J. Parallel Distrib. Comput. 70 (2010) 1068–1084

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Implementation tradeoffs in the design of flexible transactional memory support☆

Arrvindh Shriraman*, Sandhya Dwarkadas, Michael L. Scott

Department of Computer Science, University of Rochester, United States

ARTICLE INFO

Article history:
Available online 21 March 2010

Keywords:
Synchronization
Atomicity
Transactional memory
Version management
Conflict detection
FlexTM

ABSTRACT

We present *FlexTM* (FLEXible Transactional Memory), a high performance TM framework that allows software to determine when (eagerly, lazily, or in a mixed fashion) and how to manage conflicts, while employing hardware to manage transactional state and to track conflicts. FlexTM coordinates four decoupled hardware mechanisms: read and write *signatures*, which summarize per-thread access sets; per-thread *conflict summary tables (CSTs)*, which identify the processors with which conflicts have occurred; *Programmable Data Isolation*, which buffers speculative updates in the local cache and uses an overflow table to handle unbounded updates; and *Alert-On-Update*, which notifies a thread immediately when a specified location is written by another processor. The CSTs enable an STM-inspired commit protocol that manages conflicts in a decentralized manner (no global arbitration) and allows parallel commits.

We explore the implementation tradeoffs associated with FlexTM's versioning and conflict detection mechanisms. Our results demonstrate that FlexTM exhibits ∼5× speedup over high-quality software TMs, and ∼1.8× speedup over hybrid TMs (those with software always in the loop), with no loss in policy flexibility. We find that the distributed commit protocol improves performance by 2%–14% over an aggressive centralized arbiter mechanism that also allows parallel commits. Finally, we compare the use of an aggressive hardware controller (as used in the base FlexTM design) to manage and to access any speculative transaction state overflowed from the cache, to a hardware–software approach dubbed FlexTM-S (FlexTM-Streamlined), where software manages the overflow region but uses a metadata cache to accelerate speculative data replacements and their subsequent accesses. We demonstrate that FlexTM-S's performance is within 10% of FlexTM's despite its substantially simpler virtualization mechanism.

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 in an all-or-nothing manner, 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. A *conflict detection mechanism* is needed to identify such conflicts so that the system can ensure that transactions do not perform erroneous externally visible actions as a result of an inconsistent view. The *conflict resolution time* decides *when* the detected conflicts (if they still persist) are managed. To resolve conflicts, a *conflict manager* is responsible for the policy used to arbitrate among conflicting transactions and decide which should abort. Most TM systems blend detection and resolution: pessimistic (eager) systems perform both as soon as possible; optimistic (lazy) systems delay conflict resolution until commit time (although 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. [31], 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 resolution and management, and version management can be implemented in hardware (HTM) [1,19,21,31,32], software (STM) [14,15,20,27,33], or some hybrid of the two in a hardware-accelerated TM (HaTM) [13,23,30,39]. Full hardware systems are

☆ This work was supported in part by NSF grants CCF-0702505, CCR-0204344, CNS-0411127, CNS-0615139, CNS-0834451, and CNS-0509270; NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01; an IBM Faculty Partnership Award; equipment support from Sun Microsystems Laboratories; and financial support from Intel and Microsoft.

* Corresponding author.
E-mail addresses: ashriram@cs.rochester.edu (A. Shriraman), sandhya@cs.rochester.edu (S. Dwarkadas), scott@cs.rochester.edu (M.L. Scott).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.03.006

typically inflexible in policy, with fixed choices for eagerness of conflict resolution, 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 [7,39,46] have advocated *decoupling* the hardware components of TM, giving each a well-defined API that allows them to be implemented and invoked independently. Hill et al. [22] argue that decoupling makes it easier to refine an architecture incrementally. At ISCA'07 [39], we argued 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 [22] [39, TR version].

Several papers have identified performance pathologies with certain policy choices (eagerness of conflict resolution; management policy and back-off strategy) in certain applications [5,37, 39,42]. RTM [39] 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's conflict detection mechanism enforces immediate conflict resolution, software can choose (by controlling the timing of metadata inspection and updates) when conflicts are resolved. Unfortunately, metadata management imposes noticeable performance overheads and complicates the programming interface [39].

The FlexTM approach. We propose FlexTM (FLEXible Transactional Memory) [40], a TM design that separates conflict detection from resolution and management, and leaves software in charge of the latter. Simply put, hardware always detects conflicts eagerly during the execution of a transaction and records them, but software chooses when to notice and what to do about it. Unlike proposed *eager* systems, FlexTM allows conflicting transactions to execute concurrently to uncover potential parallelism and unlike proposed *lazy* systems, FlexTM does not postpone detection to the commit stage, permitting a lightweight commit. FlexTM employs a set of decoupled (separable) hardware primitives to support version management and conflict detection without the need for sophisticated software metadata, which improves performance.

Specifically, FlexTM deploys four hardware mechanisms: (1) Bloom filter *signatures* (as in Bulk [7] and LogTM-SE [46]) 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*), augmented with a hardware controller to maintain the cache overflows in a pre-allocated region; and (4) RTM's *Alert-On-Update* mechanism to help transactions respond to changes in their status. The hardware structures are fully visible and under software controls which enables FlexTM to handle context switching and paging.

A key contribution of FlexTM is a commit protocol that arbitrates between transactions in a distributed fashion, allows parallel commits of an arbitrary number of transactions, and imposes performance penalties proportional to the number of transaction conflicts. The protocol enables lazy conflict resolution without commit tokens [19], broadcast of write sets [7,19], or ticket-based serialization [9]. To our knowledge, FlexTM is the first hardware TM in which the decision to commit or abort can be an entirely local operation, even when performed lazily by an arbitrary number of threads in parallel.

Simplifying the overflow mechanism. In FlexTM [40], a notable source of complexity is that a hardware controller is used to manage transactional state evicted from the cache. Hardware is expected to maintain the overflowed cache blocks in an overflow table and has to perform all the management tasks (i.e., insert, remove, lookup, commit).

We also investigate a hardware–software approach to handling overflowed state. Specifically, we develop light-weight fine-grain address remapping support. On an overflow, software sets up the buffer space and provides hardware fast access to the metadata (which provides the address of the buffer space allocated for the original location) via a metadata cache (SM-cache). Hardware uses this information to write back and access the block from the buffer space region. The extra hardware needed for overflow management is limited to the SM-cache, installed as a lookaside on the L1 miss path. We evaluate this simplified overflow mechanism in an alternative TM system, *FlexTM-S* (FlexTM-Streamlined).

Simplifying conflict resolution. In order to support *Lazy* conflict resolution, the conflict detection and versioning mechanisms must handle potentially multiple (as many as there are sharers) copies of transactional data. In conjunction with FlexTM-S, we explore *mixed* conflict resolution, which resolves write–write conflicts eagerly while allowing lazy read–write conflict resolution [38,41]. Mixed resolution enables FlexTM-S to employ a simpler versioning mechanism (only one speculative data version like *Eager* detection) and to precisely identify writer conflicts (only one writer like *Eager* detection).

We have implemented FlexTM on a 16-core (1 thread/core) CMP prototype on the Simics/GEMS simulation framework. We interface the private L1s with the shared L2 using a directory-based protocol. We investigate performance using the STAMP [30] and STMBench7 [18] workload suite. Our results suggest that FlexTM's performance is comparable to that of fixed-policy HTMs, and 1.8× and 5× better than that of hybrid TMs and plain STMs, respectively. We demonstrate that the CST-based commit process in FlexTM can avoid significant latency and serialization penalties associated with globally-arbitrated commit mechanisms in other lazy systems. Finally, comparing with various virtualization techniques, we demonstrate that the more complexity-effective FlexTM-S suffers modest performance loss compared to FlexTM (<10%) and performs much better (≃2×) compared to other previously proposed virtualization designs [30,11].

2. Related work

Larus and Rajwar [24] provide an excellent summary of transactional memory research up to Fall 2006. We first categorize the design space for both versioning and conflict detection mechanisms, indicating the relation of FlexTM to other proposed protocols.

Most HTM designs have similar approaches to small transactions, exploiting coherence for conflict detection and (possibly) the cache hardware for versioning. However, overflowed state introduces virtualization challenges, and proposed implementations vary significantly. HTMs need to address two requirements to support overflowed state: (1) a conflict detection mechanism to track concurrent accesses and conflicts for locations evicted out of the caches and coherence framework and (2) a versioning mechanism to maintain new and old values of data.

Conflict detection. TM systems need a mechanism to track the locations read and written by a transaction and to ensure that there is no overlap of a transaction's accesses with the write set of a concurrent transaction. The implementation choices can be broadly classified as:

- Software Instrumentation: The TM runtime system can be implemented as a set of software barriers on load and store instructions. These barriers gather information about the accesses, maintain the data structures (e.g., transaction read and write sets), and query the data structures to perform conflict detection. They verify that an accessed location has not changed, and ensure that an aborted transaction does not

perform any externally visible actions prior to discovering its status. In addition to imposing the instrumentation overhead that limits gains from concurrency, the barriers add to cache pressure since they access additional metadata on each memory access.

- Hardware Acceleration: Hardware support can remove the bottlenecks associated with software instrumentation by using the cache to track accesses and piggybacking on coherence to detect conflicts. There are tradeoffs associated with different hardware options: Bloom-filter based signatures [7,46] are simple to design but are prone to false-positive based performance problems. Per-location metadata [6] is precise but requires support at all levels in the memory hierarchy and modifications to the coherence protocol. Cache tag bits either require a software algorithm [13] or a complex hardware controller [1] to handle evictions.
- Virtual Memory: OS page tables include protection bits to implement process isolation. TM systems can exploit these protection bits to set up read-only and read/write permissions at a page granularity and trap concurrent accesses to detect conflicts [10,11]. The major performance overheads involved are the TLB shootdowns and OS intervention required.

Versioning. The conflict resolution policy governs the choice of versioning mechanism. Lazy allows concurrent transactions to read or write a shared location, thus necessitating a redo log in order to avoid irreversible actions, while *Eager* detects conflicts prior to the access, thereby accommodating both forms of logging. The undo log is used to restore values if a transaction aborts, while the redo log is used to copy-update the original locations on commit; these actions need to occur in an atomic manner for all the locations in the log. Most importantly, since a redo log buffers new values, it needs to intervene on all other accesses to ensure that a thread reads its own writes; this dictates the data structure used to maintain the new values (typically a hash table). An undo log approach can make do with a simpler data structure (e.g., a dynamically resizable array or vector) and typically does not need to optimize the access cost since it is traversed only on an abort.

Like conflict detection, versioning can be implemented either with software handlers, hardware acceleration, or virtual memory (i.e., translation information in the page tables) with performance and complexity tradeoffs similar to conflict detection. The software approach adds barriers to all writes (to set up the log data structures) and possibly to all reads (to return values buffered in a redo log) and leads to significant degradation in performance. The hardware approach adds significant complexity, including new state machines that interact in a non-trivial manner with the existing memory hierarchy. The virtual memory approach reuses existing hardware and OS support, but suffers the performance overheads of having to perform page granularity cloning and buffering. An important difference between the mechanisms that implement versioning and conflict detection is that versioning deals with data values (no false positives or negatives) and cannot trade precision for complexity-effectiveness as conflict detection can (e.g., using signatures).

Table 1 specifies the mechanism used by various extant TM systems. UTM [1] and VTM [32] both implement overflow support for redo logs. On a cache miss 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 if an overflow signature indicates that the block has been speculatively modified. VTM and UTM support only eager resolution of conflicts. XTM [11] and PTM [10] use virtual memory, and accept the costs of coarse granularity and OS overhead.

Table 1
Virtualization in TM.

System	Conflict resolution	Conflict detection	Versioning
UTM [1]	Eager	H (controller)	H (undo log)
VTM [32]	Eager	H (microcode)	S (redo log)
XTM [11]	Eager/Lazy	VM	VM (redo log)
PTM-Select [10]	Eager	H (controller)	VM (undo log)
LogTM-SE [46]	Eager	H (Signature)	H (undo log)
TokenTM [6]	Eager	H (ECC)	H (undo log)
Hybrid systems			
SigTM [30]	Eager/Lazy	H (signature)	S (redo log)
UFO_TM [2]	Eager	H (ECC)	S (undo log)
HyTM [13]	Eager	S	S (undo log)
RTM [39]	Eager/Lazy	S	S (redo log)
FlexTM [40]	Eager/Lazy	H (signature)	H/S (redo log)

H — Hardware Acceleration; S — Software Instrumentation; VM — Virtual Memory. Hybrids (other than SigTM) use a best-effort HTM for small transactions.

LogTM-SE [46] integrates the undo log mechanism of LogTM [31] with Bulk-style signatures [7]. It supports efficient virtualization (i.e., context switches and paging), but this is closely tied to eager versioning (undo logs), which cannot support *Lazy* systems. Since LogTM-SE does not allow transactions to abort one another, it is possible for running transactions to "convoy" behind a suspended transaction. Like LogTM-SE, TokenTM [6] uses undo logs to implement versioning, but implements conflict detection using a hardware token scheme.

Hybrid TMs [13,23] allow hardware to handle common-case bounded transactions and fall back to software for transactions that overflow time and space resources. To allow hardware and software transactions to co-exist, hybrid TMs must maintain metadata compatible with the fallback STM and use policies compatible with the underlying HTM. SigTM [30] employs hardware signatures for conflict detection but uses a (always on) TL2 [14] 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.

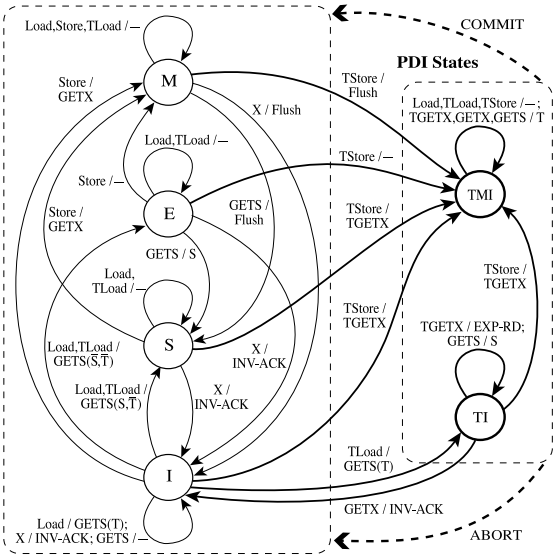
RTM [39] explored hardware acceleration of STM. Specifically, it introduced (1) Alert-On-Update (AOU), which triggers a software handler when pre-specified lines are modified remotely, and (2) Programmable Data Isolation (PDI), which buffers speculative writes in (temporarily incoherent) local caches. 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 and then discuss how they work together. Fig. 2 shows all the hardware components we add to the system.

3.1. Access tracking: signatures

Like Bulk [7], LogTM-SE [46], and SigTM [30], FlexTM uses Bloom filter *signatures* [3] 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 UTM [1] and VTM [32] in the case of overflow. Every FlexTM processor maintains a *read signature* (R_{sig}) and a write signature (W_{sig}) for the current



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*

Fig. 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 resulting 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".

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* [1,31] and *Lazy* [30,7,19] systems track conflicts 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. The *W–R* and *W–W* lists at a processor P represent the transactions that might need to be aborted when the transaction at P wants to commit. The *R–W* list helps disambiguate abort triggers; if an abort is initiated by a processor not marked in the CST, the transaction can safely avoid the abort. 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. While CSTs can be read and written independently, they do require interfacing with a mechanism to detect conflicts when they occur. In FlexTM, we use signatures to detect conflicts, but the CSTs could be adapted to interface with any of the hardware metadata schemes discussed in Section 2.

3.3. Versioning support: PDI

RTM [39] proposed *programmable data isolation (PDI)* that allowed software to exploit incoherence (when desired). It proposed a Transactional-MESI (TMESI) snooping protocol that supported multiple speculative writers and exploited the inherent buffering capabilities of private caches to isolate the potentially multiple speculative copies. 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 (non-speculative versions 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 TMESI 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. The TMESI protocol is derived from the SGI ORIGIN 2000 [25] with support for silent evictions. Directory information is maintained at the L2. Details appear in Fig. 1.

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 — Fig. 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 speculative writers. We do this by extending the existing support for multiple sharers and use the modified bit to distinguish between the possibility of 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.

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. *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

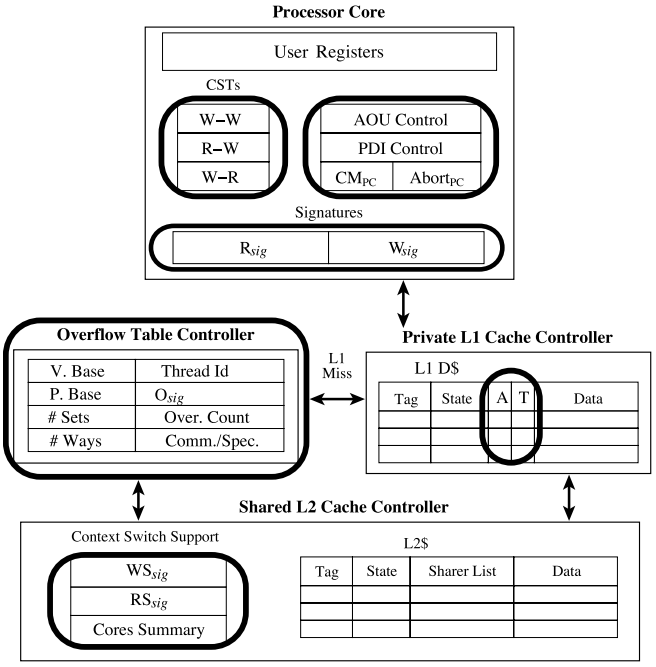


Fig. 2. FlexTM architecture overview (dark lines surround FlexTM-specific state).

sharer to the directory. (Note that a *TLoad* from *E* or *S* can never be threatened; the remote transition to *TMI* would have moved the line to *I*. Unlike RTM, which keeps track of transactional read sets within the cache state, FlexTM is able to eliminate RTM's extra states by using separate read signatures.)

FlexTM enforces the single-writer or multiple-reader invariant for non-transactional lines. For transactional lines, it ensures that (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 [7,19] 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 takes place and without the need for bulk coherence messages.

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 the cache. Software can clean up transactional state by issuing an *Abort* to the controller, which then 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 its read and write signatures and appends an appropriate message type to its response, as shown in the table in Fig. 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* indicates 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.

3.4. Explicit aborts: AOU

The *Alert-On-Update (AOU)* mechanism, borrowed from RTM [39], 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) [48]. 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 Fig. 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 requires AOU support for only one cache line (the transaction status word; see Section 4.1) and can therefore use the simplified hardware mechanism (avoiding the bit per cache tag) as proposed in [43]. More general AOU support might still be useful for non-transactional purposes.

3.5. Extending FlexTM

Support for Multi-threading. Multi-threaded cores pose two main challenges to FlexTM: each thread's transactional state needs to be disambiguated from other threads on the same core, and conflicts need to be detected among these threads.

To disambiguate each thread's transaction state, per-thread signatures, CSTs, AOU, and PDI control registers must be included. Similarly, the alert bit per cache line must be replicated. Speculatively written state is more challenging—a cache block can buffer only a single thread's speculative write and a conventional L1 cache is allowed to buffer one copy of a cache block. To permit each thread to cache speculatively modified data, we must include a thread id along with the "T"-bit, and use it to indicate which thread speculatively wrote the specific block. We could then allow each L1 set to buffer multiple versions of the same cache block, or alternatively, we could buffer only a single version of the cache block in the L1 for one of the hardware threads and use the overflow mechanisms discussed in Section 5 to maintain the other versions in each thread's private overflow region. Overflow handling must also be modified to allow per thread state.

Conflict detection between the hardware threads is challenging to handle because there are no coherence accesses within a core. Fortunately, each hardware thread's signature is maintained in the same core and we can query the signatures of other threads to detect conflicts.

Snooping protocols. Accommodating broadcast snooping protocols within FlexTM is straightforward. Consider a protocol in which the L1 cache broadcasts its requests to other L1s and to the shared L2 cache on an ordered network. The additional state required by FlexTM remains the same. The only change required is to the L1 cache response mechanism. Typically snooping protocols implement response messages using a wired-OR signal that cannot identify the sharing processors. We would need to include extra signal lines to encode the actual identities.

Nesting. While FlexTM allows operations to escape transactional semantics with normal loads and stores, it handles nested

Table 2
Alert-on-update software interface.

Registers	
%aou_handlerPC:	Address of the handler to be called on a user-space alert
%aou_oldPC:	PC immediately prior to the call to %aou_handlerPC
%aou_alertAddress:	Address of the line whose status change caused the alert
%aou_alertType:	Remote_write, lost_alert, or capacity/conflict eviction
Instructions	
set_handler %r	Move %r into %aou_handlerPC
clear_handler	Clear %aou_handlerPC and flash-clear the alert bits for all cache lines
aload [%r1], %r2	Load the word at address %r1 into register %r2, and set the alert bit(s) for the corresponding cache line
arelease %r	Unset the alert bit for the cache line that corresponds to the address in register %r
arelease_all	Flash-clear alert bits on all cache lines

Table 3
Programmable-data-isolation software interface.

Registers	
%t_in_flight:	A bit to indicate that a transaction is currently executing
Instructions	
begin_t	Set the %t_in_flight register to indicate the start of a transaction
tstore [%r1], %r2	Write the value in register %r2 to the word at address %r1; isolate the line (*TMI* state)
tload [%r1], %r2	Read the word at address %r1, place the value in register %r2, and tag the line as transactional
abort	Discard all isolated (*TMI* or *TI*) lines; clear all transactional tags and reset the %t_in_flight register
cas-commit [%r1], %r2, %r3	Compare %r2 to the word at address %r1; if they match, commit all isolated writes (*TMI* lines) and store %r3 to the word; otherwise discard all isolated writes; in either case, clear all transactional tags, discard all isolated reads (*TI* lines), and reset the %t_in_flight register

transactions via the subsumption model. The key challenge to supporting true nesting is disambiguating between the hardware state (e.g., speculative lines in the cache) of each nested level. We could include limited support for nesting using techniques such as split hardware transactions [26] but their performance overheads need further investigation.

4. Hardware/software interface

In this section, we discuss the interface provided by each FlexTM component, provide the pseudo-code for the main TM runtime macros, and discuss their usage.

Tables 2 and 3 list the instructions and registers required by the AOU and PDI mechanisms. AOU's interface includes special registers to hold the address of the user-mode handler and a description of the current alert; and instructions to set and unset the user-mode handler and to mark and unmark cache lines (i.e., to set and clear their alert bits). PDI's interface includes support for speculative reads (*TLoad*s) and writes (*TStore*s), which are interpreted as speculative when the hardware transaction bit (%hardware_t) is set. *CAS-Commit* enables the software runtime to couple the logical commit of the transaction in software with the committing of the speculative hardware state. The *CAS-Commit* instruction performs the usual function of compare-and-swap. In addition, if the CAS succeeds, *TMI* lines revert to *M*, making their data visible to other readers through normal coherence actions. If the CAS fails, the buffered state remains in the local cache for software to handle appropriately. Buffered state can be eliminated and coherence restored by issuing an *Abort*.

CSTs and signatures are treated as registers that can be loaded from and stored to memory. Software can set them up to operate in two modes (*Eager* or *Lazy*) for each types of conflict (*W–W* or *W–R* or *R–W*). In eager mode, a conflicting coherence message updates the CST and triggers the handler; in *Lazy* mode, a conflict updates the CST but does not trigger the handler — software reads the CSTs when it desires.

4.1. Bounded transactions

In this section, we discuss the execution of transactions with bounded access sets that fit within an OS quantum. We assume a subsumption model for nesting, with support for transactional pause [47], which suspends a transaction in order to perform non-transactional activity.

Table 4
Transaction Descriptor contents. All fields except TSW and State are cached in hardware registers for transactions running.

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 resolution

A FlexTM transaction is represented by a software *descriptor* (Table 4). This descriptor includes a status word, space for buffering the hardware state when paused (CSTs, Signatures, and Overflow control registers), pointers to the abort (Abort_{PC}) and contention management handlers (CM_{PC}), and a field to specify the conflict resolution mode of the transaction.

A transaction is delimited by BEGIN_TRANSACTION and END_TRANSACTION macros (see Fig. 3). BEGIN_TRANSACTION establishes the 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). Some of these operations are not intrinsically required and can be set up for the entire lifetime of a thread (e.g., Abort_{PC} and CM_{PC}). END_TRANSACTION aborts conflicting transactions and tries to atomically update the status word from active to committed using *CAS-Commit*.

Within a transaction, the processor issues *TLoad*s and *TStore*s when it expects transactional semantics, and conventional loads and stores when it wishes to bypass those semantics. *TLoad*s and *TStore*s are interpreted as speculative when the hardware transaction bit (%hardware_t) is set. This convention facilitates code sharing between transactional and non-transactional program fragments. Ordinary loads and stores can be requested within a transaction; these could be used to implement escape actions, update software metadata, or reduce the cost of thread-private updates in transactions that overflow cache resources. In order to

```
BEGIN_TRANSACTION()
1.  clear R_sig and W_sig
2.  set Abort_PC
3.  set CM_PC
4.  TSW[my_id] = active
5.  Aload(TSW[my_id])
6.  begin_t
```

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

Fig. 3. Pseudocode of BEGIN_TRANSACTION and END_TRANSACTION (for *Lazy* transactions).

avoid the need for compiler generation of the *TLoad*s and *TStore*s, 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.

Transactions of a given application can employ either *Eager* or *Lazy* conflict resolution. 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 CASing 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 local 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 *END_TRANSACTION()* routine shown in Fig. 3.

All of the work for the *END_TRANSACTION()* routine occurs in software, with no need for global arbitration [7,9,19], blocking of other transactions [19], 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 3–6 of Fig. 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 *END_TRANSACTION()* routine is restarted. In the case of a *R–W* conflict, no action is needed since *T* is the reader and is about to serialize before the writer (i.e., the two transactions can commit concurrently). Software mechanisms can be used to disambiguate conflicts and avoid spurious aborts when the writer commits.

The contention management policy (line 4) in the commit process is responsible for providing various progress and performance guarantees. The TM system can choose to plug in an application-specific policy. For example, if we used a Timestamp manager [36], then it will ensure livelock freedom. More recently, EazyHTM [45] has exploited CST-like bitmaps to accelerate a pure-HTM's commit, but does not allow pluggable policies. FlexTM's commit operation is entirely in software and its latency is proportional to the number of conflicting transactions — in the absence of conflicts there is no overhead. Even in the presence of conflicts, aborting each conflicting transaction consumes only the latency of a single CAS operation (at most a coherence operation).

4.2. Mixed conflict resolution

While *Lazy* conflict resolution generally provides the best performance with its ability to exploit concurrency and ensure progress [41], it does introduce certain challenges. In particular, it requires a multiple-writer and/or multiple reader protocol that makes notable additions to a basic MESI protocol. Multiple L1's need to be able to concurrently cache a block and read and write it (quite different from the basic "S" and "M" states). This is a source of additional complexity over an *Eager* system and could prove to be a barrier to adoption.

Furthermore, write–write conflicts need to be conservatively treated as dueling read–write and write-read conflicts since conflicts are detected using coherence actions and a transaction that obtains permissions to write a block can also read it (the read will not result in a coherence action). It is therefore not possible to allow both transactions to concurrently commit (one of them has to abort). While commit-time conflict resolution in *Lazy* mode does try to ensure forward progress by ensuring that the winning transaction is one that is already ready to commit, for some workloads, it could also lead to significant levels of wasted work due to delayed aborts (see the results for STMBench7 in our work from ICS'09 [41]).

To avoid the wasted work and to simplify the design, we extend FlexTM to support the *Mixed*-mode conflict resolution [38,41]. In *Mixed* mode, when write–write conflicts appear (a *TStore* operation receives a *threatened* response), the processor effects a call to the contention manager. On read–write or write-read conflicts, the hardware records the conflict in the CSTs and allows the transaction to proceed. When the transaction reaches its commit point, it needs to take care of only *W–R* conflicts (using an algorithm similar to Fig. 3), as its *W–W* CST will be empty. Mixed mode tries to save wasted work on write–write conflicts and to exploit the parallelism present in *W–R* and *R–W* conflicts. However, it is also possible that since *Mixed* resolves write–write conflicts eagerly, the transaction that wins the conflict and progresses will subsequently abort, thereby wasting work.

Mixed mode has more modest versioning requirements compared to *Lazy* mode. A system that supports only *Mixed* mode and *Eager* mode can simplify the coherence protocol and overflow mechanisms. Briefly, *Mixed* maintains the single writer and/or multiple reader invariant: it allows only one writer for a cache block (unlike *Lazy* mode) although the writer can co-exist with concurrent readers (unlike *Eager* mode). At any given instant, there is only one speculative copy accessed by the single writer and/or a non-speculative version accessed by the concurrent readers. This simplifies the design of the *TMI* state in the TMESI protocol. Only one of the L1 caches in the system can have the line in *TMI* (not unlike the "M" state in MESI).

The more stream-lined version of FlexTM (FlexTM-S) that we evaluate supports only *Mixed* and *Eager* modes. In Section 5.3:

FlexTM-S we demonstrate another advantage of supporting only *Mixed* and *Eager* modes: since they restrict the number of speculative writers to at most one, writer conflicts may be precisely identified.

4.3. Strong isolation

As implied in Fig. 1, transactional and ordinary loads and stores to the same location can occur concurrently. While we are disinclined to require strong isolation [4] as part of the user programming model (it's hard to implement on legacy hardware, and is of questionable value to the programmer [12]), 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 non-transactional write miss hits in the responder's R_{sig} or W_{sig}, it aborts the responder's transaction, so the non-transactional write appears to serialize before the (retried) transaction. A non-transactional 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).

5. 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 provide mechanisms to handle transactional state evicted from the L1 cache. Cache evictions must be handled carefully. 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. We first describe our approach to handling coherence-based conflict detection for evicted lines, followed by two alternative schemes for versioning of evicted *TMI* lines.

5.1. Eviction of transactionally read lines

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 if there is a hit in the local signature. 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 [31].

5.2. Overflow table (OT) controller design

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 Fig. 2. They include a thread identifier, a signature (O_{sig}) for the overflowed cache lines, a count of the number 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 OT controller 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 support this by ensuring that 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} summarizes the entries in the OT and provides quick lookaside for L1 misses. All L1 misses check the O_{sig} and hits trigger an OT lookup in parallel with the L2 access. If the block is found in the OT, the hardware fetches it and overrides the L2 response. O_{sig} is also needed to ensure atomic write-back of data buffered in the OT. When a transaction commits, it exposes the O_{sig} to forwarded coherence requests. Coherence requests that hit in the O_{sig} indicate that buffered data in the OT are possibly being copied back to the original location. The response to the request can be dealt with in two ways: the controller could either perform lookup in the OT and respond with the data block or it could NACK the request until copyback completes; our current implementation does the latter.

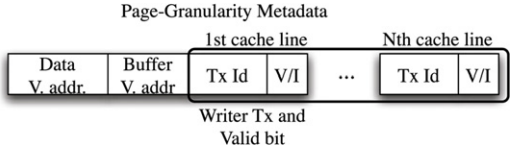
In addition to functions previously described, the *CAS-Commit* operation sets the Committed bit in the controller's OT state. This indicates that the OT content should be visible, activating NACKs or lookups. At the same time, the controller initiates a microcoded copyback operation. To accommodate page evictions of the original locations, OT tags include the virtual addresses of cache blocks. These addresses are used during copyback to ensure automatic page-in of any nonresident pages.

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 logs [31], 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 reclaimed, to be cleaned up for use by another transaction. 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.

Although we require that OTs be physically contiguous for simplicity, they can themselves be paged. A more ambitious FlexTM design could allow physically non-contiguous OTs, with controller access mediated by more complex mapping information. 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.

5.3. Software metadata cache (SM-Cache) approach

The OT controller mechanism just described consists of a hardware state machine that maintains a write buffer (organized as a



Data V. addr is the virtual page number of the original location. Buffer V. addr is the virtual page number of the buffer page. The (Tx_id, V/I) pair denote the writer transaction's id and the validity of the buffered cache block. An array of (Tx_Id, V/I) pairs is associated with the page. The (Tx_id, V/I) denote the following semantics when accessed by transaction T; (don't care,0): buffer-page cache block empty. (X,1) X≠T: T conflicts with writer transaction X; (T,1): T has speculatively written the block and evicted it.

Fig. 4. Metadata for pages that have overflowed state.

hash table) in a software-allocated region. There is implementation complexity associated with the state machine that searches (writes back and reloads) and accesses data blocks without any help from software.

In this section, we propose a more streamlined mechanism used in the *FlexTM-S* design. We move the actions of maintaining the data structure and performing the redo on commit to software, replacing the hash table with buffer pages and introducing a metadata cache that enables hardware to access the buffer pages without software intervention. Fig. 4 shows the per-page software metadata, which specifies the buffer-page address and for each cache block, the writer transaction id (Tx_id) and a "V/I" bit to indicate if the buffer block is buffering valid data. To convey the metadata information to hardware and accelerate repeated block accesses, we install a metadata cache (SM-cache) on the L1 miss path (see Fig. 5).

When a speculatively written cache line is evicted, the cache controller looks up the SM-cache for the metadata and uses the buffer page address to index into the TLB (for the buffer page's physical address[1]) for writeback redirection. Multiple transactions that are possibly writing different cache blocks on the same page can share the same buffer page. A miss in the SM-cache triggers a software handler that allocates the buffer page metadata and reloads the SM-cache. To provide the commit handler with the virtual address of the cache block to be written back, every SM-cache entry includes this information and is virtually indexed (note that the data cache is still physically indexed). While the entire buffer page is allocated when a single cache block in the original page is evicted, the individual buffer page cache blocks are used only as and when further evictions occur. This ensures that the overflow mechanism adds overhead proportional to the number of cache blocks that are evicted (similar to the OT controller mechanism). In contrast to this design, other page-based overflow mechanisms (e.g., XTM [11] and PTM [10]) clone the entire page if at least a single cache block on the page is evicted.

With data buffered, L1 misses now need to ensure that data is obtained from the appropriate location (buffer page or original). As in the *OT* controller design, we use an overflow signature (O_{sig}) to summarize addresses of evicted blocks and elide metadata checks. L1 misses check the O_{sig}, and signature hits require a metadata check. If the metadata indicates that transaction T accessing the location had written the block (i.e., V/I bit is 1 and Tx_id = T), then hardware fetches the buffer block and overrides the L2 response. It also unsets the V/I bit to indicate that the buffer block is no longer valid (block is present in the cache). Otherwise, the coherence response message dictates the action. On eviction of a speculatively written cache line that another transaction has written and overflowed as well (i.e., V/I bit is 1 and Tx_id = X, X ≠ T), a handler is invoked that either allocates a new buffer page and refills the SM-cache or resolves the conflict immediately. The former design supports multiple writers to the same location (and enables *Lazy* conflict resolution), while the latter forces eager write–write conflict resolution, but enables a simpler design. The Tx_id field supports precise detection of writer conflicts (see the FlexTM-S design below).

When a transaction commits, it copy-updates the original locations using software routines. To ensure atomicity, the transaction updates its status word to inform concurrent accesses to hold off until the copy-back completes. It then iterates through the metadata of the various buffer pages in the working set and copies back the cache blocks that it has written.

SM-Cache. The SM-cache stores metadata that hardware can use to accelerate block access and cache evictions without software intervention. It resides on the L1 miss path. On an O_{sig} hit the SM-cache is looked up in parallel with the L2 lookup (see Fig. 5). SM-cache misses are handled entirely by software handlers that index into it using the virtual page address. The L1 controller also uses a similar technique to obtain metadata for redirecting evictions and reloads.

The metadata may be concurrently updated if different speculative cache blocks in the page are evicted at multiple processor sites. To ensure metadata consistency, the SM-cache participates in coherence using the physical address of the metadata. This physical address tag is inaccessible to software and is automatically filled by the hardware when an entry is allocated. The dual-tagging of the SM-cache introduces the possibility that the two tags (virtual address of page and physical address of metadata) might not map to the same set index. We solve this with tag array pointers [17].

FlexTM-S. To evaluate the performance of the SM-cache approach, we developed FlexTM-S. For bounded transactions, it leverages the hardware presented in Section 3, but it omits support for *Lazy* conflict resolution.

Compared to FlexTM, FlexTM-S (1) simplifies hardware support for the versioning mechanism by trading in FlexTM's overflow hardware controller for an SM-cache (software metadata cache) and (2) allows precise detection of conflicting writers. By restricting support to *Mixed* and *Eager* modes, i.e., allowing only one speculative writer, the coherence protocol is also simplified.

To ensure low overhead for detecting conflicting readers, FlexTM-S uses the R_{sig} for both overflowed and cached state. To identify writer transactions, it uses a two-level scheme: if the speculative state resides in the cache, the response message from the conflicting processor identifies the transaction (the CST bits will identify the conflicter's id). If the speculative state has been evicted then the O_{sig} membership tests will indicate the possibility of a conflict. This type of conflict is also encoded in the response message. If an O_{sig} conflict is indicated, the requester checks the metadata for precise disambiguation, thereby eliminating false positives. Since a block can be written by only one transaction (*Mixed*|*Eager* invariant), the Tx_id in the metadata precisely identifies the writer. If the metadata indicates no conflict, software loads the SM-cache instructing hardware to ignore the O_{sig} response and allows the transaction to proceed. Thus the metadata for versioning helps to disambiguate writer transactions, which (1) helps identify the conflicting writer precisely and (2) allows progress of non-conflicting transactions, which would have otherwise required contention management (in *Eager* mode) due to signature false-positives.

5.4. Handling OS page evictions

The two challenges left to consider are (1) eviction of a page from physical memory and reuse of its frame for a different page

[1] Virtual page synonyms are cases where multiple virtual pages point to the same physical frame and a thread can access the same location with different virtual addresses. To resolve these, since software knows about the pages that are synonyms, it ensures that the SM-cache is loaded with the same metadata for all the virtual synonym pages.

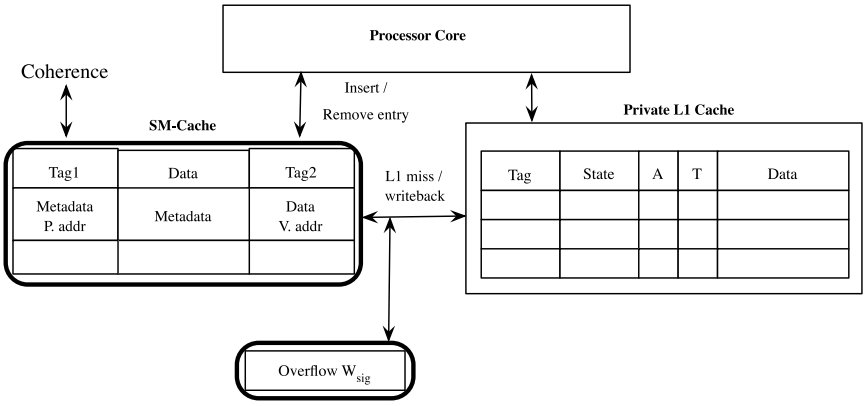


Fig. 5. Simplified overflow support with SM-cache. Dashed lines surround the new extension that replaces the OT controller (see Fig. 2).

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 address these challenges with virtual address-based conflict detection for non-resident pages.

For (2) we adapt a solution first proposed in LogTM-SE [46]. 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. Fortunately, since a typical page (4/8 KB) contains only about 64–128 cache lines, this does not impose significant overhead compared to the cost of page eviction. Finally, there are differences in the support required from the paging mechanism for the OT controller approach and the SM-Cache approach. The former indexes into the overflow table using the physical address and requires the paging mechanism to update the tags in the table entries with the new physical address. The latter needs no additional support since it uses the virtual address of the buffer page, and at the time of writeback indexes into the TLB to obtain the current physical address.

6. 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 accesses overlap with the read and write set of suspended transactions should bear the software routine overhead. Both FlexTM and FlexTM-S handle context switches in a similar manner. To remember 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.[2]

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 overflow hardware registers, (3) the current R_{sig} and W_{sig}, and (4) the CSTs. After saving this state (in the order listed), 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 overflow controller registers. This ensures that any subsequent conflicting access will miss in the L1 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. A *TStore* to a line in *M* state generates a writeback (see Fig. 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* state and a new transaction on the same processor *TStore*s it.

On summary signature hits, a software handler mimics hardware operations on a per-thread basis, testing signature membership and updating the CSTs of suspended transactions. When using the SM-cache design, the software metadata from versioning can be used to precisely identify the writer conflict. 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.

[2] FlexTM updates RS_{sig} and WS_{sig} using a *Sig* message that uses the L1 coherence request network to write the uncached memory-mapped registers. 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.

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 Fig. 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 switched 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 switched 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 that correspond to the currently switched 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 [46], FlexTM is able to place the summary signature at the directory rather than on the path of every L1 access. This avoids the need for interprocessor 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 rescheduling 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.

7. Area analysis

In this section, we briefly summarize the area overheads of FlexTM. Further details can be found in a technical report [40]. Area estimates appear in Table 5. We consider processors from a uniform (65 nm) technology generation to better understand microarchitectural tradeoffs. Processor component sizes were 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. [35]. 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. 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, we compare the OT controller to the metadata cache (SM-cache) approach. While the SM-cache is significantly more area hungry than the controller, it is a regular memory structure rather than a state machine. The SM-cache needs a separate hardware cache to store the metadata while the OT controller's metadata (i.e., hash-table index entries) contend with regular data for L2 cache space. Overall, the OT controller adds less than 0.5% to core area. Its state machine is similar to Niagara-2's Translation-Storage-Buffer walker [48]. Niagara-2, with its 16-byte data cache line, presents a worst-case design point for the SM-cache. The small cache line leads to high overhead in page-level metadata, since there are more cache blocks per page (4× more than Merom or Power6) and per-cache line metadata, since the per-cache line entry (17 bits) is a significant fraction of cache line size (16 bytes). Straightforward optimizations that would save area include organizing the metadata to represent a larger than cache line region.

Table 5
Area estimation.

Processor	Merom [34]	Power6 [16]	Niagara-2 [48]
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)	0.033	0.066	0.26
$RS_{sig}+WS_{sig}$ (mm^2)	0.033	0.033	0.033
CSTs (registers)	3	6	24
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%
OT controller (mm^2)	0.16	0.24	0.035
32 entry SM-Cache (mm^2)	0.27	0.27	0.96

ID — SMT context of 'TMI' line.

Overall, with either FlexTM (which includes the OT controller) or FlexTM-S (which includes the SM-cache) the overheads imposed on out-of-order CMP cores (Merom and Power6) are well under 1%–2%. In the case of Niagara-2 (high core multithreading and small cache lines), FlexTM add-ons require a ∼2.6% area increase while FlexTM-S's add-ons require a ∼10% area increase.

8. FlexTM evaluation

8.1. 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 6(a)), on the GEMS/Simics infrastructure [29]. 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 [25] 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.

We evaluate FlexTM using the seven benchmarks listed in Table 6(b). Workload set 1 is a set of microbenchmarks obtained from the RSTM package [49] and Workload set 2 consists of applications from STAMP [30][3] and STMBench7 [18]. Kmeans and Labyrinth spend 60%–65% of their time in transactions; all other applications spend over 98% of time in transactions. In the microbenchmark 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. For the STAMP workloads and STMBench7 we use the input setup described in Table 7. In Bayes and Labyrinth we added padding to a few data structures to eliminate frequent false conflicts.

[3] We left out SSCA since it did not exercise the TM components. It has small transactions and small working sets and is highly data parallel.

Table 6
Experimental setup.

(a) Target system parameters	
16-way CMP, Private L1, Shared L2	
Processor Cores	16 1.2 GHz in-order, single issue; non-memory IPC = 1
L1 Cache	32 KB 2-way split, 64-byte blocks, 1 cycle, 32 entry victim buffer, 2 kbit signature [7, S14]
L2 Cache	8 MB, 8-way, 4 banks, 64-byte blocks, 20 cycle
Memory	2 GB, 250 cycle latency
Interconnect	4-ary tree, 1 cycle, 64-byte links
Central Arbiter (Section 8.3)	
Arbiter Lat.	30 cycles [8]
Commit Msg. Lat.	16 cycles/link
Commit messages also use the 4-ary tree.	

(b) Workload description

Benchmark	Inst/tx	Wr_{set}	Rd_{set}	CST conflicts per-tx	Avg. per-tx *W–W*	Avg. per-tx *R–W*
Workload set 1						
HashTable	110	2	5	0	0	0
RBTree	1000	3	25	1	0	1.8
LFUCache	125	1	2	6	0.8	0.8
RandomGraph	11K	9	60	5	0.6	3
Workload set 2						
Bayes	70K	150	225	3	0	1.7
Delaunay	12K	20	83	1	0.10	1.1
Genome	1.8K	9	49	0	0	0
Intruder	410	14	41	2	0	1.4
Kmeans	130	4	19	0	0	0
Labyrinth	180K	190	160	3	0	2
Vacation	5.5K	12	89	1	0	1.6
STMBench7	105K	110	490	1	0.1	1.1

Setup: 16 threads with lazy conflict resolution; *Inst/Tx*-Instructions per transaction. K—Kilo.
Wr_{set} (Rd_{set}): Number of written (read) cache lines.
CST conflicts per tx: Number of CST bits set. Median number of conflicting transactions encountered.
Average per-tx *W–W (R–W):* Avg. number of common locations between pair-wise conflicting transactions.

Table 7
Workload inputs.

Benchmark	Inputs
Workload set 1	
HashTable	1/3rd lookup, 1/3rd insert, 1/3rd delete
RBTree	1/3rd lookup, 1/3rd insert, 1/3rd delete
LFUCache	100% insert operation
RandomGraph	1/3rdlookup, 1/3rd insert, 1/3rd delete
Workload set 2	
Bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2
Delaunay	-a20 -i inputs/633.2
Genome	-g256 -s16 -n16384
Intruder	-a10 -l16 -n4096 -s1
Kmeans	-m10 -n10 -t0.05 -i inputs/random2048-d16-c16.txt
Labyrinth	-i random-x48-y48-z3-n64
Vacation	-n4 -q45 -u90 -r1048576 -t4194304
STMBench7	Reads-60%, Writes-40%. Short Traversals-40%. Long Traversals 5%, Ops.-45%, Mods. 10%

As Table 6(b) shows, the workloads we evaluate have varied dynamic characteristics. Delaunay and Genome perform a large amount of work per memory access and represent workloads in which time spent in the TM runtime is small compared to overall transaction latency. Kmeans is essentially data parallel and, along with the HashTable microbenchmark, represents workloads that are highly scalable with no noticeable level of conflicts. Intruder also has small transactions, but there is a high level of conflicts due to the presence of dueling write–write conflicts. The short transactions in HashTable, KMeans, and Intruder suggest that TM runtime overheads (if any) may become a significant fraction of total transaction latency. LFUCache and Randomgraph have a large number of conflicts and do not scale; any pathologies introduced by the TM runtime itself [6] are likely to be exposed. Bayes, Labyrinth, and Vacation have moderate working set sizes and significant levels of read–write conflicts due to the use of tree-like data structures. RBTree is a microbenchmark version of Vacation. STMBench7 is the most sophisticated application in our suite. It has a varied mix of large and small transactions with varying types and levels of conflicts [18].

Evaluation dimensions. We have designed the experiments to address the following questions

- How does FlexTM perform relative to hybrid TMs, hardware-accelerated STMs, and STMs?
- How does FlexTM's CST-based parallel commit compare to a centralized hardware arbiter design?
- How do the virtualization mechanisms deployed in FlexTM and FlexTM-S compare to previously proposed software instrumentation (SigTM [30]) and virtual memory-based implementations [11]?

8.2. FlexTM vs. hybrid TMs and STMs

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

Runtime systems. We evaluate FlexTM and compare it against two different sets of hybrid TMs and STMs with two different sets of workloads.

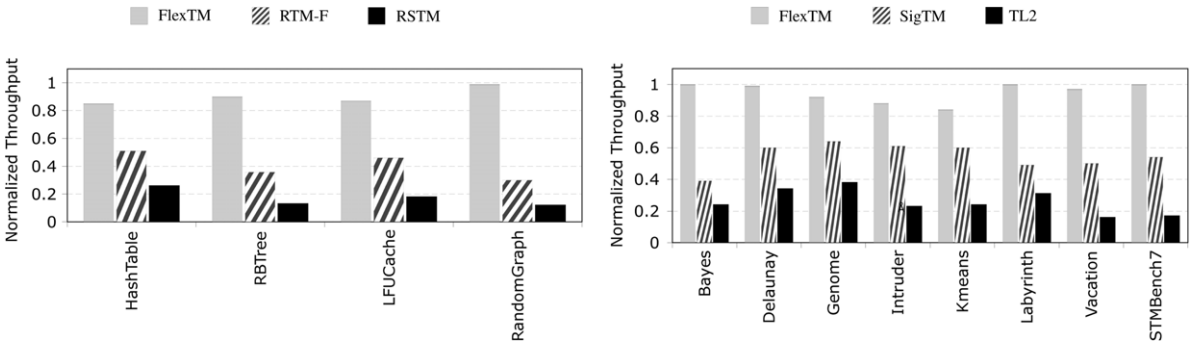
Workload set 1 (WS1) interfaces with three TM systems: (1) FlexTM; (2) RTM-F [39], a hardware accelerated STM system; and (3) RSTM [28], a non-blocking STM for legacy hardware (configured to use invisible readers, with self validation for conflict detection). Workload set 2 (WS2), which uses a different API, interfaces with (1) FlexTM, (2) TL2, a blocking STM for legacy hardware [14], and (3) SigTM [30], a hybrid TM derived from TL2 that uses hardware to accelerate conflict detection. FlexTM, the hybrids (SigTM and RTM-F), and the STMs (RSTM and TL2) have all been set up to perform *Lazy* conflict resolution.

We use the "Polka" conflict manager [36] in FlexTM, RTM-F, SigTM, and RSTM. TL2 limits the choice of contention manager and uses a timestamp manager with backoff. While all runtime systems execute on our simulated hardware, RSTM and TL2 make no use of FlexTM's extensions. RTM-F uses only PDI and AOU, and SigTM uses only the signatures (R_{sig} and W_{sig}). FlexTM uses all the presented mechanisms. Average speedups reported are geometric means.

Results. Fig. 6 shows the performance (transactions/sec) normalized to sequential thread performance for 1 thread runs. This demonstrates that the overheads of FlexTM are minimal. For small transactions (e.g., Hashtable) there is some overhead ($\simeq$15%) for the checkpointing of processor registers, which FlexTM performs in software — it could take advantage of checkpointing hardware if it exists.

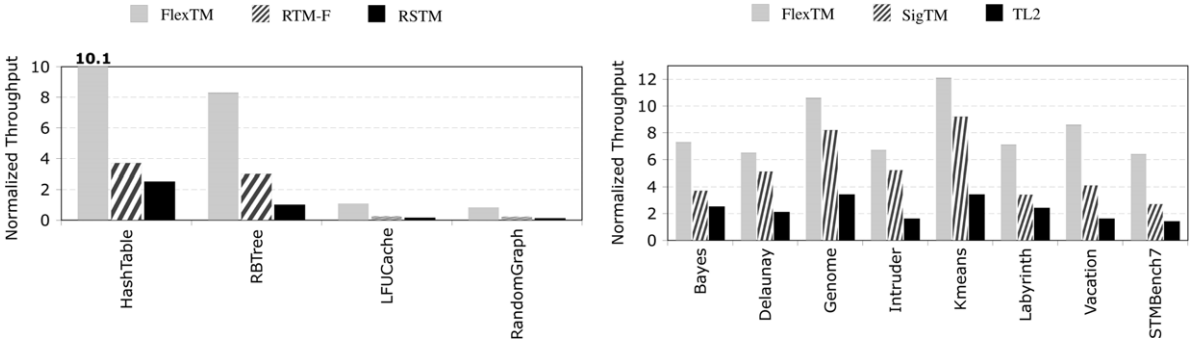
We study scaling and performance with 16 thread runs (Fig. 7). To illustrate the usefulness of CSTs (see the table in Fig. 7), we also report the number of conflicts encountered and resolved by an average transaction—the number of bits set in the *W–R* and *W–W* CST registers.

The performance of both STMs suffer from the bookkeeping required to track data versions, detect conflicts, and guarantee a



(a) Workload set 1. (b) Workload set 2.

Fig. 6. Throughput (transactions/10^6 cycles), normalized to sequential thread. All performance bars use 1 thread.



(a) Workload set 1. (b) Workload set 2.

Fig. 7. Throughput (transactions/10^6 cycles), normalized to sequential thread. All performance bars use 16 threads.

consistent view of memory (validation). RTM-F exploits AOU and PDI to eliminate validation and copying overhead, but still incurs bookkeeping that accounts for 40%–50% of execution time. SigTM uses signatures for conflict detection but performs versioning entirely in software. On average, the overhead of software-based versioning is smaller than that of software-based conflict detection, but it still accounts for as much as 30% of execution time for some workloads (e.g., STMBench7). Because it supports only lazy conflict detection, SigTM has simpler software metadata than RTM-F. RTM-F tracks conflicts for each individual transactional location and could vary the eagerness on a per-location basis.

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 sequential thread performance. 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 and 5.5× over RSTM on WS1. On WS2, FlexTM has an average speedup of 1.7× over SigTM and 4.5× over TL2.

HashTable and RBTree both scale well and have significant speedup over sequential thread performance, 10.3× and 8.3× respectively. 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. RTM-F manages to eliminate the validation cost and copying cost, but unfortunately the metadata management hinders performance improvement. FlexTM streamlines transaction execution and provides 2.8× and 8.3× speedup over RTM-F and RSTM respectively.

LFUCache and RandomGraph do not scale (no performance improvement compared to sequential thread performance). In LFUCache, conflict for popular keys in the Zipf distribution forces transactions 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 conflicts between transactions updating the same region in the graph cause all TM systems to experience significant levels of wasted work. The average RandomGraph transaction reads ∼60 cache lines and writes ∼9 cache lines. 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.

In applications with large access set sizes (i.e., Vacation, Bayes, Labyrinth, and STMBench7), TL2 suffers from the bookkeeping required prior to the first read (i.e., for checking write sets), after each read, and at commit time (for validation) [14]. This instrumentation accounts for ≃40% of transaction execution time. SigTM uses signatures-based conflict detection to eliminate this overhead. Unfortunately, both TL2 and SigTM suffer from another source of overhead: given lazy conflict resolution, reads need to search the redo log to see previous writes by their own transaction. Furthermore, the software commit protocol needs to lock the metadata, perform the copyback, and then release the locks. FlexTM eliminates the cost of versioning and conflict detection and improves performance significantly, averaging 2.1× speedup over SigTM and 4.8× over TL2.

Genome and Delaunay are workloads with a large ratio between the transaction size and the number of accesses. TL2's instrumentation on the reads does add significant overhead and affects its scalability—only 3.4× and 2.1× speedup (at 16 threads) over sequential thread performance for Genome and Delaunay respectively. SigTM eliminates the conflict detection overhead and significantly improves performance—an average of 2.4× improvement over TL2. FlexTM, in spite of the additional hardware support,

improves performance by 22%, since the versioning overheads account for a smaller fraction of overall transactional execution.

Finally, Kmeans and Intruder have unusually small transactions. Software handlers add significant overhead in TL2. In Kmeans, SigTM eliminates conflict detection overhead to improve performance by 2.7× over TL2. Since the write sets are small, eliminating the versioning overheads in FlexTM only improves performance a further 24%. Intruder has a high level of conflicts, and doesn't scale well, with a 1.6× speedup for FlexTM over sequential thread performance (at 16 threads). Both SigTM and FlexTM eliminate the conflict detection handlers and streamline the transactions, which leads to a change in the conflict pattern (fewer conflicts). This improves performance significantly—3.3× and 4.2× over TL2 for SigTM and FlexTM respectively. As in Kmeans, the versioning overheads are smaller and FlexTM's improvement over SigTM is restricted to 23%.

8.3. FlexTM vs. central-arbiter Lazy HTMs

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

As shown in Table 6(b), the number of conflicts encountered by a transaction is small compared to the total number of concurrent transactions in the system. Even in workloads that have a large number of conflicts (LFUCache and RandomGraph) a transaction typically encounters conflicts only about 30% of the time. Scalable workloads (e.g., Vacation, Kmeans) encounter essentially no conflicts. 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. Also, a transaction's commit overhead in FlexTM is not a constant, but rather proportional to the number of conflicting transactions encountered.

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 and arbiter (latencies are listed in Table 6(a)). The *Serial* version services only one commit request at a time (queuing up any others); the *Parallel* services all non-conflicting transactions in parallel (assuming infinite buffers in the arbiter). *Central* arbiters are similar in spirit to BulkSC [8], but serve only to order commits; they do not interact with the L2 directory.

We present results (see Fig. 8) for all out workloads and 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, RBTree, LFUCache, Kmeans, and Intruder). CSTs simplify the commit process: in the absence of conflicts, a commit requires only a single memory operation on a transaction's cached status word. On these workloads, CSTs improve performance by an average of 25% even over the aggressive *Central-Parallel*, which only serializes a transaction commit if it conflicts with an already in flight commit.

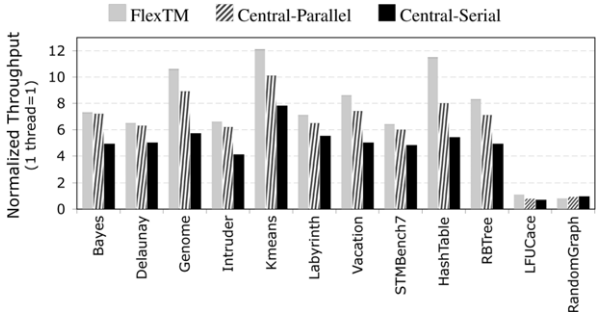


Fig. 8. FlexTM vs. centralized hardware arbiters.

- Workloads that exhibit inherent parallelism with *Lazy* conflict resolution (all except LFUCache and RandomGraph) suffer from serialization of commits in *Central-Serial*. *Central*-Serial essentially queues up transaction commits and introduces the commit latency of even other non-conflicting transactions onto the critical path. The serialization of commits could also change the conflict pattern. In some workloads (e.g., Intruder, STMBench7), in the presence of reader-writer conflicts as the reader transaction waits for predecessors to release the arbiter resource, the reader could be aborted by the conflicting writer. In a system that allows parallel commits, the reader could finish earlier and elide the conflict entirely. CST-based commit provides an average of ≃50% and a maximum of 112% (HashTable) improvement over *Central-Serial*. *Central-Parallel* removes the serialization overhead, but still suffers from commit arbitration latency.
- In benchmarks with high conflict levels (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 interprocessor message latency), and eliminate the problem as effectively as *Central*. *Lazy* conflict resolution inherently eliminates livelocks as well [41,44].

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).

8.4. FlexTM-S vs. other virtualization mechanisms

To study TM virtualization mechanisms, we downgrade our private L1 caches to 32 KB 2-way. This ensures that, in spite of the moderate write set sizes in our workloads, they experience overflows due to associativity constraints. Every L1 has access to a 64 entry SM-cache. Each metadata entry is 136 bytes.

We use five benchmarks in our study: Bayes, Delaunay, Labyrinth, and Vacation from the STAMP suite, and STMBench7. As Table 6(b) shows, these benchmarks have the largest write sets and are most likely to generate L1 cache overflows, enabling us to highlight tradeoffs among the various virtualization mechanisms. The fraction of total transactions that experience overflows in Bayes, Delaunay, Labyrinth, Vacation and STMBench7 is 11%, 8%, 25%, 9% and 32% respectively.

We compare FlexTM-S's performance against the following *Lazy* TM systems: (1) FlexTM, which employs a hardware controller for overflowed state and signatures for conflict detection;

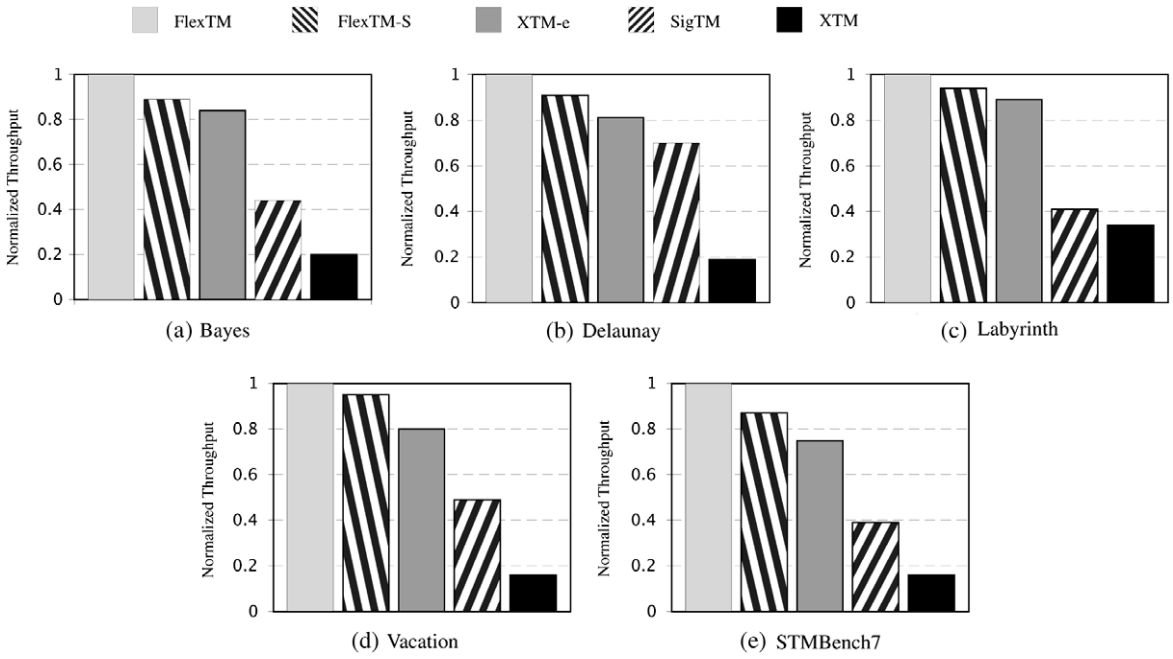


Fig. 9. Throughput at 16 threads for FlexTM-S vs. other TMs, normalized to FlexTM.

(2) XTM [11], which uses virtual memory to implement all TM operations; (3) XTM-e, which employs virtual memory support for versioning but performs conflict detection using cache-line granularity tag bits; and (4) SigTM [30], which uses hardware signatures for conflict detection and software instrumentation for word-granularity versioning. All systems employ the Polka [37] contention manager.

Result 3. *A software maintained metadata cache is sufficient to provide virtualization support with negligible overhead.*

As shown in Fig. 9, FlexTM-S imposes modest performance penalty (10%) compared to FlexTM. This is encouraging since it is vastly simpler to implement the SM-cache than the controller in FlexTM. The SM-cache miss and copyback handlers are the main contributors to the overhead. Unlike FlexTM and FlexTM-S, which version only the overflowed cache lines, XTM and XTM-e suffers from the overhead of page-granularity versioning. XTM's page-granularity conflict detection also leads to excessive aborts. XTM and XTM-e both rely on heavyweight OS mechanisms; by contrast, FlexTM-S requires only user-level interrupt handlers. Finally, SigTM incurs significant overhead due to software lookaside checks to determine if an accessed location is being buffered.

We also analyzed the influence of signature false positives. In FlexTM-S, write signature false positives can lead to increased handler invocation for loading the SM-cache, but the software metadata can be used to disambiguate and avoid abort penalty. In FlexTM, signature responses are treated as true conflicts, and cause contention manager invocations that could lead to excessive aborts. We set the W_{sig} and O_{sig} to 32 bits (see Fig. 10) to investigate the performance penalties of small write signatures.

Result 4. *As Fig. 10 shows, FlexTM-S's use of software metadata to disambiguate false positives helps reduce the needed size of hardware signatures while maintaining high performance.*

9. Conclusions

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. The resulting system provides

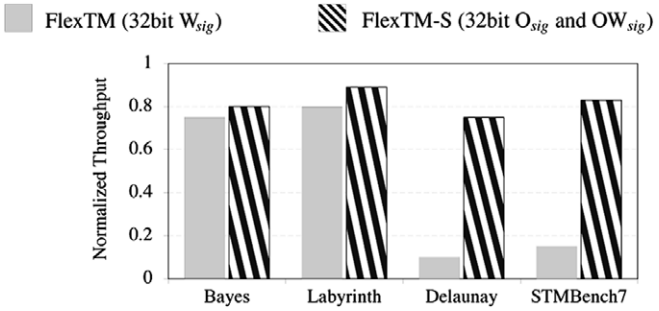


Fig. 10. Signature size effect (relative to FlexTM with 2048-bit W_{sig}).

TM support that decouples the conflict detection mechanism from conflict resolution time and allows software to control the latter (i.e., *Eager* , *Lazy* or *Mixed*), resulting in a high performance TM substrate on which software can dictate policy. To the best of our knowledge, it is the first hardware TM to admit an STM-like distributed commit protocol, allowing an unbounded number of *Lazy* and/or *Eager* transactions to arbitrate and commit in parallel. To virtualize transaction state, we propose two alternative designs—an aggressive hardware controller and a complexity-effective hardware–software design. The latter was evaluated via the FlexTM-S TM system, which further simplifies the versioning mechanism by supporting a *Mixed* mode for conflict resolution.

On a variety of benchmarks, FlexTM imposes minimal TM runtime overheads (comparable to sequential thread latency) and attains $\sim$5$\times$ more performance than STM and $\sim$1.8$\times$ more performance than hybrid TMs. Experiments with centralized commit schemes indicate that FlexTM's distributed protocol is free from the arbitration and serialization overheads of centralized hardware managers. Finally, comparing FlexTM-S with other virtualization mechanisms, we find that it is a complexity-effective alternative with $<$10% performance loss compared to the base FlexTM with full hardware-based overflow controller support.

Though we do not elaborate on the possibility here, we have also begun to experiment with non-TM uses of our decoupled hardware [39,40, TR version]; we expect to extend this work by developing more general interfaces and exploring their applications.

References

[1] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, S. Lie, Unbounded transactional memory, in: Proc. of the 11th Intl. Symp. on High Performance Computer Architecture, San Francisco, CA, February 2005, pp. 316–327.

[2] L. Baugh, N. Neelakantan, C. Zilles, Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory, in: Proc. of the 35th Intl. Symp. on Computer Architecture, Beijing, China, June 2008.

[3] B.H. Bloom, Space/time trade-off in hash coding with allowable errors, Communications of the ACM 13 (7) (1970) 422–426.

[4] C. Blundell, E.C. Lewis, M.M.K. Martin, Subtleties of transactional memory atomicity semantics, IEEE Computer Architecture Letters 5 (2) (2006).

[5] J. Bobba, K.E. Moore, H. Volos, L. Yen, M.D. Hill, M.M. Swift, D.A. Wood, Performance pathologies in hardware transactional memory, in: Proc. of the 34th Intl. Symp. on Computer Architecture, San Diego, CA, June 2007, pp. 32–41.

[6] J. Bobba, N. Goyal, M.D. Hill, M.M. Swift, D.A. Wood, TokenTM: efficient execution of large transactions with hardware transactional memory, in: Proc. of the 35th Intl. Symp. on Computer Architecture, Beijing, China, June 2008.

[7] L. Ceze, J. Tuck, C. Cascaval, J. Torrellas, Bulk disambiguation of speculative threads in multiprocessors, in: Proc. of the 33rd Intl. Symp. on Computer Architecture, Boston, MA, June 2006.

[8] L. Ceze, J. Tuck, P. Montesinos, J. Torrellas, BulkSC: bulk enforcement of sequential consistency, in: Proc. of the 34th Intl. Symp. on Computer Architecture, San Diego, CA, June 2007.

[9] H. Chafi, J. Casper, B.D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, K. Olukotun, A scalable, non-blocking approach to transactional memory, in: Proc. of the 13th Intl. Symp. on High Performance Computer Architecture, Phoenix, AZ, February 2007.

[10] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M.V. Biesbrouck, G. Pokam, B. Calder, O. Colavin, Unbounded page-based transactional memory, in: Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 2006, pp. 347–358.

[11] J. Chung, C. Cao Minh, A. McDonald, T. Skare, H. Chafi, B.D. Carlstrom, C. Kozyrakis, K. Olukotun, Tradeoffs in transactional memory virtualization, in: Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 2006, pp. 371–381.

[12] L. Dalessandro, M.L. Scott, Strong isolation is a weak idea, in: 4th ACM SIGPLAN Workshop on Transactional Computing, Raleigh, NC, February 2009.

[13] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, D. Nussbaum, Hybrid transactional memory, in: Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 2006.

[14] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: Proc. of the 20th Intl. Symp. on Distributed Computing, Stockholm, Sweden, September 2006, pp. 194–208.

[15] K. Fraser, T. Harris, Concurrent programming without locks, ACM Transactions on Computer Systems 25 (2) (2007) article 5.

[16] 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, M. Lanzerotti, Design of the Power6 Microprocessor, in: Proc. of the Intl. Solid State Circuits Conf., San Francisco, CA, February 2007, pp. 96–97.

[17] J.R. Goodman, Coherency for multiprocessor virtual address Caches, in: Proc. of the 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, October 1987, pp. 72–81.

[18] R. Guerraoui, M. Kapaĺka, J. Vitek, STMBench7: a benchmark for software transactional memory, in: Proc. of the 2nd EuroSys, Lisbon, Portugal, March 2007.

[19] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, K. Olukotun, Transactional memory coherence and consistency, in: Proc. of the 31st Intl. Symp. on Computer Architecture, München, Germany, June 2004.

[20] M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer III, Software transactional memory for dynamic-sized data structures, in: Proc. of the 22nd ACM Symp. on Principles of Distributed Computing, Boston, MA, July 2003, pp. 92–101.

[21] M. Herlihy, J.E. Moss, Transactional memory: architectural support for lock-free data structures, in: Proc. of the 20th Intl. Symp. on Computer Architecture, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, December 1992.

[22] M.D. Hill, D. Hower, K.E. Moore, M.M. Swift, H. Volos, D.A. Wood, A case for deconstructing hardware transactional memory systems, Technical Report 1594, Dept. of Computer Sciences, Univ. of Wisconsin–Madison, 2007.

[23] S. Kumar, M. Chu, C.J. Hughes, P. Kundu, A. Nguyen, Hybrid transactional memory, in: Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming, New York, NY, March 2006.

[24] J.R. Larus, R. Rajwar, Transactional Memory, in: Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2007.

[25] J. Laudon, D. Lenoski, The SGI origin: a ccNUMA highly scalable server, in: Proc. of the 24th Intl. Symp. on Computer Architecture, Denver, CO, June 1997.

[26] Y. Lev, J.-W. Maessen, Split hardware transaction: true nesting of transactions using best-effort hardware transactional memory, in: Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming, Salt Lake City, UT, February 2008.

[27] V.J. Marathe, W.N. Scherer III, M.L. Scott, Adaptive software transactional memory, in: Proc. of the 19th Intl. Symp. on Distributed Computing, Cracow, Poland, September 2005.

[28] V.J. Marathe, M.F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W.N. Scherer III, M.L. Scott, Lowering the overhead of software transactional memory, in: Proc. of the 1st ACM SIGPLAN Workshop on Transactional Computing, Ottawa, ON, Canada, June 2006. Expanded version available as TR 893, Dept. of Computer Science, Univ. of Rochester, March 2006.

[29] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, D.A. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, in: ACM SIGARCH Computer Architecture News, September 2005.

[30] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, K. Olukotun, An effective hybrid transactional memory system with strong isolation guarantees, in: Proc. of the 34th Intl. Symp. on Computer Architecture, San Diego, CA, June 2007.

[31] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, D.A. Wood, LogTM: log-based transactional memory, in: Proc. of the 12th Intl. Symp. on High Performance Computer Architecture, Austin, TX, February 2006.

[32] R. Rajwar, M. Herlihy, K. Lai, Virtualizing transactional memory, in: Proc. of the 32nd Intl. Symp. on Computer Architecture, Madison, WI, June 2005.

[33] B. Saha, A.-R. Adl-Tabatabai, R.L. Hudson, C. Cao Minh, B. Hertzberg, McRT-STM: a high performance software transactional memory system for a multi-core runtime, in: Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming, NY, USA, March 2006.

[34] N. Sakran, M. Yuffe, M. Mehalel, J. Doweck, E. Knoll, A. Kovacs, The implementation of the 65 nm Dual-Core 64b merom processor, in: Proc. of the Intl. Solid State Circuits Conf., San Francisco, CA, February 2007.

[35] D. Sanchez, L. Yen, M.D. Hill, K. Sankaralingam, Implementing signatures for transactional memory, in: Proc. of the 40th Intl. Symp. on Microarchitecture, Chicago, IL, December 2007.

[36] W.N. Scherer III, M.L. Scott, Advanced contention management for dynamic software transactional memory, in: Proc. of the 24th ACM Symp. on Principles of Distributed Computing, Las Vegas, NV, July 2005.

[37] W.N. Scherer III, M.L. Scott, Randomization in STM contention management (poster paper), in: Proc. of the 24th ACM Symp. on Principles of Distributed Computing, Las Vegas, NV, July 2005.

[38] M.L. Scott, Sequential specification of transactional memory semantics, in: Workshop on 1st ACM SIGPLAN Workshop on Transactional Computing, Ottawa, ON, Canada, June 2006.

[39] A. Shriraman, M.F. Spear, H. Hossain, S. Dwarkadas, M.L. Scott, An integrated hardware–software approach to flexible transactional memory, in: Proc. of the 34th Intl. Symp. on Computer Architecture, San Diego, CA, June 2007. Earlier but expanded version available as TR 910, Dept. of Computer Science, Univ. of Rochester, December 2006.

[40] A. Shriraman, S. Dwarkadas, M.L. Scott, Flexible decoupled transactional memory support, in: Proc. of the 35th Intl. Symp. on Computer Architecture, Beijing, China, June 2008. Expanded version available as TR 925, URCS, November 2007.

[41] A. Shriraman, S. Dwarkadas, Refereeing conflicts in hardware transactional memory, in: Proc. of the 2009 ACM Intl. Conf. on Supercomputing, NY, USA, June 2009.

[42] M.F. Spear, V.J. Marathe, W.N. Scherer III, M.L. Scott, Conflict detection and validation strategies for software transactional memory, in: Proc. of the 20th Intl. Symp. on Distributed Computing, Stockholm, Sweden, September 2006.

[43] M.F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, M.L. Scott, Alert-on-update: a communication aid for shared memory multiprocessors (poster paper), in: Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming, San Jose, CA, March 2007.

[44] M.F. Spear, L. Dalessandro, V. Marathe, M.L. Scott, A comprehensive strategy for contention management in software transactional memory, in: Proc. of the 14th ACM Symp. on Principles and Practice of Parallel Programming, March 2009.

[45] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, M. Valero, EazyHTM, eager–lazy hardware transactional memory, in: Proc. of the 42th Intl. Symp. on Microarchitecture, NY, USA, December 2009.

[46] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Valos, M.D. Hill, M.M. Swift, D.A. Wood, LogTM-SE: decoupling hardware transactional memory from Caches, in: Proc. of the 13th Intl. Symp. on High Performance Computer Architecture, Phoenix, AZ, February 2007.

[47] C. Zilles, L. Baugh, Extending hardware transactional memory to support non-busy waiting and non-transactional actions, in: Proc. of the 1st ACM SIGPLAN Workshop on Transactional Computing, Ottawa, ON, Canada, June 2006.

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

[49] The Rochester Software Transactional Memory Runtime, 2006, www.cs.rochester.edu/research/synchronization/rstm/.

Arrvindh Shriraman is a graduate student in computer science at the University of Rochester. He received his B.E. from the University of Madras, India, and his M.S. from the University of Rochester. His research interests include multiprocessor system design, hardware–software interface, and parallel programming models.

Sandhya Dwarkadas is a Professor of Computer Science and of Electrical and Computer Engineering at the University of Rochester. Her research lies at the interface of hardware and software with a particular focus on concurrency, resulting in numerous publications that cross areas within systems. She has recently been associate editor for IEEE Computer Architecture Letters (2006–2009), program and general chair for *ISPASS'07* and *ISPASS'08* respectively, and is a past associate editor for IEEE Transactions on Parallel and Distributed Systems.

Michael L. Scott is a Professor and past Chair of the Computer Science Department at the University of Rochester. He is a Fellow of the ACM and the IEEE, a recipient of the Dijkstra Prize in Distributed Computing, and author of the textbook *Programming Language Pragmatics* (3rd edition, Morgan Kaufmann, 2009). He was recently Program Chair of *TRANSACT'07* and of *PPoPP'08*.