer Architecture*, June 2007.

[7] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. *13th Intl. Symp. on High Performance Computer Architecture*, Feb. 2007.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. *12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. *20th Intl. Symp. on Distributed Computing*, Sept. 2006.

[10] K. Fraser and T. Harris. Concurrent Programming Without Locks. *ACM Trans. on Computer Systems*, 25(2), May 2007.

[11] J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti. Design of the Power6 Microprocessor. *Intl. Solid State Circuits Conf.*, Feb. 2007.

[12] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. *31st Intl. Symp. on Computer Architecture*, June 2004.

[13] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. *22nd ACM Symp. on Principles of Distributed Computing*, July 2003.

[14] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *20th Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993.

[15] M. D. Hill, D. Hower, K. E. Moore, M. M. Swift, H. Volos, and D. A. Wood. A Case for Deconstructing Hardware Transactional Memory Systems. TR 1594, Dept. of Computer Sciences, Univ. of Wisconsin–Madison, June 2007.

[16] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. *11th ACM Symp. on Principles and Practice of Parallel Programming*, Mar. 2006.

[17] J. R. Larus and R. Rajwar. *Transactional Memory*, Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[18] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *24th Intl. Symp. on Computer Architecture*, June 1997.

[19] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. *19th Intl. Symp. on Distributed Computing*, Sept. 2005.

[20] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. *1st ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006.

[21] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *ACM SIGARCH Computer Architecture News*, Sept. 2005.

[22] C. C. Minh, M. Trautmann, J. Chung,, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. *34th Intl. Symp. on Computer Architecture*, June 2007.

[23] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. *12th Intl. Symp. on High Performance Computer Architecture*, Feb. 2006.

[24] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. *32nd Intl. Symp. on Computer Architecture*, June 2005.

[25] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. *11th ACM Symp. on Principles and Practice of Parallel Programming*, Mar. 2006.

[26] N. Sakran, M. Yuffe, M. Mehalel, J. Doweck, E. Knoll, and A. Kovacs. The Implementation of the 65nm Dual-Core 64b Merom Processor. *Intl. Solid State Circuits Conf.*, Feb. 2007.

[27] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. *40th Intl. Symp. on Microarchitecture*, Dec. 2007.

[28] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. *24th ACM Symp. on Principles of Distributed Computing*, July 2005.

[29] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. *IEEE Intl. Symp. on Workload Characterization*, Sept. 2007.

[30] A. Shriraman, M. F. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. *34th Intl. Symp. on Computer Architecture*, June 2007. TR 910, Dept. of Computer Science, Univ. of Rochester, Dec. 2006.

[31] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. TR 925, Dept. of Computer Science, Univ. of Rochester, Nov. 2007.

[32] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. *20th Intl. Symp. on Distributed Computing*, Sept. 2006.

[33] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott. Alert-on-Update: A Communication Aid for Shared Memory Multiprocessors (poster paper). *12th ACM Symp. on Principles and Practice of Parallel Programming*, Mar. 2007.

[34] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. TR 915, Dept. of Computer Science, Univ. of Rochester, Feb. 2007.

[35] Sun Microsystems Inc. OpenSPARC T2 Core Microarchitecture Specification. July 2005.

[36] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Valos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. *13th Intl. Symp. on High Performance Computer Architecture*, Feb. 2007.

[37] C. Zilles and L. Baugh. Extending Hardware Transactional Memory to Support Non-Busy Waiting and Non-Transactional Actions. *1st ACM SIGPLAN Workshop on Transactional Computing*, June 2006.