LiteTM: Reducing Transactional State Overhead

Syed Ali Raza Jafri, Mithuna Thottethodi, and T. N. Vijaykumar
School of Electrical and Computer Engineering
Purdue University
{sjafri, mithuna, vijay}@ecn.purdue.edu

Abstract- Transactional memory (TM) has been proposed to address some of the programmability issues of chip multiprocessors. Hardware implementations of transactional memory (HTMs) have made significant progress in providing support for features such as long transactions that spill out of the cache, and context switches, page and thread migration in the middle of transactions. While essential for the adoption of HTMs in real products, supporting these features has resulted in significant state overhead. For instance, TokenTM adds at least 16 bits per block in the caches which is significant in absolute terms, and steals 16 of 64 (25%) memory ECC bits per block, weakening error protection. Also, the state bits nearly double the tag array size. These significant and practical concerns may impede the adoption of HTMs, squandering the progress achieved by HTMs. The overhead comes from tracking the thread identifier and the transactional read-sharer count at the L1-block granularity. The thread identifier is used to identify the transaction, if only one, to which an L1-evicted block belongs. The read-sharer count is used to identify conflicts involving multiple readers (i.e., write to a block with non-zero count). To reduce this overhead, we observe that the thread identifiers and read-sharer counts are not needed in a majority of cases. (1) Repeated misses to the same blocks are rare within a transaction (i.e., locality holds). (2) Transactional read-shared blocks that both are evicted from multiple sharers' L1s *and* are involved in conflicts are rare. Exploiting these observations, we propose a novel HTM, called *LiteTM*, which completely eliminates the count and identifier and uses software to infer the lost information. Using simulations of the STAMP benchmarks running on 8 cores, we show that LiteTM reduces TokenTM's state overhead by about 87% while performing within 4%, on average, and 10%, in the worst case, of TokenTM.

1 Introduction

Chip multiprocessors (CMPs) are emerging as a better alternative to uniprocessors in terms of power dissipation and performance. However, CMPs require parallel programming which is significantly harder than sequential programming. A key programmability issue is that locks can result in undesirable behavior (e.g., deadlocks, livelocks, and data races). To address this issue, researchers are currently exploring transactional memory programming models (TM), based on databases' transactional processing [15]. Transactions achieve atomic behavior without specifying explicit locks by ensuring that the read and write accesses of one transaction do not conflict with another transaction (i.e., a read from or write to a memory location should not witness a write to the same location from another concurrent transaction). As such, transactions can better avoid the above undesirable behavior than locks. Building on the idea of providing hardware support for TM pioneered in [17], several hardware and software (HTM and STM) and hybrid implementations (e.g., [3,12,14,16,24,26,27,28]) have emerged.

While STMs are slow due to their overhead of software conflict detection for every transactional access (true also for software transactions in HTM-STM hybrids), HTMs use hardware to achieve fast conflict detection. Specifically, by exploiting the fact that both TM and coherence enforce the multiple-reader-single-writer invariant at the block granularity, HTMs optimize performance by piggybacking conflict detection on coherence. That is, HTMs elide conflict detection on cache hits and bundle conflict detection on misses as part of miss processing with little increase in latency. Therefore, we focus on HTMs which now provide support for features such as (1) long transactions that exceed the cache capacity [3,6,7,12,13,24], (2) context switches and page and thread migrations in the middle of a transaction [7,28]; and most recently, (3) avoiding coherence protocol changes, which invariably lead to subtle correctness issues and hinder wide-spread adoption [7]. These features are *essential* for HTMs to be adopted in real products.

TokenTM [7], a comprehensive and elegant proposal, supports all the above features, but incurs high state overhead. While some HTMs do not incur such overhead (e.g., signature-based HTMs [10,11,27,28]), they do not provide all the above features (as discussed in Section 2). Other HTMs which support *some* of the features using per-block state (e.g., VTM [24], OneTM-concurrent [6]) also incur such overhead. Though we focus on TokenTM, we discuss later that our techniques are applicable to these other HTMs. TokenTM's overhead comes from two sources. First, to allow conflict detection in the presence of L1 cache evictions of transactional blocks, TokenTM maintains a count in the shared L2 of L1-evicted, transactional read sharers. The count can quickly detect conflicts (i.e., if writes encounter a non-zero count). Further, when a block has only one transactional sharer then the storage space for the count can be used to hold the sharer's thread identifier. This identifier serves two purposes (1) to identify the conflicting threads in the case of a conflict, and (2) to allow a transaction access to its own transactional blocks that have either been evicted to lower levels or been moved to other caches by coherence (i.e., to avoid self-conflict which would lead to a livelock). To allow evictions from the L2, the count or the identifier need to be spilled to *all* the levels of the memory hierarchy, including main memory and even the disk. In addition to the count and identifier, TokenTM employs two state bits per block to distinguish among single reader, single writer and multiple readers. Second, to avoid changes to coherence, TokenTM uses additional bits, the count and identifier in L1, in addition to the traditional *R* and *W* bits. Thus, TokenTM incurs significant state overhead (e.g., at least 16 bits per 64-byte block in all levels of the memory hierarchy). In addition, TokenTM requires additional flash-copy support in the L1 for context switches, increasing L1 area and latency.

One may think that the state overhead is a mere 3.3% (16 state bits for 64 data bytes). However, relative overhead does not capture

the following two concerns. First, to retrieve the transactional state with the data in memory in one access, TokenTM and other HTMs advocate stealing some of the memory ECC bits to hold the state. (Storing transactional state in regular memory would preserve ECC but require two accesses which would increase bandwidth pressure.) However, stealing as many as 16 bits weakens error protection (e.g., 16 bits correspond to 25% of the 64 SECDED bits per 64-byte blocks), a concern in soft- and hard-error-prone scaled technologies. Second, an overhead of 16 bits per block in L1 and L2 is significant in absolute terms. For instance, such overhead is equivalent to nearly doubling the tag array in a system with 40-bit physical addresses and 32-KB L1 and 8-MB L2. Because HTMs target commodity multicores where cost is a first-order constraint (as opposed to niche products where high-cost mechanisms may be acceptable), this overhead is a concern. Moreover, as cache and memory size scale in future generations, L1 block sizes are likely to remain around 64 bytes, causing the absolute overhead to grow considerably. These significant and practical concerns may impede the adoption of HTMs in real products, squandering the progress achieved by HTMs.

We propose a novel HTM, called *LiteTM*, to reduce the state overhead of the read-sharer count and thread identifier while supporting all the above features and maintaining high performance. Because the state bits are fundamental to guaranteeing transactional semantics, naively shrinking the state to fewer bits would violate correctness. Any such state reduction needs careful techniques to infer the lost information. LiteTM is based on the key observation that the counts and identifiers are needed neither for conflict detection in all cases nor for identifying conflicting transactions in a majority of cases. Consequently, we completely eliminate the counts and identifiers from the entire memory hierarchy and use software to handle the rest of the cases. LiteTM employs only two state bits per block in L1, L2, and main memory, which are adequate for hardware conflict detection. This overhead corresponds to only 3.3% of the 64 SECDED bits per 64-byte block compared to TokenTM's 25%. Additionally, there is no flash-copying in L1. LiteTM is a new design point in the spectrum of HTMs' hardware-software functionality split. TokenTM uses software to rollback program state upon aborts and to clear transactional state of L1-evicted blocks upon both commits and aborts, while detecting conflicts and identifying conflicting transactions in hardware. In contrast, LiteTM pushes the hardware-software split more towards software and decouples key parts of conflict handling for L1-evicted blocks; conflict detection is still in hardware but the conflicting transactions are identified in software using transactional logs. This decoupling is fundamental and can be applied to reduce the state overhead of other unbounded HTMs with per-block state (e.g., VTM, OneTM-concurrent). Because conflict detection is in hardware for all accesses, LiteTM provides strong atomicity. While LiteTM may seem like another HTM-STM hybrid, there is a key difference: conventional hybrids switch an *entire* transaction from HTM to STM when even a single transactional block is evicted from the L1, whereas LiteTM uses software *only for the L1-evicted blocks* while continuing to use hardware for L1-resident blocks. Because STMs detect conflicts in software incurring significant overhead for *every* access, conventional hybrids incur significant overhead as they switch to STM on routine hardware events like evictions. In contrast, LiteTM uses software only for evicted blocks, performing close to HTMs.

Targeting the read-sharer count, we observe that transactional read-shared blocks that both are evicted from multiple sharers' L1s *and* are involved in conflicts are rare. As mentioned above, the read-sharer count enables fast detection of such conflicts. However, eliminating the count poses a hurdle for clearing L2's and memory's transactional state in the uncommon case of read-shared L1-evicted blocks; without the count, we do not know when the last of the sharers commits or is aborted. To address this issue, we employ a novel *lazy clearing* in software by walking the logs of all the current transactions upon a conflict on an L1-evicted block. Because such all log-walks are expensive, we ensure that this case remains uncommon. LiteTM's two state bits in L2 and memory encode states that isolate the more common cases of single reader or writer for a block, where the state is cleared when the single reader or writer commits or aborts. The lazy clearing in OneTM-concurrent [6] refers to the *lazy update of the thread identifier* without any log-walks and works only in the restricted case where at most one transaction may spill out of L1. In contrast, LiteTM's *lazy clearing of transactional state* handles the general case of multiple, spilled transactions, requiring all log-walks. Furthermore, OneTM's requirement of an identifier per block is not removed by the lazy update.

Targeting the thread identifier, which exists only in single-sharer cases (multi-sharer cases have the count), we observe that both uses of the identifier — to identify conflicting transactions and to allow a transaction to access its own blocks — are uncommon. For the first use, most conflicts occur for in-L1-cache blocks where the conflicting transactions are trivially identified by the caches involved in the conflict. For the less-common conflicts on evicted blocks, we identify the conflicting transactions in software by walking all the transactions' logs (as also done in TokenTM in extremely rare cases). For the second use, we observe that repeated misses to the same blocks are rare within a transaction (i.e., locality holds). For the infrequent case of a transaction accessing its own evicted block, we employ a novel *self log-walk* to check the transaction's own read and write sets.

Finally, targeting TokenTM's *R*, *W*, and additional bits in L1, we observe that we can leverage coherence for implicitly differentiating between reads and writes in the common case, without explicitly using *R* and *W* (not differentiating would lead to many false conflicts, as seen in some STAMP benchmarks). Accordingly, LiteTM uses a single *T* bit and employs the novel idea that *conservatively but closely* approximates *W* by combining L1 'Modified' coherence state and the *T* bit. That is, a modified block with the *T* bit set is considered as transactionally written. Because Modified and *T* combination is a superset of *W*, this *W-approximation* does not miss any real conflicts. However, false conflicts are possible but only in uncommon cases which we explain later.

The key contributions of this paper are:

- LiteTM compensates for the loss of information in terms of separate *R* and *W* bits, the read-sharer count, and the thread identifier, respectively, via the following novel ideas: (1) *W-approximation* for L1-resident blocks, (2) lazy clearing of transactional state in L2 and memory, (3) self log-walk to identify a transaction's own blocks;
- Using simulations of the STAMP benchmarks running on 8 cores, we show that LiteTM reduces TokenTM's state overhead

by about 87% while performing within 4%, on average, and 10%, in the worst case, of TokenTM. LiteTM uses two bits per block in L1, L2, and main memory, whereas TokenTM uses 19 bits in L1, and 16 bits in L2 and memory. In contrast, STMs and HTM-STM hybrids need at least one bit per block in L1, L2, and memory for strong atomicity and an upper bound on hybrids' performance shows at least 44% average performance degradation over TokenTM.

The rest of the paper is organized as follows. In Section 2, we contrast LiteTM to previous proposals. We describe TokenTM in Section 3 and LiteTM in Section 4. We describe our experimental methodology in Section 5. We discuss our experimental results in Section 6 and conclude in Section 7.

2 Related Work

Conceptually, TMs maintain metastate in a matrix of memory blocks (rows) and threads (columns) where each entry records read and write accesses for a block-thread pair. The key challenge in making TM support fast is that each access (transactional and non-transactional) requires a lookup of the entire row for the memory block to check for conflict across threads followed by an update of the row with the access, whereas a transaction commit or abort requires clearing of the entire column holding the thread's transactional state. However, quick access to the entire matrix indexed by both rows and columns is hard to implement. To address this issue, HTMs exploit the fact that both TM and coherence enforce the multiple-reader-single-writer invariant at the block granularity to employ the crucial performance optimization of piggybacking conflict detection on coherence (i.e., the functional equivalent of row-lookup in our matrix analogy). In addition to optimizing conflict detection, HTMs optimize clearing of transactional state on commits and aborts by flash-clearing the state in the cache whenever all the transactional data fits in the cache (i.e., the column-clear in our matrix analogy). While "coherence+flash-clear" offers a natural way to implement the TM matrix for the case when all transactional state is within the caches, the conceptual 2-D matrix model is impractical when we consider virtualizing TM implementations to accommodate blocks that are evicted and threads that are context switched in/out.

Early solutions maintain spilled transactional metastate in custom hardware or software structures which caused significant hardware complexity [3] or software overhead [12,24]. More recent TM implementations commonly rely on one of two common simplifications. The first approach uses signature-based TM implementations effectively maintain the entire column (per-thread read/write sets) in hash-based Bloom-filter signatures [10,11,27,28]. They do not maintain any per-block information. Such implementations that omit per-block transactional state are not scalable in (a) system size, since conflict detection requires comparison with signatures of all other threads, which inherently requires broadcast across all hardware thread contexts to effectively lookup the per-address state across all threads, and (b) read/write set sizes, since large transactions cause signature saturation which results in a sharp performance loss when transactions are large [7]. One may think that signature saturation may be eliminated by increasing the size of the signature. However, hardware limitations prevent hash signatures that are large enough to avoid saturation because large signatures slow down *all* accesses [25].

An alternative approach to avoid the signature saturation problem is to associate some state with each memory/cache block (e.g., VTM [24], OneTM-concurrent [6], and TokenTM [7]). As a natural consequence of maintaining per-block state, commits may become slower since the transactional state associated with that transaction (the column in the matrix) that has spilled to memory must be cleared one-at-a-time (unlike cache-bits which may be flash cleared). Although VTM stores per-block metadata, the metadata is not co-located with the corresponding memory block. Instead the metadata is spilled to separate global table (called XADT) that must be searched for conflict detection. In addition to the slow-commit problem, XADT searches slow down conflict detection for all accesses in the presence of any spilled metadata (although some searches may be avoided by using a Bloom filter). OneTM-concurrent overcomes the slow-commit problem by limiting concurrency to at most one overflowed transaction which enables logical clearing of metastate by keeping track of the current overflowed transaction. Metastate of older transactions is implicitly invalid and may be cleared lazily. TokenTM may be viewed as a generalization of OneTM-concurrent that allows multiple "spilled" transactions simultaneously. Unfortunately, both TokenTM and OneTM-concurrent require large amounts of transactional state (16 bits per L1 block). LiteTM's state reduction techniques are applicable to unbounded HTMs with per-block state (e.g, TokenTM, VTM, OneTM-concurrent).

Finally, hybrid TMs use HTM-based execution as a preferred fast-path and fall-back on a slower STM upon virtualization events (e.g., replacements and context-switch) [14,19]. The limited HTMs in hybrids, though simpler because the HTMs do not need to support virtualization, may add significant hardware complexity (e.g., MetaTM [18] requires coherence changes). Further, hybrid TMs may not achieve strong atomicity (e.g., provide only single global lock isolation [14,18,19]) without additional per-block state (e.g., using UFO [5]). In contrast, LiteTM offers strong atomicity and outperforms hybrids by a significant margin while requiring modest additional state overhead.

3 TokenTM: Background

While our techniques are generic and applicable to other HTMs that use per-block state (e.g., VTM [24], OneTM-concurrent [6]), we choose TokenTM as the base scheme to describe the details of LiteTM because TokenTM comprehensively supports all the features described in Section 1. This choice allows to demonstrate that LiteTM can also support all the features while incurring less state overhead and maintaining high performance.

TokenTM maintains the invariant of multiple readers and single writer for transactional blocks. TokenTM implements the invariant by using an abstraction based on *tokens*, where (1) a transactional read to a block must acquire a token for the block; (2) a transactional write to a block must acquire all the tokens for the block; (3) a transaction commit or abort releases all the tokens acquired during the transaction. Read-write conflicts are detected when a read or a write cannot acquire its requisite number of tokens because a conflicting access already holds some or all of the tokens. TokenTM employs LogTM's transaction log [22] to maintain a transaction's read and write sets, and the previous version of the memory state to rollback memory state upon transaction aborts.

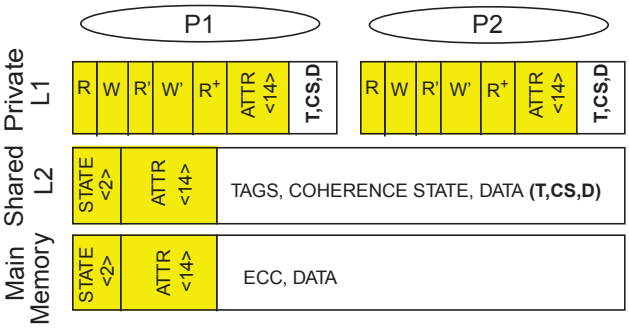

FIGURE 1: TokenTM Transactional State

TokenTM addresses two key issues. First, TokenTM allows long transactions, that may evict transactional blocks from the cache, by pushing transactional state all the way to memory, as shown in Figure 1. While this idea was previously proposed in OneTM which allows only one transaction to spill out of cache, TokenTM generalizes OneTM to allow more than one transaction to spill. Second. TokenTM provides TM support without changing the coherence protocol by using two techniques called *token fusion* and *fission*.

Despite the name, TokenTM does not *require* token coherence. The implementation for TokenTM in [7] assumes a conventional directory-based coherence protocol for private L1s with a shared L2. While every transactional access needs to acquire the appropriate number of tokens, piggybacking on coherence allows cache *hits* "to generate" token(s) locally. Such generation does not lead to undetected conflicts because both coherence and TokenTM enforce multiple-readers-single-writer invariant so that any conflicting access must trigger a global coherence event. TokenTM piggybacks on this event to detect the tokens held by other transactions and thus the conflict. Any non-conflicting transactional cache miss simply receives the data (via coherence) and the appropriate tokens. All acquires of tokens (reads or writes) are logged. TokenTM maintains a read bit (*R*) and a write bit (*W*) per L1 block *to signify* the holding of one token for a read and all the tokens for a write (first row under TokenTM column in Table 1). Transaction commit or abort releases the tokens by clearing the *R* and *W* bits using hardware in some cases and software handlers in the rest. All TM-related software functionality is implemented via *escape actions* [23].

We provide a high-level summary of TokenTM's state in Table 1. However, because TokenTM's state ensures transactional semantics in TokenTM and because our purpose is to reduce the state, a high-level description alone would not suffice. Any detailed description would inevitably involve some subtle correctness issues. The rest of the section gives some of these details.

3.1 Fusion and Fission

It is relatively easy to see that the acquire and release of tokens in non-conflicting cases can be done without changing the coherence protocol. While some previous HTMs, such as LogTM [22], handle conflicts via nacks which require protocol changes and rollbacks to break deadlocks of mutually-nacking transactions, TokenTM employs fusion of tokens to ensure that the coherence protocol remains unchanged even on conflicting accesses. The no-change stipulation requires that coherence actions should *complete as usual even on conflicts* (nacks disallow coherence completion creating potential for deadlocks). A key point here is that though coherence actions complete, the conflicting access does *not* complete and instead raises an access-fault exception which performs conflict resolution (i.e., rollback of a conflicting transaction). Coherence completion requires that the previously-acquired tokens of the conflicting block must be kept intact *through* the conflict so that the tokens are released properly either by a commit or abort. We explain how fusion achieves this goal by considering the three cases of conflicts: reads-followed-by-write, write-followed-by-read, and write-followed-by-write.

In a reads-followed-by-write conflict (writer and readers in different cores), the writer invalidates the readers as usual. However, the readers' tokens should not be lost in the invalidated blocks (so the tokens can be released properly in either case of the readers committing or aborting) and the conflict should be detected. To these ends, the readers' tokens are sent to the writer in the invalidation-acknowledgement payloads. Note that adding bits to payloads does not constitute a protocol change as long as there are no changes to states or transitions which are what raise correctness issues. The writer fuses the readers' tokens into its modified block and flags a conflict. Though the readers' tokens are physically present in the writer's block, the tokens belong to the readers and cause the write to fault. The writer proceeds only after the conflict is resolved assuming the writer is not aborted in the resolution. To record the readers' tokens in the writer's block, TokenTM uses the *R'* bit (for single reader), the R^+ bit (for multiple readers), and holds the thread identifier for single sharer or the read-sharer count for multiple readers (Figure 1 and second and third rows in Table 1) As part of the conflict handling, if the readers are aborted then their tokens are released (i.e., the *R'* in the writer's block is cleared or the read-sharer count is decremented) allowing the writer to acquire all the tokens and proceed. Or if the writer is aborted then there are no tokens to be released because none were acquired (the write did not complete).

In a write-followed-by-read conflict (writer and readers in different cores), the writer's token goes to the reader and sets the W' bit and the thread identifier to indicate a conflict, analogous to the R' bit (second row in Table 1). In addition, as part of the modified-block writeback, the state bits and thread identifier in the L2 are updated to prevent *new* reads before the writer commits or aborts. We explain the state bits in the L2 later in Section 3.3. A write-followed-by-write conflict is handled similarly.

While fusion involves many cases as described above, fission is for the relatively-easy case of readers joining non-conflicting sharing where new tokens are "generated" on the fly.

3.2 Commits and Aborts

Because fusion occurs only on conflicts without which a transaction's blocks remain in the cache (assuming no evictions which we handle a little later), commit of a transaction that does not encounter any conflicts in its lifetime, is fast. In *fast commit*, all the acquired tokens are released by a flash-clear of the *R* and *W* bits in the cache. However, the tokens of the transactions, that survive conflicts and reach commit, are fused in *other* caches. Therefore, such transactions undergo *slow commit* in which a software commit handler performs a log-walk of the read and write sets to release the tokens one at a time. Each token release for a read clears *exactly one* of an *R* or *R'* bit with matching thread identifier,

Table 1: TokenTM vs. LiteTM : Transactional state for Conflict Detection

	TokenTM		LiteTM	
	State	**Interpretation**	**State**	**Interpretation**
Private L1	*R/W*	Local thread has transactionally read/ written to block	*T+Clean/ T+Modified*	Local thread has transactionally read/ written to block.
	R'/W' with *ID*	Remote transaction *ID* has read/write to block.	*T'*	At least one unknown remote transaction has accessed block
	R+, count (=*N*)	At least *N* remote transactional readers exist	—	No explicit tracking of multiple remote transactional accesses.
Shared L2, memory (L1-evicted transactional state. Four states in L2)	*Single reader* (*ID*)	Transaction (*ID*) has read block	*Single-reader*	Some transaction has read block
	Multiple reader with count *N*	A total of *N* transactions (identities unknown) have read block	*Multiple-reader*	Multiple readers (number and identities unknown) have read block
	Single Writer (*ID*)	Transaction (*ID*) has written to block	*Single-Writer*	Some transaction has written to block
	Idle (encoded as *multiple reader* with *N* = 0)	No transactional state for block has overflowed from upper caches.	*Idle*	Same as TokenTM's state

in the absence of which decrements the read-sharer count associated with an R^+ bit (token release uses coherence to contact all the sharers with *R'* and R^+). Each token release for a write clears the *W, W'* and L2/directory state bits.

Though the blocks have moved from a transaction's cache, future conflicts on the blocks can be traced back to the transaction through the thread identifiers accompanying *R'* and *W'*, as long as there is only one reader. We call this case as a *fast abort*. If multiple readers have fused then only the read-sharer count is available and the identity of the readers is lost, requiring future conflicts to walk the logs of *all* the current transactions to identify the readers. We call this case as a *slow abort*. Because either cases of abort occur due to conflicts which cause tokens to be fused into non-local caches, flash-clear of the *R* and *W* bits is not possible. Consequently, a software abort handler releases the aborted transactions's tokens, similar to the commit handler.

3.3 Handling evictions

One of TokenTM's key features is support for long transactions that spill out of the cache. As L1 blocks get evicted, any tokens they carry are held in the L1 directory (at L2) and in memory when evicted from L2. Each L2 and memory entry maintains two state bits providing four states — *single-reader*, either *idle* (no sharers) or *multiple-reader* (more than one reader), *writer*, and *overflow* of read-sharer count — along with the thread identifier/read-sharer count (Figure 1 and "Shared L2" rows in Table 1). It is easy to see that *R*, *W*, *R'*, *W'*, and R^+ can be mapped to these states. The block's thread identifier or read-sharer count is held as is. As more *R*, *R'*, or R^+ evictions occur (evictions are non-silent), the read-sharer count goes up. Accesses from a transaction to its own previously-evicted blocks can be recognized using the identifier on the blocks and proceed without any questions of conflict. Because complete token information is available, conflicts on evicted blocks can be flagged. As in the case of in-cache conflicts, the thread identifier allows a *fast abort* in the *single-reader* case, whereas the *multiple-reader* case requires a *slow abort* (i.e., all log-walk to identify the conflicting transactions).

If a transaction evicts a transactional block then the transaction cannot perform a fast commit, so that a slow commit or an abort ensures that the state in the L2 and/or memory is cleared properly. When the same transactionally-read block (i.e., *R*, *R'*, and R^+) is evicted and accessed again multiple times, a new token is acquired and the block's read-sharer count increases. Each such new token is logged so it can be released at an abort or commit.

3.4 Handling OS interactions

The remaining issues are OS interactions such as page migration, context switch, and thread migration in the middle of a transaction. Because the transactional state can be flushed all the way to memory and even to disk, page migration simply moves the page along with the transactional state. The key issue with context switch is that conflicts between the switched-in and switched-out threads should not go undetected. While evicted tokens or fused tokens (*R'*, *W'*, R^+) are accompanied by the identifier or count which detect conflicts, in-cache tokens (*R* and *W*) are not. Therefore, the *R* and *W* bits of the switched-in and switched-out threads cannot be distinguished. To address this issue, upon a context switch, TokenTM flash-ORs the *R* and *W* bits to the *R'* and *W'* bits, and sets the thread identifier. While *W* and *W'* cannot co-exist due to the implied conflict, TokenTM exploits R^+ to ensure that *R* and *R'* are not both set. Thus, there is no flushing of the switched-out transaction's blocks (i.e., constant-time context switch). Finally, thread migration (preceded by a context switch) can occur without any problems. The blocks that are already accessed by the migrated thread are identified by the thread identifier which is preserved through the context switch, and access to other blocks require new tokens as usual.

3.5 State overhead

While comprehensive in supporting all the desired features, TokenTM incurs state overhead at all the levels of the memory hierarchy (19 bits per block in L1, and 16 bits in L2 and memory). As mentioned in Section 1, TokenTM and other TMs [5] advocate stealing some of the ECC bits in memory to hold the transactional state which can then be retrieved in one access with the data. The idea is that while SECDED for 64 bits requires 8 bits, SECDED for 256 bits requires only 10 bits, thus sparing 22 bits for SECDED-protected transactional state. However, stealing as many as 16 bits weakens error protection (e.g., 16 bits are 25% of the 64

Table 2: TokenTM vs. LiteTM

	TokenTM vs. LiteTM	Missing information	LiteTM's compensation	Performance impact on LiteTM
Private L1	*R*, *W*, *R'*, *W'*, R^+, and 14 bits of count/id (19 bits) vs. *T*, *T'*	R, W not separate	Approximate *W* by Modified and *T* (in hardware)	Extra false conflicts — extra *fast-aborts* (conflict on L1-resident block); *slow-aborts* (conflict on evicted block)
		No thread id	*Self/all log-walk* for potential conflict — hit or miss to *T'* (in software)	*self log-walk* overhead (no conflict) — *slow-commit* in both TMs; *all log-walk* overhead (conflict) — *fast-aborts* in TokenTM become *slow-aborts*
		No read-sharer count	Abort all but one reader for multiple-reader conflict (in software)	Extra aborts — extra *fast-aborts*
Shared L2 and memory	2 state bits and 14 bits of count/id vs. 2 state bits	No thread id	*Self/all log-walk* for potential conflict on evicted *single-reader* or *writer* block (in software)	*Self log-walk* overhead (no conflict) — *slow-commit* in both TMs; *all log-walk* overhead (conflict) — *fast-aborts* in TokenTM become *slow-aborts*
		No read-sharer count	*Lazy clearing all log-walk* for a write's potential conflict on evicted *multiple-reader* block (in software)	*All log-walk* overhead — *fast- or slow-commit* depending on evictions in both TMs (no conflict); *slow-aborts* in both TMs (conflict)

SECDED bits per 64-byte blocks). Also, 16 bits per block is a significant overhead in absolute terms, equivalent to nearly doubling the tag array in a system with 40-bit physical addresses and 32-KB L1 and 8-MB L2.

4 LiteTM

Recall from Section 1 that we propose *LiteTM* to reduce TokenTM's state overhead based on the key observation that read-sharer counts and thread identifiers are not needed for conflict detection. Even to identify conflicting transactions, the state is not needed in a majority of cases (i.e., when the transactional state has not been evicted from L1). As such, we completely eliminate the counts and identifiers from the entire memory hierarchy. LiteTM decouples key parts of conflict handling for L1-evicted blocks; conflict detection is still in hardware but the conflicting transactions are identified in software using transactional logs. LiteTM employs only two state bits per block in L1, L2, and main memory, which are adequate for conflict detection in hardware. Because conflict detection is in hardware for all accesses, LiteTM provides strong atomicity.

To eliminate the read-sharer count, we observe that transactional read-shared blocks that both are evicted from multiple sharers' L1s *and* are involved in conflicts, which are detected by the count, are rare. To eliminate the thread identifier, which exists only in single-sharer cases (multi-sharer cases, instead, use the count), we observe that both uses of the identifier — to identify conflicting transactions involved in a conflict on an L1-evicted block and to allow a transaction to access its own blocks — are uncommon. The first use is uncommon because most conflicts occur for in-L1-cache blocks where the conflicting transactions are trivially identified (by the caches involved in the conflict). The second use is uncommon because repeated misses to the same block are rare within a transaction (i.e., locality holds). Finally, we replace TokenTM's *R*, *W*, *R'*, *W'*, and R^+ bits in the L1 with only *T* and *T'* bits by observing that we can leverage coherence for implicitly differentiating between reads and writes in the common case, without explicitly using *R* and *W*. We employ the novel idea that *conservatively but closely* approximates the *W* bit by combining L1 'Modified' state and the *T* bit.

In the rest of this section, we explain how LiteTM uses software to handle the uncommon cases, for which TokenTM uses separate read and write bits, the count, and the identifier. Table 2 shows a high-level summary of these differences. As mentioned in Section 1, because TokenTM's state bits are fundamental to guaranteeing transactional semantics, naively shrinking TokenTM's state to fewer bits would violate correctness. LiteTM carefully compensates for the lost information.

4.1 Modifications to transactional state bits: T (transactional) bit in L1 and two bits in L2, memory

While TokenTM uses *R* and *W* bits in L1, LiteTM merges read and write into a single *T* bit (transactional bit) (first row under LiteTM column in Table 1). As in TokenTM, a transactional read or write hit locally sets the *T* bit. If multiple readers concurrently get cache hits, then multiple *T* bits are set, as are multiple *R* bits in TokenTM. To detect conflicts, we need to infer that a given block was transactionally written using a single *T* bit and no *W* bit. To that end, we approximate *W* by considering modified blocks with the *T* bit set to be transactionally written. Thus, any request to the modified block gets a reply with the modified state and the *T* bit as part of the payload, allowing the requestor to detect the conflict and incur a fault.

Because Modified and *T* combination is a superset of *W*, the *W-approximation* does not miss any real conflicts. However, false conflicts are possible but only in the following case: A block is non-transactionally modified, or transactionally modified and committed. Then a new transaction on the same core reads the block which becomes modified and transactional (i.e., approximated as a transactional write). Finally, a remote transaction reads the block, resulting in the abort. However, if the remote read occurs *before* the local read then there is no abort because the block would not be transactional at the time of the remote read. With even three or more read sharers, the chances of the local read occurring first and causing the false abort are low. Also, if there is no read sharing then there are no false aborts. Therefore, such false aborts are rare in general. We could have avoided the false aborts by confirming the conflict via a log-walk of the writer transaction. However, such log-walks are pure overhead for true conflicts, which are more common than false conflicts. Therefore, we do not perform this log-walk (first row in Table 2). Note that W-approximation does

not impact cache hits in any way (i.e., transactional write hits to modified blocks with or without *T* bit set proceed as in TokenTM).

W-approximation is recorded in the transactional state, called *writer-state*, in the L2 (and memory) whenever a modified L1 block with the T bit set is evicted or is fused upon a conflict. LiteTM's *writer-state* is similar to that of TokenTM's though TokenTM's state is exact. We explain fusion details next and eviction details in Section 4.4.

4.2 Modifications to Fusion: *T'* and log-walks

To avoid coherence changes to handle conflicts, LiteTM employs fusion but with some modifications. First, reads-followed-by-a-write conflicts in LiteTM (writer and readers on different cores) fuse the readers' tokens at the writer, as in TokenTM. However, LiteTM does not have read-sharer counts in the L1 and there is only a single *T'* bit to track exactly one reader's token (second and third rows in Table 1). Consequently, *all but one* reader are *always* aborted in this type of conflict so that either the writer or exactly one reader survives (third row in Table 2). Because conflicts involving multiple read-sharers are not common, LiteTM's extra aborts over TokenTM do not degrade LiteTM's performance by much.

Second, in write-followed-by-read conflicts, the usual modified-block writeback is accompanied with the *T* bit in the payload, allowing the transactional state in the L2 (and memory) to go to the *writer-state.* Just as TokenTM sets the *W'* bit in the reader's block, LiteTM sets the *T'* bit (second row in Table 1).

Accesses or miss requests to blocks with the *T'* bit set can neither differentiate whether the *T'* bit is from a read or a write, nor identify the transaction whose *T* bit was converted into the *T'* bit given that LiteTM does not have thread identifiers. Consequently, such an access raises a potential-conflict exception so that the exception handler performs log-walks of *all* the current transactions to determine whether there is a conflict. We discuss evictions of *T'* blocks in Section 4.4.

To avoid unnecessary all log-walks when the block is already in the accessor's read or write set (as appropriate), the accessor first performs a lower-overhead *self log-walk* of its own log and triggers an all log-walk only if the block is not in the accessor's read or write set (second row in Table 2). The log-walks are optimized to look up only the read set or the write set where appropriate (e.g., only the write set for self log-walk by a write) and to scan the log starting from the end to find the block sooner due to locality. Fortunately, such all log-walks are not frequent as they correspond to read-shared access to a previously-conflicted block, and hence do not degrade performance much.

Finally, there are some subtle details about log-walks. While one thread performs an all log-walk due to a conflict, other threads can continue concurrently and update their logs. Because all conflicts result in a fault and perform a retry, there is no risk of permanently missing a conflict. A new access, *N,* that conflicts with the faulting access, *F,* and intervenes or races with *F*'s log walk may be missed temporarily by the log walk. However, *N* would take coherence permissions away from *F* which upon retry, would coherence miss, fault again, do a log walk, and catch the missed conflict. To avoid unlikely, indefinitely-repeated retries, we stop all other threads after a fixed threshold on the number of retries (this condition did not occur in our runs).

4.3 Modifications to Commits and Aborts: Log-walks

As in TokenTM, transactions that do not encounter any conflicts (hence, their tokens have not moved) undergo fast commits in LiteTM, whereas transactions that survive conflicts undergo slow commits. In the case of aborts, because there is no thread identifier in LiteTM, only in-cache conflicts on *T* blocks can identify the conflicting transactions and undergo fast aborts (i.e., no all log-walks). All conflicts on evicted blocks require all log-walks, as do in-cache conflicts on *T'* blocks, as discussed before.

4.4 Modifications to handling evictions: Lazy clearing

One key difference from TokenTM pertains to token release for L1-evicted blocks. Tokens in L1-evicted blocks are fused into the L2 which may spill into memory, as done in TokenTM. LiteTM's L2 and memory have two state bits per block to record the following states which are similar to TokenTM's states (Section 3.3) as well as the directory states in [4]: *idle*, *single-reader*, *writer*, and *multiple-readers* ("Shared L2" rows in Table 1). A clean, *T* or *T'* block that is evicted starts in the *single-reader* state in L2 and goes to the *multiple-reader* state if more such copies are evicted. If a modified, *T* block is evicted, the block enters the *writer-state* in L2. A modified, *T'* block cannot exist due to the implied conflict.

Because there is no identifier, a transaction cannot recognize its own evicted block in the *single-reader* or *writer* states and must perform a *self log-walk*. As discussed in Section 4.2, if the self log-walk determines that the appropriate token is not already held, then an all log-walk follows to identify the conflicting transactions (LiteTM in second last row in Table 2). Because transactions do not miss repeatedly on their own blocks, these log-walks are uncommon.

TokenTM combines idle and multiple readers cases into one state and uses the read-sharer count to separate the cases. In contrast, because LiteTM does not have the count, LiteTM's *idle* and *multiple-readers* must be separate states. Because *single-reader* and *writer-state* imply only one sharer that has evicted the corresponding block, any commit or abort that tries to clear these states must be from that sharer and hence can proceed. In the case of *multiple-reader* state, however, only the last sharer's commit or abort can clear the states. But because there is no count in LiteTM, the last sharer cannot be distinguished from the rest. Consequently, *all* clearings of the *multiple-reader* state are ignored by the hardware, causing the read sharers to leave behind this state.

When a conflicting access occurs due to this left-behind state, LiteTM employs *lazy clearing* which performs an all log-walk to determine whether the previous sharers still exist. If so, there is a true conflict requiring an abort, and if not, the state is cleared allowing the access to proceed (last row in Table 2). Note that if there is a non-conflicting access (i.e., a read) to the left-behind *multiple-reader* state, the access can proceed without any lazy clearing or log-walks. Fortunately, conflicts with multiple L1-evicted readers, and hence lazy clearings, are uncommon.

There are two correctness issues: a minor one with self log-walks and a major one with lazy clearing. The issue with self log-walks involves the retry semantics of faults. Self log-walks may evict the block for which the log-walk was triggered. Such evictions would cause another self log-walk upon retry, potentially resulting in a livelock. We resolve this issue by touching the block at the end of the self log-walk to bring the block into L1. In rare

cases, a concurrent conflicting access may steal the block's coherence permissions away between the touch and retry, in which case the retry would fault. Repeated occurrences of such stealing is possible, though extremely rare, and would be caught by our retry threshold.

The major correctness issue with lazy clearing involves a race. It is possible that *after* the all log-walk checks a transaction's log and does not find the block in the read set, but *before* the state is cleared, the transaction may read miss on the block and proceed with the read as the state is *multiple-reader*. Then, the state clearing would be incorrect. This issue does not arise in TokenTM because TokenTM's log-walks decrement the read-sharer count while new readers increment the count. Increments and decrements are commutative, unlike setting and clearing, so that a reader's increment can come before or after a log-walk's decrement without making the count incorrect.

One option is to stop all other threads during such a lazy clearing but this option would be slow. Another option is to use an extra state bit per block in L2 and memory so that a lazy clearing starts by changing to *busy* state. Access to a *busy* block triggers a potential-conflict exception whose service is serialized after the lazy-clearing. Because this option increases the state overhead from 2 bits to 3 bits per block, we explore a third option by observing that only a few blocks undergo lazy-clearing at any given moment (e.g., 3-4). Therefore, we employ a few buffers, called *busy buffers*, in the L2 and memory controller to hold the blocks' addresses. The buffers are common to all the threads of a process. Lazy clearing starts by placing the block address in the buffer and removing the address upon exit. Misses that address-match on a buffer trigger potential-conflict exceptions which are serialized after the lazy clearing in software (without any hardware stalling). For the rare case of exceeding the number of buffers, we employ a single counter for a process, called *busy counter*, to track the excess lazy clearings. Any miss that encounters a non-zero *busy counter* triggers a potential-conflict exception irrespective of address-match on a buffer, and is serialized after all the current lazy clearings.

4.5 Modifications to handling OS Interactions

Any context switch in the middle of a lazy clearing, though rare, must preserve the busy buffers and busy counter for correct operation upon resumption. Thus, the buffers and counter are part of the process state. To reduce the amount of the process state, only the sum of the busy counter and the number of non-empty busy buffers is saved. Upon resumption, the sum is loaded into the busy counter. Thus, in this rare case, all misses trigger potential-conflict exception until all the resumed lazy clearings are complete and the *busy counter* goes to zero. Note that because of the replay semantics of conflicting accesses as discussed in Section 4.2, there is no risk of missed conflicts while a transaction is switched out.

LiteTM handles mid-transaction page migration similar to TokenTM but deals with mid-transaction thread migration differently. While TokenTM leverages R^+ to guarantee that R and R' are not both set (Section 3.4), LiteTM does have an R^+-equivalent to give the same guarantee for T and T'. Therefore, the switched-out thread performs a self log-walk in LiteTM and flushes the transactional blocks to memory, so that conflicts with the switched-in thread are detected correctly. Because context switches are rare, such flushing being slow may not be a concern.

4.6 Multithreaded hardware support

LiteTM can support multithreaded cores by replicating the T bits per hardware thread context while continuing to keep a single T' bit per block. Each transaction can infer the existence of other transactional accesses via T' (remote) or T (another local context). In TokenTM's case, though it has thread identifiers, the identifiers are used for tracking transactional accesses solely from remote cores and not from local contexts. As such, even TokenTM has to replicate its *R/W* bits for each context. As in LiteTM, TokenTM need not replicate R', W', and the attribute bits for each context.

4.7 LiteTM's generality

LiteTM decouples key parts of conflict handling for L1-evicted blocks; conflict detection is in hardware but the conflicting transactions are identified in software using transactional logs. This decoupling is fundamental and can be applied to other unbounded HTMs using per-block state. For instance, LiteTM can eliminate OneTM-concurrent's thread identifiers [6]), and VTM's identifiers (i.e., pointers to XSW in XADT) and implicit counts (i.e., number of entries in XADT) [24].

4.8 State overhead

LiteTM needs only two state bits per block in L1, L2, and main memory, while TokenTM needs at least 16 bits. Assuming that the ECC memory bits are stolen to hold transactional state, this overhead corresponds to only 3.3% of the 64 SECDED bits per 64-byte block compared to TokenTM's 25%. Also, LiteTM's two bits add about 12% to the tag entries compared to TokenTM's overhead of nearly 100%.

To reduce the state overhead even beyond LiteTM, we experimented with a LiteTM variant that has one state bit per L1 block in L2 and memory. Because the bit cannot distinguish between transactional reads and writes, and single and multiple readers, this variant employs self and all log-walks to make these distinctions. The bit is lazy-cleared, if possible, upon a potential conflict. We show this variant's performance in Section 6.1. Another variant is to have R, W, R', W' in L1 (and two state bits per block in L2 and memory) which would reduce L1 overhead from 19 bits to 4 bits, as compared to TokenTM. This variant may be acceptable because L1 is a custom structure unlike main memory. However, because W-approximation works well in practice, this variant does not offer any major advantages over the original LiteTM. We emphasize that the bulk of the state reduction comes from removing the thread identifier and sharer count, and not from collapsing R and W into T.

5 Methodology

To evaluate our ideas, we implement LiteTM in the Wisconsin GEMS HTM simulator [21] which uses Simics [20] to perform full-system simulations. We simulate a SPARC-based multicore running Solaris 10. Table 3 summarizes the parameters of the simulated system. Using GEMS's user-level exception handlers, we faithfully capture all the cases requiring self log-walks, all log-walks, lazy clearing, slow-commits, fast-aborts, and slow-aborts.

We use STAMP with the smallest input dataset [9] (many benchmarks do not scale beyond 4 cores with larger datasets, which also slow down simulations). Table 4 characterizes the

Table 3: Hardware parameters

Processors	8, 1 GHz, in-order issue
Private L1	32K, 4-way, 64 byte blocks, 1-cycle latency
Shared L2	8M, 8-way, 64 byte blocks, 34-cycle latency
Memory	8 GB, 448-cycle latency
Coherence	Directory MOESI with full bit-vector sharer list
TokenTM	n state bits + 14 bits of thread id/sharer count per block in L1 (n = 5) and in L2 and memory (n = 2)
LiteTM	2 state bits per L1 block in L1, L2, memory + 4 busy buffers + 1 busy counter

benchmarks showing the fraction of TokenTM's execution time spent in transactions (*%xact time*), the length of transactions expressed quantitatively as the number of instructions per transaction (*#instrs/xact*) and qualitatively (*xact length*), and the amount of contention expressed quantitatively as the ratio of number of aborts to number of commits in TokenTM (*#aborts/#commits*) and qualitatively (*contention*). These data mostly match the STAMP paper [9] and show that STAMP covers a wide spectrum of transactional behavior (i.e., evictions, read-sharing instances, and conflicts), even for the small dataset, providing confidence in the generality of our results. Because we compare LiteTM against TokenTM, we validate TokenTM's performance obtained by us against the TokenTM paper [7]. In the column *TokenTM vs. LogTM-SE*, we show TokenTM's performance normalized to that of LogTM-SE with 2K-bit Bloom filters (2Hx3 in [7]). On average, TokenTM performs 42% better than LogTM-SE (not shown). The numbers agree with the TokenTM paper [7] and show that Bloom filter saturation (Section 2) hurts performance, justifying TokenTM's "full-map" approach of per-block transactional state without hashing. In the last column (*TokenTM vs. Single*), we show TokenTM's speedups on 8 cores over single-thread runs to confirm that TokenTM scales well over at least a modest number of cores.

The benchmarks with long transactions and high contention (e.g., *yada*, *bayes,* and *labyrinth*) perform better if the oldest conflicting transaction is chosen to survive the abort, as proposed in [8] to alleviate the problem of "the starving elder". For uniformity, we apply this policy to all the benchmarks. In LiteTM, we apply this policy also to abort of all but one L1-resident read-sharer (Section 4.2). To account for statistical variations (e.g., the randomized back-off delay for a transaction relaunch after an abort for reducing repeated conflicts [8]), we use enough randomly-perturbed runs to achieve 95% confidence [2].

6 Experimental Results

LiteTM eliminates TokenTM's read-sharer count and the thread identifier to reduce the state overhead from 16 bits per L1 block to 2 state bits. To compensate for this missing information, LiteTM perform parts of conflict detection in software for L1-evicted blocks. That is, conflict detection is still in hardware using the state bits but the conflicting transactions are identified in software by walking transactional logs. In doing so, LiteTM incurs some performance overhead. Therefore, we begin with a performance comparison of LiteTM against TokenTM. We then explain the performance numbers by quantifying the number of self and all log-walks and their extra work, and by providing breakdowns of fast and slow commits and aborts.

Table 4: Benchmarks

Benchmark	%xact time	#instrs / xact	xact length	#aborts / #commits	contention	TokenTM vs. LogTM-SE	TokenTM vs. Single
ssca2	13	13	short	0.01	low	1.01	5.6
k-means-low	7	106	short	0.03	low	0.99	3.5
k-means-high	11	106	short	0.07	low	1.02	3.1
intruder	56	187	short	1.25	med	1.11	2.6
genome	61	1209	med.	0.11	low	1.05	4.0
vacation-low	92	1640	med.	2.78	high	1.38	4.4
vacation-high	92	2218	med	2.56	high	1.46	4.1
yada	97	5715	long	1.17	med	2.36	2.5
bayes	91	39213	long	1.82	high	2.19	2.7
labyrinth	99	147515	long	3.86	high	2.72	2.0

6.1 LiteTM performance

In Figure 2, we compare LiteTM's performance to that of TokenTM. Both the LiteTM 1-bit variant (Section 4.8) and HTM-STM hybrids have only one bit per L1 block in L2 and memory which is half the state overhead of LiteTM. (Hybrids need one bit to ensure isolation among hardware and software transactions, and among transactions and non-transactions (i.e., strong atomicity) [5].) Therefore, we include these two TMs in this comparison with one twist: Because we do not have access to a fully-optimized hybrid, we use a hybrid variant which provides an upper bound on hybrid's performance. In hybrids, transactions switch from HTM to STM when a transactional block is evicted. In our variant, hardware transactions use TokenTM, whereas software transaction perform a single extra memory write to a globally-shared hash table whenever a new token is obtained (i.e., the first transactional access to a block). We do not impose any other overheads on software transactions upon commit or abort (e.g., to clear the hash table). Because software transactions perform at least one write to a globally-shared hash table to update read sets and write sets for conflict detection (in reality, STMs incur more memory accesses to update other structures such as per-transaction private read/write set and undo log [1]), our variant provides an upper bound on hybrids' performance.

Figure 2 shows the performance of LiteTM, LiteTM 1-bit variant, and hybrid upper-bound variant normalized to that of TokenTM. The X axis shows the benchmarks in the order of primarily increasing transaction length and secondarily increasing contention (Table 4). This ordering clearly shows trends across benchmarks. LiteTM incurs 4% average degradation over TokenTM with the worst degradation of about 10% for *bayes*. LiteTM-1-bit incurs significantly higher average (24%) and worst-case (72%) degradation than LiteTM. LiteTM-1-bit requires numerous self and all log-walks to distinguish between transac-

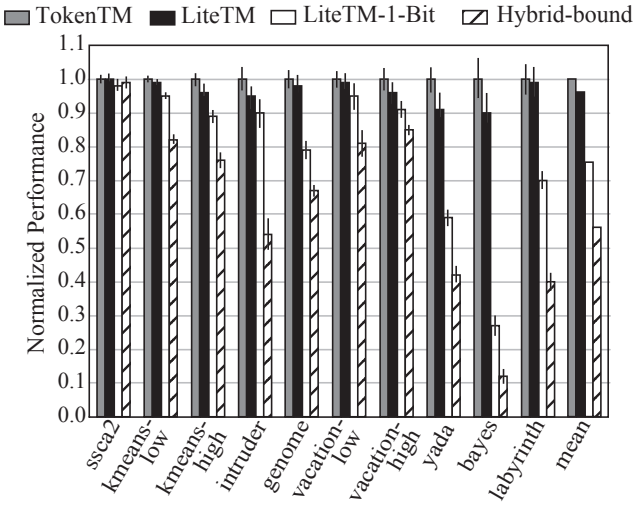

FIGURE 2: LiteTM Performance

tional reads and writes, and single and multiple readers, whereas LiteTM uses its two state bits to make the distinction (Section 4.4). Hybrid-bound incurs 44% average degradation over LiteTM. This degradation comes from hybrids' software conflict detection on *every* access of software transactions whereas LiteTM incurs log-walks only for L1-evicted blocks, as mentioned in Section 1. Given that both LiteTM-1-bit and hybrids need one state bit per block, these results justify the use of two state bits in LiteTM. Combining TokenTM's speedups over LogTM-SE from Table 4 and LiteTM's slowdowns over TokenTM, LiteTM performs 36%, on average, better than LogTM-SE, maintaining TokenTM's performance advantage over LogTM-SE.

Finally, LiteTM's degradation mostly increases as transaction length and contention increase across benchmarks (from left to right). Longer transactions and higher contention result in more evictions and conflicts which require the use of thread identifier and read-sharer count triggering more log walks and hence incurring higher performance degradation in LiteTM. The main outlier is *labyrinth* which degrades less than *yada* and *bayes* despite having higher contention and longer transactions. Because of *labyrinth's* extremely long transactions and high contention (Table 4), TM's optimistic execution incurs significant overhead — aborts and back-off time for relaunch after aborts (Section 5). We tried turning off back-off which led to starvation due to excessive aborts. Compared to this high overhead, LiteTM's additional overhead is relatively small, resulting in less degradation for *labyrinth* as compared to those for *yada* and *bayes*. The trend of increasing degradation from left to right also holds, albeit with more outliers, for LiteTM-1-bit and hybrid-upper-bound.

We ran LiteTM on 16 cores with a slight implementation variant of the W-approximation. The degradations over TokenTM were statistically similar to those of the 8-core runs (not shown). We also experimented with adding *R*, *W*, *R'*, and *W'* bits to LiteTM and saw less than 5% improvement in performance.

6.2 LiteTM performance analysis

The key explanation for LiteTM performing close to TokenTM is as follows. LiteTM's self and all log-walks with lazy clearing compensate for the loss of information in terms of separation of reads and writes, read-sharer count, and thread identifier (Table 2). LiteTM's log-walks not only increase the amount of compute work, but also (1) replace transactional data with log data causing more slow-commits (commit after eviction) and slow-aborts (conflict on block evicted by multiple read sharers); and (2) stretch transactions' life times and induce more conflicts (i.e., log-walks delay token release keeping blocks in transactional state longer). However, as stated before, the cases requiring the information and, consequently, the log-walks are uncommon. Hence, the log-walks' performance overhead is low. Accordingly, we quantify how often these pieces of information are needed and the number of log-walks in Section 6.2.1, and the number of extra slow commits and aborts in Section 6.2.2.

6.2.1 Log-walks

Recall that (1) TokenTM uses the thread identifier to recognize a transaction's own blocks that either are fused in another cache (Section 3.1) or are evicted from L1 (Section 3.3), and to identify the transactions involved in conflicts on evicted blocks (Section 3.3); (2) TokenTM uses the read-sharer count to determine conflicts on read-shared blocks (Section 3.3); and (3) LiteTM approximates *W* as a combination of modified state and *T* (Section 4.1). Table 5 quantifies how often (1) these uses occur which require self and all log-walks with lazy clearing, and (2) W-approximation is inaccurate and induces false aborts. The table shows the misses to a transaction's own blocks as a percent of all transactional misses (*%miss to own block*), the out-of-cache aborts due to conflicts on evicted blocks as a percent of all aborts (*%out-of-cache abort*), the conflicts on read-shared L2 blocks which have been evicted from more than one sharer's L1 — true conflicts and false conflicts requiring lazy clearing — as percent of all conflicts (*%conflict on L2 rd-shared*), and the false aborts due to W-approximation as percent of true aborts (*%false abort*).

We see that *% miss to own block* and *% out-of-cache aborts*, which together correspond to use of thread identifier in TokenTM, are low, confirming that repeated misses to the same blocks and conflicts on L1-evicted blocks are rare. *% conflicts on L2 rd-shared block*, which corresponds to the use of read-sharer count in TokenTM, is low, confirming that conflicts involving multiple L1-evicted read-shared blocks are rare. For the most part, only the benchmarks with higher contention and longer transactions — *yada*, *bayes,* and *labyrinth* (Table 4) — have significant quantities, as expected. LiteTM replaces the identifier and count, respectively, with self log-walks and all log-walks with lazy clearing, which, consequently, are infrequent. We also see that *% false aborts* is low implying that the W-approximation is rarely inaccurate. The number of self and all log-walks per committed transaction (*#self + all log-walk / #commit*) reconfirm that the self and all log-walks are mostly infrequent with the self log-walks occurring more often than the all log-walks. *labyrinth* with its numerous self log-walks is an exception. However, *labyrinth's* log-walks do not degrade performance as they are amortized over the extremely long transactions (Table 4). Across the benchmarks, however, each instance of the log-walks scan many addresses as shown by the number of addresses per self log-walk and all log-walk, respectively, in the last two columns in Table 5 (*self log length* and *all log length*). The high log-walk volume, especially of the more-often-occurring self log-walks, increases the compute work, degrading LiteTM's performance. The volume also results in evictions inducing extra

Table 5: Log-walks

Bench-marks	% miss to own block	% out-of-cache abort	% conflict on L2 rd-shared	% false abort	#self + all log-walk / #commit	self log length	all log length
ssca2	0.04	0	0	0	~0	2	0
k-means-low	0.16	0	0	0	~0	3.3	0
k-means-high	0.24	0	0	0	~0	4	0
intruder	0.4	0.04	0.04	0	~0	22	34
genome	0.39	0	0.65	2.5	0.02 + ~0	17	27
vacation-low	0.01	0	0.36	0	~0	28	36
vacation-high	0.05	0.03	0.40	0	0.02 + 0.01	38	54
yada	2.3	0.8	0.5	0.9	0.3 + ~0	97	102
bayes	6	1.1	1.9	0.3	3.9 + 0.08	162	327
labyrinth	15	5.6	2.5	0.1	58 + 0.94	272	1408

slow-commits and slow-aborts, and stretches transactions' life times inducing more aborts. We analyze these effects next.

6.2.2 Commits and aborts

Table 6 compares TokenTM and LiteTM in terms of the ratio of number of all aborts to number of all commits (*#aborts / #commits*), the percent of slow commits over all commits (*%slow-commits*), and the percent of slow aborts over all aborts (*%slow-aborts*). Comparing the abort-to-commit ratios in TokenTM and LiteTM, the benchmarks with relatively more log-walks (*#self + all log-walk / #commit* in Table 5) — *bayes* and *labyrinth* — have more aborts (the two TMs have the same number of commits). The increase in aborts is due to the stretching of the transactions' lifetimes. *yada* is an exception with many log-walks but fewer aborts. Instead of more aborts, *yada* incurs more back-off delay for relaunch after abort (Section 5). *k-means low*, *vacation low,* and *vacation high* are exceptions in the opposite direction: relatively few log-walks but many more aborts. The increase in the percent of aborts appears to be large for *k-means low* because the absolute number of aborts is small, as confirmed by the small performance degradation (Figure 2). In *vacation low* and *vacation high*, the extra aborts are due not to the log-walks but include the aborts of all-but-one L1-resident read-sharer involved in a conflict (Section 4.2). These all-but-one aborts serialize the read-sharer inducing even more conflicts in these high-contention benchmarks (*#aborts/#commits* in Table 4).

Comparing the percent of slow-commits in TokenTM and LiteTM, the benchmarks with relatively more log-walks — *yada*, *bayes,* and *labyrinth* (Table 5) — evict many transactional blocks leading to slow commits which contribute to their performance

Table 6: Commits and aborts

Benchmark	TokenTM			LiteTM		
	#aborts / #commits	%slow-commits	%slow-	#aborts / #commits	%slow-commits	%slow-aborts
ssca2	0.01	0.4	0	0.01	0.5	0
k-means low	0.03	2.1	0	0.04	2.7	0
k-means high	0.07	2.87	0	0.08	2.9	0
intruder	1.3	31	0	1.4	33	0.04
genome	0.1	3.1	0	0.1	9.2	0
vacation low	2.8	38	0	3.1	54	0
vacation high	2.6	33	0	2.9	56	0.03
yada	1.2	15	~0	1.1	30	0.8
bayes	1.8	17	~0	2.7	38	1.1
labyrinth	3.9	26	0	5.3	95	5.6

degradation (Figure 2). Though *labyrinth* has the largest increase in slow commits, its basic optimistic execution overhead, as discussed in Section 6.1, dwarfs its slow-commit overhead. *vacation-low* and *vacation-high* are exceptions which have more slow commits but relatively few log-walks (*#self+all log-walk / #commit* in Table 5). These two benchmarks' extra aborts (discussed above) imply more conflicts which lead to more slow-commits because transactions that survive conflicts undergo slow-commits (Section 3.2). However, because the aborts far outnumber the commits in these benchmarks (Table 6), the abort overhead dwarfs the extra slow-commit overhead so that overall performance degradation is not much (Figure 2). Finally, we see that slow-aborts are infrequent in both TokenTM and LiteTM.

6.3 Sensitivity to number of busy buffers

Recall from Section 4.4 that LiteTM uses busy buffers and a busy counter to handle races between log-walks for lazy clearing and transactional accesses. In Figure 3, we show LiteTM's performance varying the number of buffers as four (same as Figure 2, see Table 3) and zero normalized to that of TokenTM. With no buffers, a non-zero busy counter implies that a lazy clearing is underway and flags *all* misses during a lazy clearing to be serialized after the lazy clearing, under the conservative assumption that the misses are to the block being lazy-cleared.

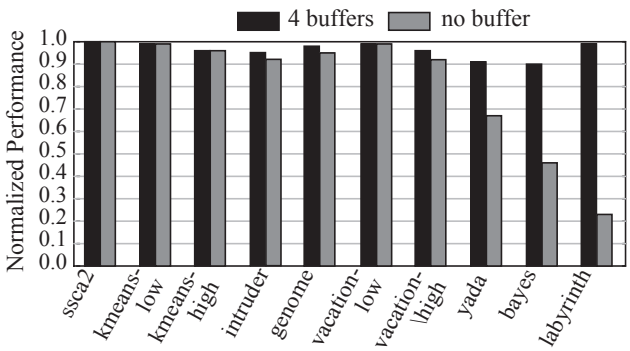

FIGURE 3: Sensitivity to number of busy buffers

Because of the severely-conservative assumption, the no-buffer configuration incurs more performance loss for the benchmarks with longer transactions and higher contention which have more lazy clearing than the other benchmarks (*%conflict on L2 rd-shared* in Table 5). Nevertheless, the number of lazy clearings per transaction is low, even in the high-contention benchmarks (*#self + all log-walk / #commit* in Table 5). Consequently, not many lazy clearings occur in parallel (we found at most 3 in *labyrinth*) requiring only a few busy buffers. However, because each lazy clearing all log-walk is long, the no-buffer configuration holds up all misses for these long time periods incurring significant performance loss.

7 Conclusions

To allow transactional state to exceed cache capacity, recent HTMs (e.g., TokenTM, VTM, OneTM-concurrent) employ per-block thread identifiers and sharer counts. The resulting overhead can be high, especially if the state is held in stolen ECC bits. LiteTM cuts the overhead by decoupling the detection of conflicts (done in hardware) from the identification of conflicting transactions (done in software using transactional logs, in the uncommon case). LiteTM compensates for the lost information via the following novel ideas: (1) approximating *W* for L1-resident blocks, (2) lazy clearing of transactional state in L2 and memory, (3) self log-walk to identify a transaction's own blocks. LiteTM requires just 2 bits per block in L1, L2, and memory. Experiments show that LiteTM reduces TokenTM's state overhead by about 87% while performing within 4%, on average, and 10%, in the worst case, of TokenTM. By reducing transactional state overhead while maintaining performance, LiteTM lowers the barrier for adoption of HTMs in real products.

Acknowledgments: We thank Jayaram Bobba, Kevin Moore, and the anonymous reviewers for their feedback. This work was funded, in part, by the National Science Foundation (Award No.:CCF-0644183).

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proc. of 2006 PLDI*, pages 26–37. ACM, 2006.

[2] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proc. of 9th HPCA*, page 7, 2003.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. of Eleventh HPCA*, pages 316–327. Feb 2005.

[4] J. Archibald and J. L. Baer. An economical solution to the cache coherence problem. In *Proc. of 11th ISCA*, pages 355–362, 1984.

[5] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *Proc. of 35th ISCA*. June 2008.

[6] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comp. Arch. News*, 35(2):24–34, 2007.

[7] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proc. of 35th ISCA*. Jun 2008.

[8] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proc. of 34th ISCA*, pages 81–91. 2007.

[9] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of IISWC*, September 2008.

[10] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proc. of 34th ISCA*. Jun 2007.

[11] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of 33rd ISCA*, pages 227–238. 2006.

[12] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. *SIGPLAN Not.*, 41(11):347–358, 2006.

[13] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. *SIGOPS Oper. Syst. Rev.*, 40(5):371–381, 2006.

[14] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. of 12th ASPLOS*, pages 336–346, 2006.

[15] J. Gray. The transaction concept: Virtues and limitations. In *Proc. of Seventh VLDB*, pages 144–154. Sep 1981.

[16] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of 20th ISCA*, pages 289–300. May 1993.

[18] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In *Proc. of 14th ASPLOS*, pages 145–156. 2009.

[19] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proc. of PPoPP*, Mar 2006.

[20] P. S. Magnusson et al., Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[21] M. M. K. Martin et al., Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comp. Arch. News*, 33(4):92–99, 2005.

[22] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. of the 12th HPCA*, pages 254–265. Feb 2006.

[23] M. J. Moravan et al., Supporting nested transactional memory in LogTM. *SIGPLAN Not.*, 41(11):359–370, 2006.

[24] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. of 32nd ISCA*, pages 494–505.Jun 2005.

[25] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proc. of the 40th Micro*, pages 123–133. 2007.

[26] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th PODC*, pages 204–213. Aug 1995.

[27] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *Proc. of 35th ISCA*, pages 139–150. 2008.

[28] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of 13th HPCA*. Feb 2007.