ZEBRA: A Data-Centric, Hybrid-Policy Hardware Transactional Memory Design

Rubén Titos-Gil
Universidad de Murcia, Spain
rtitos@ditec.um.es

Anurag Negi
Chalmers University of Technology, Sweden
negi@chalmers.se

Manuel E. Acacio
Universidad de Murcia, Spain
meacacio@ditec.um.es

José M. García
Universidad de Murcia, Spain
jmgarcia@ditec.um.es

Per Stenstrom
Chalmers University of Technology, Sweden
per.stenstrom@chalmers.se

ABSTRACT

Hardware Transactional Memory (HTM) systems, in prior research, have either fixed policies of conflict resolution and data versioning for the entire system or allowed a degree of flexibility at the level of transactions. Unfortunately, this results in susceptibility to pathologies, lower average performance over diverse workload characteristics or high design complexity. In this work we explore a new dimension along which flexibility in policy can be introduced. Recognizing the fact that contention is more a property of data rather than that of an atomic code block, we develop an HTM system that allows selection of versioning and conflict resolution policies at the granularity of cache lines. We discover that this neat match in granularity with that of the cache coherence protocol results in a design that is very simple and yet able to track closely or exceed the performance of the best performing policy for a given workload. It also brings together the benefits of parallel commits (inherent in traditional eager HTMs) and good optimistic concurrency without deadlock avoidance mechanisms (inherent in lazy HTMs), with little increase in complexity.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; C.1.4 [**Processor architectures**]: Parallel Architectures

General Terms

Performance, Design, Experimentation

Keywords

Hardware Transactional Memory, Contention Management

1. INTRODUCTION

Fast implementations of transactional programming constructs that provide optimistic concurrency control with stringent guarantees of atomicity and isolation are necessary for Transactional Memory (TM) to gain widespread usage. Software TM (STM) implementations impose too high an overhead and do not fare well against traditional lock based approaches when performance is important. Hardware TM (HTM) systems show much greater promise. Yet, within the design space of HTM systems, there are tradeoffs to be made among various pertinent metrics like design complexity, speed and scalability. Early work on HTM proposals [6] [19] fixed critical TM policies like versioning (how speculative updates in transactions are dealt with) and conflict resolution (how and when races between concurrent transactions are resolved). These designs choose a point in the HTM design space and analyze utilization of available concurrency in multithreaded applications within that framework.

Results in research so far do not show a clear winner or an optimal design point. Lazy HTMs, that confine speculative updates locally and run past data races until a transaction ends, do seem to be more efficient at extracting concurrency [16] but require elaborate schemes [5][13] to make race free publication of speculative updates (i.e. transaction commit) scalable. Eager HTMs, that version data in place and resolve conflicts as they occur, make such publication rather trivial at the expense of complicating behavior when speculative execution needs to be undone to avoid data races (i.e. transaction abort). Eager HTMs fit very naturally into existing scalable cache coherent architectures and can tolerate spills of speculative data into the shared memory hierarchy, unlike their lazy counterparts. When comparing the performance of the two such designs, a clear winner cannot be established. With workloads that demand high commit throughput eager systems perform substantially better, while with high contention workloads lazy designs come out on top.

This reasoning suggests that a new HTM design that selects the best performing policy (eager or lazy) depending on workload characteristics would be close to the most suitable HTM design for the scalable architectures under consideration. A key factor would then be the complexity involved in realizing such a design in hardware. Some solutions have been proposed that attempt to provide a hybrid-policy HTM design. UTCP [8] is a cache coherence protocol that allows transactions in a multithreaded application run either eagerly or lazily based on some heuristics like prior behavior of transactions. Although it lays down an interesting approach, the authors feel that the protocol is a significant departure from existing cache coherence designs and the additional complexity involved for just supporting TM represents too high a design cost. FlexTM [16] allows flexibility in policy but it does so by implementing critical

ICS'11, May 31–June 4, 2011, Tucson, Arizona, USA.

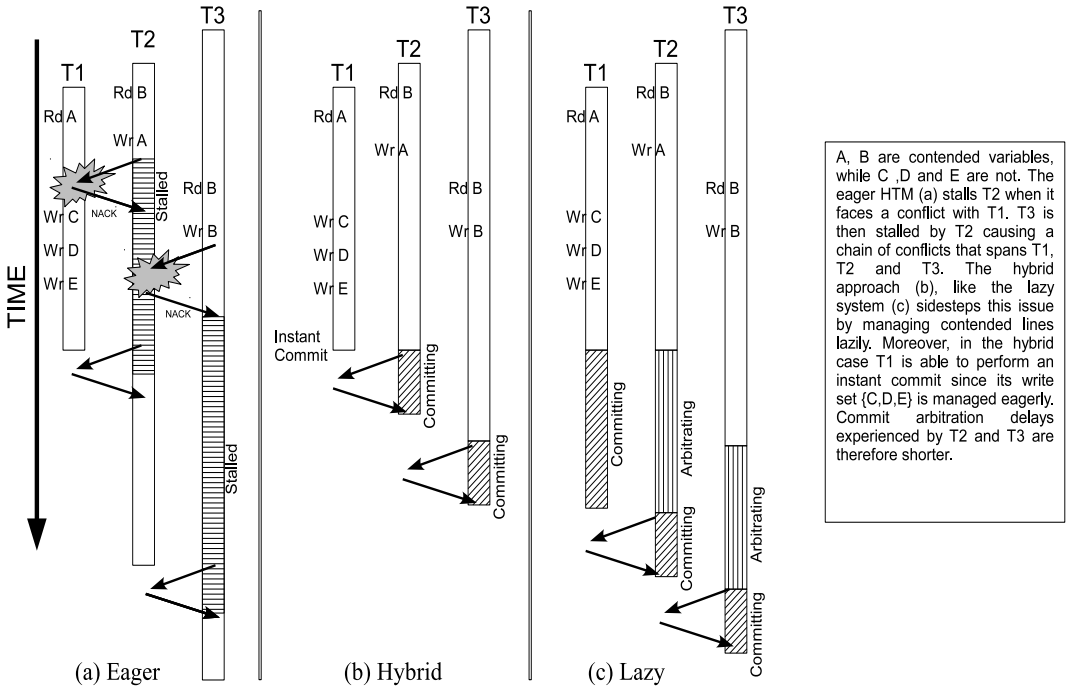


Figure 1: Behavioral differences between different HTM design points.

policy managers in software. It provides a significant improvement in speed over software TM implementations by proposing the use of Alert-On-Update (AOU) hardware, but the considerable cost of software intervention renders a comparison with pure HTMs moot. LV* [12], a proposal that utilizes snoopy coherence, allows programmer control over policy in hardware but with the constraint that all transactions in an application must use the same policy at any given time. A scalable alternative has not yet been proposed. The requirement of programmer-assisted policy change is a drawback too since the same phase of an application can exhibit different behavior with varying datasets.

In this work we propose a solution that is simple and yet powerful and flexible. We recognize the fact that assuming all data accessed in a transaction possesses the same characteristics can lead to sub-optimal solutions. Based on our study of conventional HTM design points we infer that only a relatively small fraction of data accessed inside transactions is actively contended. The rest is either thread-private (stack or thread-local memory) or not actively contended. Treating these two categories of data the same inside transactions leads to inefficiencies – a prolonged publication phase at commit when using a lazy design or increased contention leading to expensive aborts when using an eager approach. This work attempts to break this restriction by choosing a granularity for data at which minimal changes are required in existing scalable architectures – that of the cache line. Efficient scalable cache coherence implementations exist and have been extensively studied for a long time. Our design leverages these by annotating cache lines as being either contended or not. Contended lines are managed lazily thereby permitting greatest concurrency among transactions. It should be noted that eager systems disallow reader-writer concurrency while in lazy systems it can occur quite naturally if the reader commits before the writer. All non-contended lines are versioned eagerly and thus, on transaction commit, only contended lines need to be published. When contention is discovered (e.g. when aborting or stalling) the offending cache line(s) is (are) marked as contended. Over the course of execution of a workload, versioning of lines that are contended transitions from eager to lazy. In the steady state we can expect only the contended subset of the working dataset to be managed lazily. As we shall show in the analysis presented here, substantial gains over existing fixed policy HTM designs can be seen. The incremental cost of implementing this approach is minimal since only very modest behavioral changes are required in the cache coherence protocol. We call this hybrid-policy HTM protocol *ZEBRA*. An African folktale speaks of how the white zebra fell into a fire and burning sticks scorched black stripes on its flawless coat. Here, transactions manage data purely eagerly (white) to begin with but acquire lazy lines (black stripes) when they conflict (fall into a fire).

Figure 1 depicts an interleaving of three concurrent transactions and highlights some important behavioural aspects of our proposal. In the eager case (Figure 1-a), we see that although transactions *T1* and *T3* are independent, *T3* is stalled because of a chain of dependencies created via transaction *T2*. This does not occur in the hybrid-policy ZEBRA design (Figure 1-b) or the purely lazy case (Figure 1-c) and in the example shown all three transactions commit without conflicts. It should be noted here that in the hybrid case writes to A and B by T2 and T3 are managed lazily, since the lines were annotated as contended at an earlier stage of the execution. On the other hand, in the lazy case *T2*'s commit is delayed because *T1*, having a relatively large write-set, has locked resources that *T2* needs to publish its updates. This in turn delays *T3*'s commit. In the hybrid scenario *T1* is able to perform an instant commit since none of the lines in its write-set are contended and, hence, are managed eagerly, allowing *T2* and *T3* to proceed with their commit operations without any delay on account of *T1*.

There are certain other benefits that stem from using such an approach. Deadlock avoidance mechanisms are not required since contended lines are eventually managed lazily, thereby guarantee-

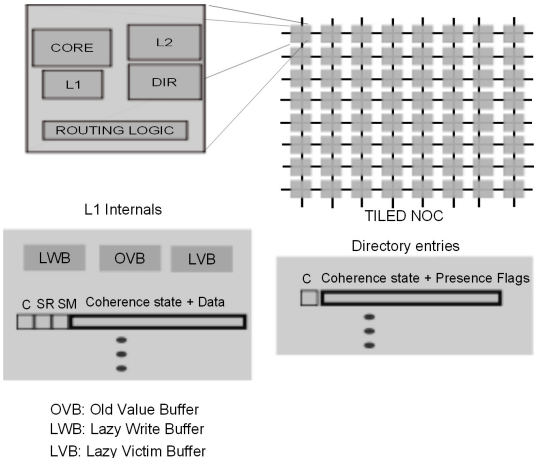


Figure 2: ZEBRA – Salient architectural features

ing forward progress. Significant reductions in transaction commit delays result in a major contraction of the window of contention for concurrent transactions. The burden on lazy versioning mechanisms is considerably reduced enabling much larger transactions to run without resorting to safety nets (like serialization via a single global lock). This effect combines synergistically with a coherence-decoupled lazy version buffer – write-write conflicts, downgrade and abort misses (defined later) can be largely eliminated, amplifying gains achieved from the central idea. Since the design does not lock policy it can adapt to changing workload conditions and is resistant to pathologies that fixed policy HTMs suffer from. The authors feel that this proposal touches upon a sweet spot in the HTM design space that offers both simplicity of design and robust performance.

The rest of this paper is organized as follows. Section 2 describes the salient architectural and behavioral features of the ZEBRA HTM protocol. Section 3 first describes the experimental methodology adopted to evaluate our approach, and then presents our results and analyses. Section 4 puts our work here in perspective of other work in HTM systems on related issues. Section 5 summarizes the paper and looks at future work that can be done to build upon the ideas presented here.

2. DESIGN AND OPERATION

2.1 Conceptual Overview

We choose a tiled CMP (chip multiprocessor) architecture where each tile comprises a processing core, a slice of a shared inclusive L2 cache and corresponding directory entries. The tiles are interconnected by a mesh-based routing network. Figure 2 shows the salient features of the architectural framework. Each processing core has private Level 1 instruction and data caches. The directory keeps private caches coherent using a MESI protocol. Two single-bit speculative access annotations are maintained at the private caches for each cache line - SR (for speculatively read lines) and SM (for speculatively modified lines). Such annotations have been used by several prior HTM proposals [6, 11] to track transactional reads and writes. Read set signatures [4] are employed to permit speculatively read lines to be evicted from private caches.

In order to track contention in the ZEBRA HTM design, we extend per-cache line metadata at the directory and at the private caches with just one additional bit – *"contended bit"* – hereafter referred to as the *C-bit*. The C-bit is transported with all coherence requests and data responses. A C-bit value of "1" indicates that the line has experienced contention in the past. The bit is reset if a line is flushed from the on-chip cache hierarchy or when a non-transactional update is seen by the directory.

The number of contended lines accessed by a transaction is usually quite small in the workloads we have experimented with. Keeping such writes away from the cache improves performance by reducing the number of *contamination misses*[18] – misses due to invalidation of speculatively updated lines on aborts – and redundant permission downgrades from exclusive or dirty state to shared state (which we term *downgrade misses*) that allow detection of conflicts. Moreover, this also mitigates the effect of false writer-writer conflicts. Therefore, we deemed it prudent to introduce a *Lazy Write Buffer (LWB)* to contain speculative updates to contended lines. This buffer is sized to be large enough to accommodate the contended fraction of the write set of a transaction in the common case. We have found that a 32-word buffer is sufficient to handle most commonly occurring cases. This buffer is drained when committing a transaction and discarded when aborting. Writes buffered in this structure do not participate in coherence until the transaction starts commit. Occasional situations when the buffer is completely filled up are handled by buffering subsequent contended-line updates in the cache. Prior to such a cache line update, non-exclusive (shared) access is acquired to the line (line-fill if not present; or downgrade to shared with write-back if dirty) in order to preserve its old value. Figure 4 shows how writes from the processor are dealt with by the private cache controller. To minimize the possibility of spilling lazy speculative state from the L1 cache we prioritize retention of such lines in the private cache and add a small (4 cache lines) *Lazy Victim Buffer (LVB)* to contain rare spills due to limited associativity. This approach works well for the workloads considered here. In the rare case of spills of contended lines, we enforce serialization.

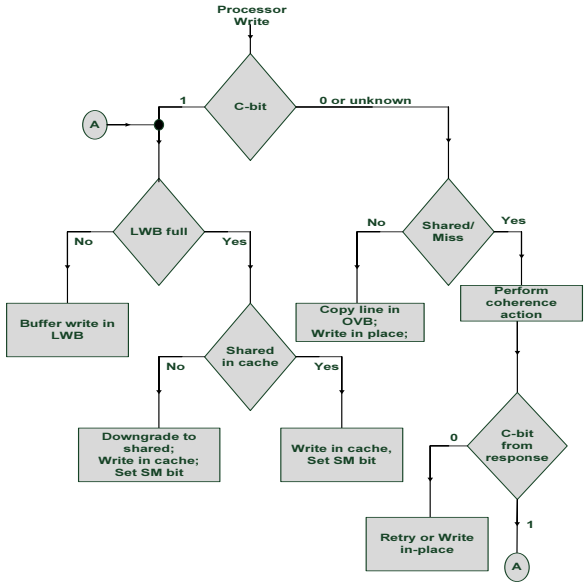


Figure 4: Write handling at the private cache.

Updates to lines that are either non-contended or have unknown C-bit status bypass this buffer (see Figure 4) . This can cause coherence requests to be issued to the directory if L1 line-fill or write permissions are required. If the result of a coherence operation indi-

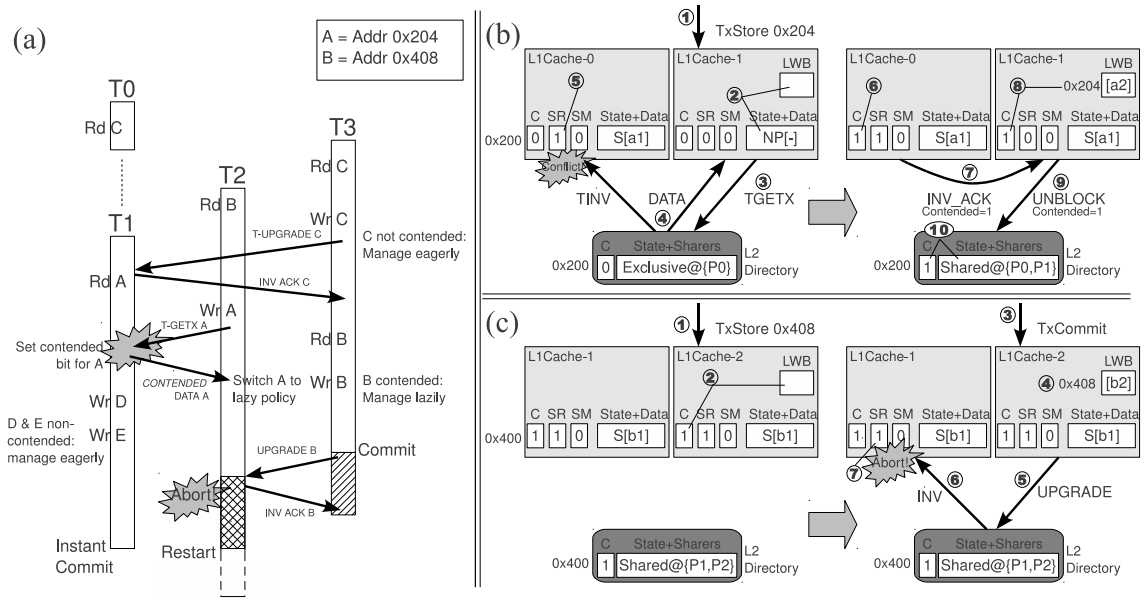


Figure 3: ZEBRA – Key protocol actions.

cates that the line is contended, the write is buffered in the LWB. In either case, the line is allocated in the cache (if not already present) and its C-bit state is updated. If the C-bit is not set, the update happens in place and the old contents of the line are recorded in an *Old Value Buffer (OVB)* or written to a thread-private log in virtual memory in case OVB capacity is exceeded. This aspect of eager behavior is similar to that of LogTM [19].

A transaction with no updates to contended lines can commit without delay, permitting true commit parallelism in such a case. If there are some lazy updates, they must be validated and made globally visible. We adopt the simplest possible approach to do so by having the committer acquire a global commit token. Our results show that in workloads where lazy conflict resolution yields best results we compete very well against or better the performance of the more sophisticated scalable commit approach adopted by STCC [5]. While a more scalable lazy commit scheme would further enhance our proposal, the design choice is orthogonal to the key ideas described in this work. All writes in the LWB are made globally visible at commit. All lines in the shared (S) state in cache with SM indicator set are upgraded to modified (M) state after the directory grants exclusive permissions to the line.

All coherence messages generated in response to speculative accesses by the core are distinguished from ordinary ones by setting a special flag in such messages. An abort occurs when any non-speculative coherence message hits a line speculatively accessed by a transaction. It should be noted that invalidations that result when lazily managed lines are committed are non-transactional. For eagerly managed lines a requester-retry policy similar to the one adopted by LogTM [19] is used. If a cyclic dependency on eager lines is detected (refer usage of possible cycle flag in [19]) at one or more transactions in the dependency chain, they abort to break the deadlock. No software intervention is required. A unique aspect of our design is that offending cache lines will henceforth be treated lazily during re-execution and, thus, will no longer have the potential to cause deadlock. This effect renders LogTM's usage of TLR-like timestamps [14] unnecessary for guaranteeing forward progress.

C-bits at the directory are set when unblock messages sent by cores to indicate completion of in-flight coherence operations indicate contention. Contention may be reported if a requester discovers a conflict with another transaction or when a committer publishes its contended lines. The directory reports this status in all subsequent coherence messages. The C-bit is cleared when a non-transactional update to the line is completed allowing memory to be recycled without the old C-bit value affecting behavior in the new usage context. In most applications it is highly unusual to find non-transactional updates to a cache line interleaved with transactional accesses. The C-bit is also cleared if a line must be evicted from the directory.

2.2 Protocol behavior

Standard directory-based MESI cache coherence is employed for detecting and managing conflicts. Coherence messages now contain two new flags - *transactional status* and *contended status*. An additional flag, *commit status* is added to UNBLOCK requests indicating whether they correspond to commit-time updates. Figure 3 depicts key protocol actions that occur when contended lines are accessed. All cache lines are managed eagerly by default.

Figure 3-b shows steps taken when a switch to lazy management occurs on encountering contention on a cache line for the first time. The transaction interleaving considered here is the one between transactions *T1* and *T2* shown in Figure 3-a. Core 1 (running *T2*) initiates a write to line A (address *0x204*, step 1). The store misses in the private cache structures (step 2) and results in a *TGETX* (GETX with transactional flag set) request to the directory (step 3). This coherence request results in a *TINV* (transactional invalidation) being sent to the reader, Core 0, and data being sent from the L2 to Core 1 (step 4). Core 0, running transaction *T1*, on receiving *TINV* checks if it is currently managing the line eagerly. It finds that it has only read the line transactionally (SR is set) (step 5). Hence it is in a position to forward data to Core 1 for lazy management (otherwise the requester would be stalled till T1 commits or aborts). It marks the line as contended in its private cache (step 6) causing any future write from Core 0 to be managed lazily. A acknowledgment with *contended status* is sent to Core 1 (step 7). Core 1 on receiving such a response places the line in

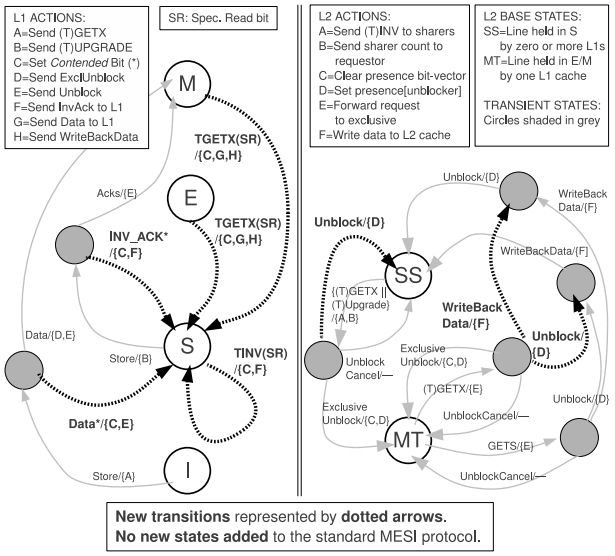


Figure 5: Support for new transitions at L1 (left) and L2-directory (right) controllers.

shared state in its cache and sets the local C bit. The write, instead of updating the cache line, is now buffered in the LWB (step 8). It then indicates completion of the coherence operation by sending an *UNBLOCK* message with contended status to the directory (step 9). On finding a contended status in the *UNBLOCK* message the corresponding C-bit is set at the directory (step 10). The line will now be managed lazily by all accessors until a non-transactional access causes a C-bit reset.

Figure 3-c shows protocol actions that occur when lazily managed lines are published upon commit. The details correspond to interactions between transactions *T2* and *T3* in Figure 3-a. Core 2 (running *T3*) initiates a write to line B (address 0x408, step 1). The line is found in cache with C-bit set. Hence, the write is buffered in the LWB (step 2). When *T3* commits (step 3), it first acquires a global commit token. It then drains the LWB (step 4) acquiring exclusive ownership over line B by sending a non-transactional *UPGRADE* request to the directory (step 5).The directory responds by sending a *INV* (non-transactional invalidation) set to Core 1 (step 6). *T2* on Core 1 aborts when a non-transactional invalidation conflicts with a speculatively accessed line (SR is set, step 7). Since *T2* had the lone lazy write to A in its write set, no old value restoration is required. LWB is reset and re-execution of *T2* can start immediately or when deemed right by a back-off algorithm. It should also be noticed that line C, also part of *T3*'s write set, does not need to be published since it was managed eagerly. Core-1 completes its upgrade operation by sending an *UNBLOCK* message to the directory. This message has the *commit flag* set, causing the directory to maintain a value of 1 for the C-bit. Ordinary requests for exclusive ownership generated from non-transactional code result in *UNBLOCK* messages without the *commit-flag* set and cause the directory to reset the bit.

Cache controllers at both L1 and the L2-directory now support a few new transitions summarized in Figure 5. New transitions are represented by black dotted lines in the figure, while transitions that already exist in the baseline MESI protocol are shown in light grey. For clarity, only states and baseline transitions that aid in illustrating the changes are shown. At the directory level, the behavior of *TGETX*/*TUPGRADE* requests is similar to that of their non-transactional counterparts, but the transactional variants can eventually result in the reception of *contended UNBLOCK* messages that cause a transition to shared state (SS). For example, a *TGETX* from Core 3 could cause a directory transition from *SS@{1,2}* (shared by Cores 1 and 2) to *SS@{1,2,3}* if contention is detected. This permits lazy versioning of contended data at the new requester while still allowing conflict detection to happen. Similarly, the transition from *MT@{2}* (exclusive/dirty at Core 2) to *SS@{1,2}* is supported when handling *TGETX* requests. Such a situation might arise if Core 2 forwards a contended line to Core 1, behaving as if the request had been a GETS (the line is also written back to L2). At the L1, we support transitions to shared state on local write misses and upon receiving *TGETX* or *TINV* requests from the directory. This allows coherence mechanisms to be used to detect conflicts on such data after this event has occurred.

The examples above highlight key behavioral aspects of the ZEBRA protocol. Other cases are handled in a similar fashion. If a transaction is managing a line eagerly, it is given a chance to reach commit by permitting it to stall other requesters. When such events occurred the C-bit for the line is set. This causes the line to be managed lazily once the transaction commits or aborts. The risk of deadlocks is avoided by using a LogTM-like *possible-cycle* bit, but in a different way. The bit is set if the transaction has stalled a requester attempting to access eagerly managed data. When a transaction is stalled by another, it checks if *possible-cycle* flag is set. An abort is triggered if so. The eagerly managed line will henceforth be managed lazily and will no longer be able to cause deadlocks. Transaction timestamps, employed by LogTM for conservative deadlock avoidance, are no longer transported in coherence messages.

Coherence requests generated by non-transactional codes result in an abort if they hit a transactionally accessed line. The flexibility to ask for the requester to retry such requests can be incorporated by transporting the *commit status* flag with invalidation messages. This flag would be set for non-transactional invalidations sent out when the lazy portion of a transaction's write-set is being committed. The receiver, if transactional, would then know that it cannot ask for a retry and must abort if such an invalidation is hits a speculatively accessed line.

3. METHODOLOGY AND EVALUATION

3.1 Experimental Setup

We use a full-system execution-driven simulator based on the Wisconsin GEMS tool-set (v2.1) [10], in conjunction with Wind River Simics [9]. We use the detailed timing model for the memory subsystem provided by GEMS, with the Simics in-order processor model. Simics provides functional simulation of the SPARC-V9 ISA and boots an unmodified Solaris 10 operating system. This simulation infrastructure provides support for LogTM-SE [19], as well as an implementation of a global commit token-based, lazy-lazy (LL) HTM system described by Bobba et al. in [2]. We extend it with detailed implementations of STCC [5] and ZEBRA, our hybrid-policy HTM protocol, allowing fair comparison of several major HTM design points within the same architectural framework. While Bobba's LL system models a private, per processor infinite write buffer, for this study we extended the simulator to precisely model finite buffering for transactional writes. Special status bits are added to coherence messages. Behavior of cache and directory controllers is suitably modified. We use an ideal bookkeeping scheme to track read sets (*perfect signatures*) even when some speculatively read lines have been evicted, in an attempt to isolate our study from the effects of false conflicts arising from non-ideal signature schemes like bloom filters.

Table 1: System parameters.

MESI Directory-based CMP	
Core Settings	
Cores	16, single issue in-order, non-memory IPC=1
Memory and Directory Settings	
L1 I&D caches	Private, 32KB, split 4-way, 1-cycle latency
Write Buffer	Non-coherent, private, 128 bytes
L2 cache	Shared, 512KB per tile, unified 8-way, 12 cycle-latency
L2 Directory	Bit vector, 6-cycle latency
Memory	4GB, 300-cycle latency
Network Settings	
Topology	2D Mesh
Link latency	1 cycle
Link bandwidth	40 bytes/cycle
Flit size	16 bytes

Table 2: STAMP Workload Characteristics.

Workload	Trans Size	Contention	Commit rate
genome	Moderate	Moderate	Moderate
intruder	Small	High	Moderate
kmeans	Small	Low	Low
labyrinth	Large	Moderate	Low
SSCA2	Small	Low	High
vacation	Large	Low	Moderate
yada	Large	High	Moderate

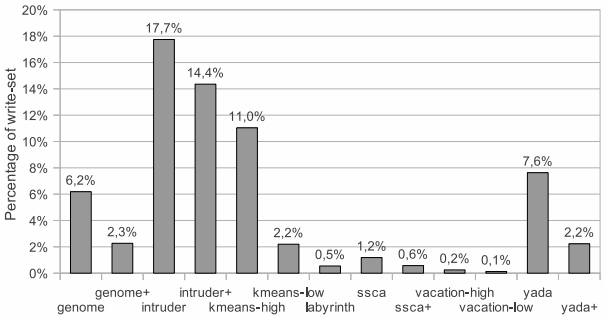


Figure 6: Relative sizes of conflict sets for STAMP applications.

Experiments were performed on a 16-core tiled CMP system, as described in Figure 1. We use a 16-core configuration with private L1I and L1D caches and a shared, multi-banked L2 cache consisting of 16 banks of 512KB each (one L2 slice per tile). For each workload - HTM configuration pair we gathered average statistics over 10 randomized runs designed to produce different interleavings between threads. STAMP workloads were chosen as they are among the most representative TM benchmarks available so far that are suitable for architecture simulation and exhibit a fair diversity in behavior. The parameters for applications have been taken from [3]. We simulate both small and medium datasets for workloads that exhibit differences in transactional behavior across various design points. This not only yields more credible statistics but also allows the hybrid design to achieve steady-state performance.

3.2 Workload Characteristics

STAMP workloads cover a broad spectrum of transactional behavior. Table 2 shows some relevant qualitative characteristics. We have excluded the application, bayes, from our analyses because it exhibits significant variability in execution times. It should be noted that even labyrinth, which implements Lee's routing algorithm, exhibits divergent executions that depend on thread interleavings but these deviations are not so severe as in the case of bayes and hence, we have retained it in this evaluation.

Conflict set sizes. To measure the proportion of contended data in a typical transaction's write set we define conflict set of a transaction as the set of lines that were written and managed lazily over the duration of execution of a transaction. Figure 6 shows the cardinality of the conflict set as a percentage of the corresponding write set size averaged over an entire run. We see that even in applications with moderate to high contention, like yada and intruder, the conflict set is far smaller than the write set. Workloads like ssca2, that have both high concurrency and a high commit rate, experience contention on less than 1% of the write-set. Moreover, as we move to longer running workloads (small to medium in Figure 6), the ratio of the two set sizes drops even further. The common case size of less than 20% of the write-set bears out our choice to the separately manage the large non-contended fraction of the write set.

3.3 Performance Analysis

Figure 7 shows the relative performance of the four HTM designs evaluated in this study. Results are normalized to the execution time of the LogTM system, represented throughout this evaluation by the *EE* label. The *Hybrid* bar corresponds to the ZEBRA HTM design, while the two right-most bars are the lazy designs, the global commit token scheme (*LL-GCT*) and Scalable TCC (*LL-STCC*), respectively. The average for long running workloads (marked with the suffix +) has been calculated separately (appears as Average+). The ZEBRA HTM shows noticeable improvement in overall performance. It closely tracks the performance of the best policy and excels when applications show mixed transactional behavior – having both contended and non-contended phases of execution. Figure 8 zooms in on TM protocol overheads for different designs, normalized to the LL-GCT design. The hybrid policy shows remarkable overall efficiency here – showing 25-30% improvement over EE or LL-STCC. Figure 9 shows two measures – average deviation from the best observed performance over all workloads and the standard deviation of performance normalized to the best across all workloads. The hybrid approach achieves by far the lowest swings, implying consistent performance and robustness. Figure 10 shows scalability of workloads and design points considered in this study and validates the performance of the parallel architecture modeled in the simulator.

We have further investigated the behavior of the hybrid approach. Figure 11 shows the distribution of purely eager, partly-eager-partly-lazy (hybrid) and purely lazy commits in each application. Table 3 shows utilization of LWB and OVB by each transaction (identified by *TID*) in various STAMP benchmarks. Write-set sizes (WS in the table) and OVB occupancy have been shown in cache lines. LWB occupancy represents the number of bytes that were managed lazily in the structure.

The discussion below highlights important observations and presents insights gained from detailed study of interactions between HTM policies and the behavior of individual workloads.

Genome. This workload exhibits a high contention phase early in its execution where lazy designs outperform the EE system. This phase involves removal of duplicates (hash-table insertions). Reader - writer conflicts dominate at the beginning of the phase and lazy approaches inherently allow greater concurrency in such a situation. The hybrid design quickly switches the management of contended cache lines to lazy and completes the phase faster than EE, but a bit slower than LL-GCT or LL-STCC. The second phase is

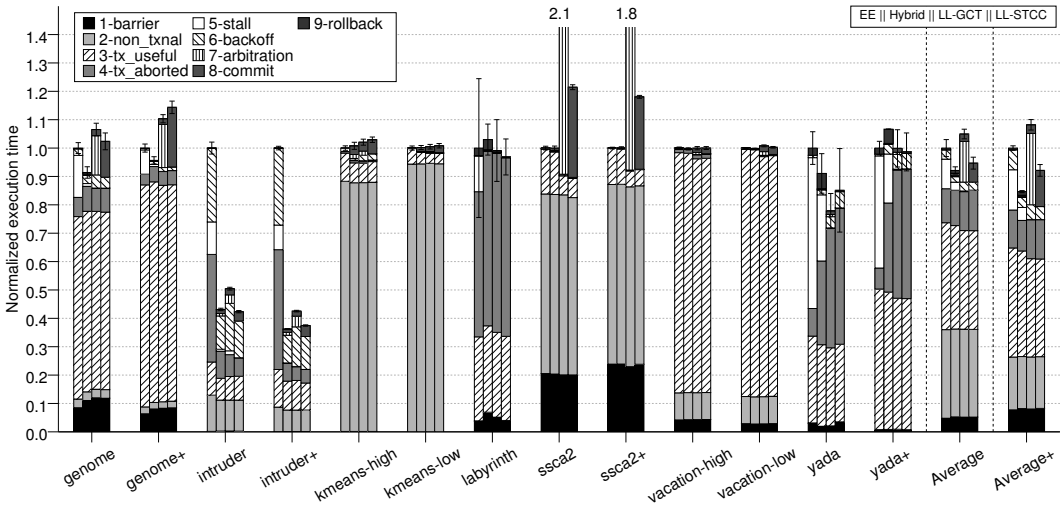


Figure 7: Normalized execution time breakdown.

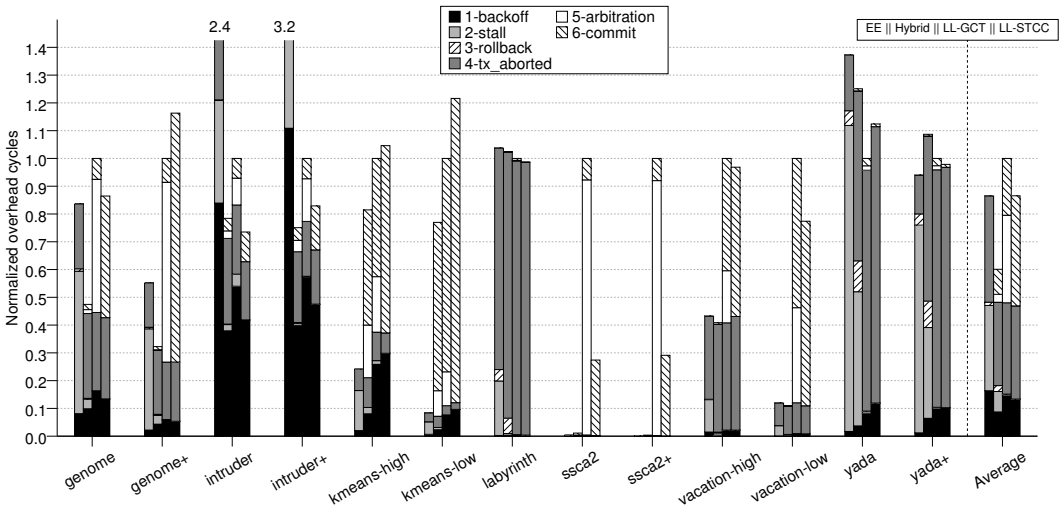


Figure 8: Protocol overheads.

dominated by transactions with moderate write-sets (3.4 cache lines on average) with accesses to predominantly non-contended data and is the determinant of overall performance. The eager approach proves to be the quickest here. The hybrid system run most transactions in a completely eager way and does not suffer from commit overheads seen in the lazy designs. Occupation measurements of the OVB and LWB structures shown in Table 3 confirm how the hybrid system adapts to the transactions of this second phase: TID1 is always eagerly managed (OVB usage equals write-set size), while TID2 and TID3 are purely eager in 70% and 80% of their commits, respectively. No transaction in genome ever commits in a purely lazy fashion (contended lines always comprise only a small fraction of the Wset), demonstrating the benefits of the proposed data-centric approach for policy selection on a per-cache line granularity. The third phase again exhibits low to moderate contention. Since the application shows mixed behavior, the hybrid approach outperforms all others, as depicted in Figure 7. Overall, we find that this result demonstrates the efficacy of quick adaptability to changing workload conditions in the hybrid approach. Genome+ (genome with a medium sized dataset) shows much less contention, thereby widening the gap between the purely lazy and EE or Hybrid designs. As we show in Figure 11, in the hybrid system almost 90 % of commits happen eagerly for genome+.

Intruder. This workload shows high contention, even with large input sizes. Eager transactions acquire exclusive ownership to data before they are guaranteed to commit. This, in conjunction with a high probability of conflicts, leads to prolonged stalls and pathological cases where transactions form chains of dependencies causing aborts. In this contended scenario, lazy systems are able to exploit concurrency much better resulting in far fewer aborts. This workload has 3 transactions. TID0 extracts elements from a highly contended queue of packets, causing the EE system to experience 15K aborts (out of total of 29K aborts overall). Lazy designs reduce this number to 4K (13K aborts overall). The hybrid system quickly discovers contended lines, and the conflicting location (pointer to the head of the queue) becomes lazy (as indicated by an

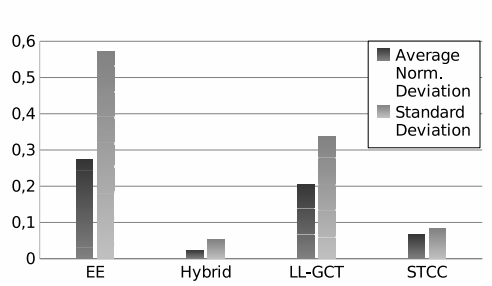


Figure 9: Deviation from best observed performance.

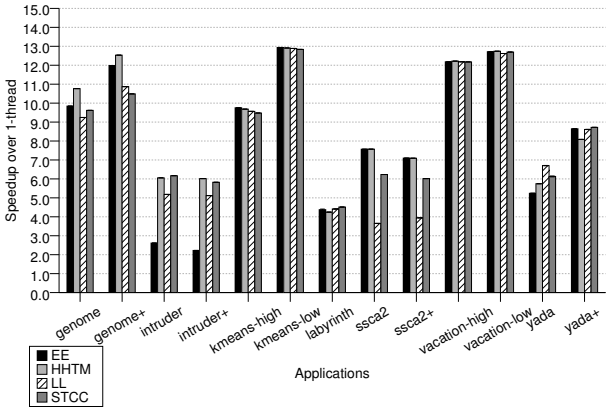


Figure 10: Design scalability.

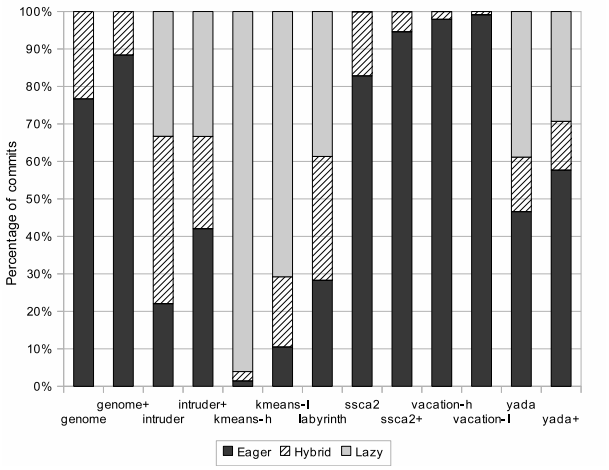


Figure 11: ZEBRA – Policy distribution at commit.

LWB occupancy of 4 bytes in this transaction) decreasing the number of aborts when preforming transactional dequeue operations to 3K (total of 11K). With the hybrid approach, the largest transaction (TID1) can commit eagerly 25% of the time on average, even though it accesses relatively large amounts of contended data (see figure 6). A large fraction of TID1's write set (6.1 lines) is still non-contended and thus an average of 3.8 lines are managed eagerly, as revealed by the OVB occupancy in Table 3. TID2 also exhibits a predominantly eager behavior, committing eagerly about

Table 3: ZEBRA – LWB and OVB utilization.

	TID0			TID1			TID2			TID3			TID4		
Workload	**WS**	**OVB**	**LWB**	**WS**	**OVB**	**LWB**	**WS**	**OVB**	**LWB**	**WS**	**OVB**	**LWB**	**WS**	**OVB**	**LWB**
genome+	1.3	1.2	1.4	1.0	1.0	0.0	**3.4**	**3.3**	**0.7**	3.4	3.4	0.4	2.2	1.8	2.8
genome	1.3	1.1	1.6	1.0	1.0	0.0	**3.5**	**3.1**	**1.9**	3.5	3.3	0.6	2.5	1.4	6.1
intruder+	1.0	0.0	4.0	**5.7**	**4.5**	**7.1**	1.2	1.0	0.9	-	-	-	-	-	-
intruder	1.0	0.0	4.0	**6.1**	**3.8**	**13**	1.5	1.0	2.2	-	-	-	-	-	-
kmeans-h	**2.0**	**0.1**	**65**	1.0	0.0	3.9	1.0	0.0	4.0	-	-	-	-	-	-
kmeans-l	**2.0**	**0.5**	**47**	1.0	0.0	4.0	1.0	0.0	4.0	-	-	-	-	-	-
labyrinth	0.9	0.1	3.1	**217**	**8.0**	**41**	3.8	2.9	4.1	-	-	-	-	-	-
ssca2+	1.0	0.1	3.6	1.0	0.0	4.0	**2.0**	**1.9**	**0.2**	-	-	-	-	-	-
ssca2	1.0	0.1	3.8	1.0	0.0	4.0	**2.0**	**1.8**	**0.7**	-	-	-	-	-	-
vacation-h	**6.8**	**6.8**	**0.2**	5.7	5.7	0.1	4.0	4.0	0.1	-	-	-	-	-	-
vacation-l	**6.1**	**6.1**	**0.1**	5.3	5.3	0.1	2.5	2.5	0.0	-	-	-	-	-	-
yada+	2.5	0.0	11	0.0	0.0	0.0	**70**	**8.0**	**18**	1.0	0.8	0.7	1.3	0.3	5.1
yada	2.0	0.0	8.1	0.0	0.0	0.0	**60**	**8.0**	**40**	1.0	0.5	2.1	1.4	0.2	7.0

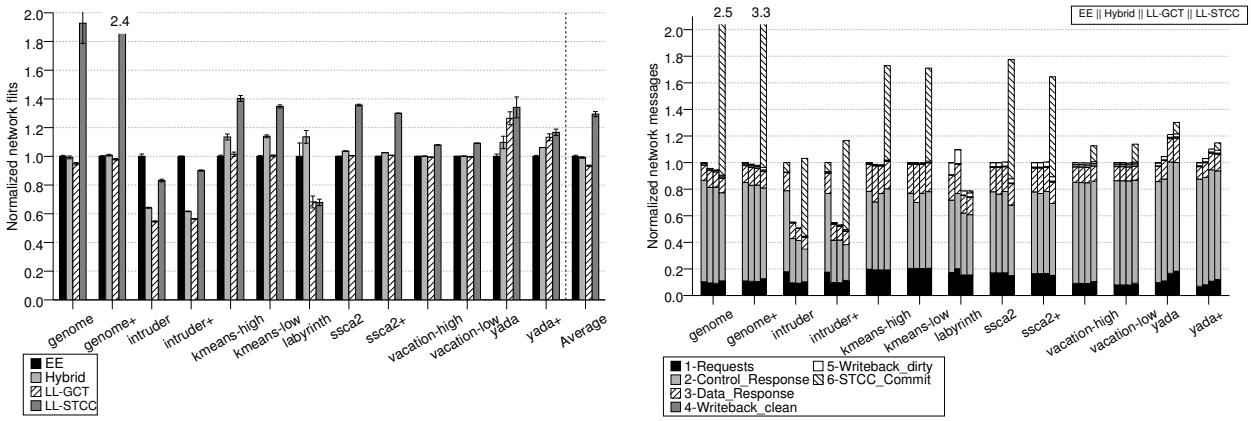
50% of the time, with one eagerly managed write on average (out of 1.5 written lines). Hence, it outperforms lazy approaches since commit durations for TID1 and TID2 are significantly shorter as a large number of the transactionally modified lines are not contended and, therefore, committed instantly. We can see in Figure 8 how the overhead due to the arbitration and commit is substantially lower in the hybrid system, in comparison to both LL-GCT and LL-STCC systems.

SSCA2. It has a large number of tiny transactions that demand high commit bandwidth. Inherently parallel commits in eager approaches serve this requirement very well. Lazy approaches suffer, even STCC, which has a degree of scalability. This is clearly evident in 8 where commit delays represent the primary overhead in lazy designs. The hybrid design is able to manage almost the entire write-set eagerly for most transactions. This can be clearly seen in the high proportion of eager commits (see Figure 11) and the low LWB utilization (see Table 3). Hence, the hybrid approach is able to match the performance of the EE design.

Yada. It has a rather large working set and exhibits high contention. The workload traverses the dataset in a manner which makes it longer for the hybrid approach to complete the discovery of contended lines. A longer time to achieve steady state means that in short duration runs we do not perform as well as the lazy systems. TID2, the dominant transaction (see Table 3), exceeds OVB capacity. This coupled with a relatively high fraction of contended data in the write-set (see Figure 6) results in expensive software rollback operations that degrade performance of EE and the hybrid designs. With longer runs we notice that the differences tend to become less marked. The contention in this case is less severe (as is evident the higher proportion of eager commits for yada+ in Figure 11) and the eager approach regains lost performance.

Labyrinth. As noted earlier, the results generated by this workload depend significantly on the interleaving of threads resulting in marked variability in execution times (note the error bars in Figure 7). The data presented thus should be viewed keeping this fact in mind. The dominant transaction is TID1 as can be seen in Table 3. A significant fraction of data is managed eagerly (see Figure 6) but the OVB capacity is exceeded since write-set sizes are large. Thus, relatively high contention results in expensive rollback operations on abort (see Figure 8). Consequently, lazy designs perform slightly better than the EE or the hybrid approach.

Kmeans /Vacation. These applications are highly concurrent and do not show major differences in execution times with changes in policies. Nevertheless, protocol efficiencies at commit in the EE and the hybrid designs result in minor improvements in performance.



(a) Flit count

(b) Message Type Distribution

Figure 12: Network Traffic.

3.4 Traffic Considerations

Traffic generated by each HTM design when running STAMP applications is shown in Figure 12. Figure 12-a shows traffic volumes in flits normalized to the EE design. Figure 12-b plots the distribution of various protocol messages types transported through the network. In terms of traffic the hybrid approach performs well across all workloads and puts signficantly lower demands on network bandwidth than LL-STCC. LL-STCC shows remarkably high flit counts for several applications, which, as can be seen in the traffic distribution plot, arises due to messaging for scalable commits. In the experimental setup used for this study, large intra-chip communication bandwidth is available as only 16 in-order cores run. The parallel commit algorithm employed by the design is thus able to hide most of messaging latency. In architectures that have a low peak bandwidth or run workloads that impose high communication demands, this latency may not remain hidden and LL-STCC protocol efficiency could suffer.

4. RELATED WORK

Research in HTM design has been very active since the introduction of multicores in mainstream computing. The early proposal by Herlihy and Moss [7] was revived by new, more elaborate designs like UTM[1], TCC [6] and LogTM [19]. In particular TCC and LogTM explored two opposite corners of the HTM design space. Transactional Coherence and Consistency (TCC) contains speculative updates within private caches and resolves races when a committing transaction broadcasts its write-set. In its basic form it employs a bus to serialize transaction commits. The design was later extended to scalable architectures using directory based coherence. A simple variant employs a global commit token to serialize commits. A more sophisticated approach, Scalable TCC (STCC) [5], employs selective locking of directory banks to avoid arbitration delays and thereby improve commit throughput. LogTM, on the other hand, proposes the use of an undo log to incrementally preserve consistent state and abort handlers that restore it. Directory coherence is used to detect and resolve conflicts eagerly with occasional fallback to handlers in software to break deadlocks.

Parallelism at commit is important when running applications with low contention but a large number of transactions. Transactions that do not conflict should ideally be able to commit simultaneously. The very nature of lazy conflict resolution protocols makes it difficult since only actions taken at commit time permit discovery of data races among transactions. Simple lazy schemes like ones employing a global commit token do not permit such parallelism. Hence most lazy protocols employ more complex approaches like finer-grained locks on shared memory [5], optimizing certain safe interleavings [13] and early discovery of conflicts [17]. Eager schemes do not suffer from this problem and our proposal, under such workload conditions, would allow parallel commits since most transactions would be managed eagerly. Thus, complicated protocol extensions to support higher commit parallelism are not critical to improve common case performance for such workloads.

Sanyal et al. [15] proposed filtering of thread-private data with support from the cores and the operating system. While this reduces pressure on versioning mechanisms in HTMs, it does not separate contended data from non-contended data. This separation is not as distinct as that between thread-private and shared data and can only be known by runtime adaptability, as we propose in this work, or by fine-grained profiling of application behaviour and access patterns. The latter is not always feasible because of large variations due to different datasets and thread interleavings.

Mixed-policy HTM designs like DynTM (UTCP) [8] and LV* [12] have been introduced earlier in this work. DynTM deserves further discussion since it chooses a different dimension and granularity of data to work with when compared to the work presented here. It works at the granularity of a transaction and then develops a cache coherence protocol around it that supports multiple ways to version the same shared memory block. This choice of granularity does not match that of the underlying coherence infrastructure which works at the granularity of cache lines. The result, in the opinion of the authors, is increased complexity of design, which will be a significant criterion in any decision to incorporate TM in silicon.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have outlined a fresh approach to hybrid-policy HTM design. Instead of viewing contention as a characteristic of an atomic section of code, we view it as a characteristic of the data accessed therein. Our observation that contended data forms a relatively small fraction of data written inside transactions reinforces our decision to incorporate mechanisms that support efficient management of such data in the common case. In the process, our proposal – the ZEBRA HTM system – manages to bring together the good aspects of both eager and lazy designs with very mod-

est changes in architecture and protocol. ZEBRA supports parallel commits for transactions that do not access contended data and allows reader-writer concurrency when contention is seen. We have shown, both qualitatively and quantitatively, that it can utilize concurrency better and consistently track or outperform the best performing scalable single-policy design – performing as well as the eager design when high commit rates limit performance of lazy designs and, on average, substantially better than both eager and lazy systems when contention dominates. On average, it places lower demands on intra-chip communication bandwidth. It also achieves the lowest deviation from the best measured performance over a diverse set of workloads corroborating our claim that the design is robust and less susceptible to pathological conditions. We hope this work would spur further efforts in the area of low complexity hybrid-policy HTM systems. More research can be done to develop designs that adapt to workload needs quicker and are still cost-effective enough to attract the attention of computer architects.

Acknowledgments

This collaborative work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under grants Consolider Ingenio-2010 CSD2006-00046 and TIN2009-14 475-C04. It was also partly supported by the HiPEAC-2 NoE under contract FP7/IST-217068. Anurag's work at Chalmers has been supported by the European Commission FP7 project VELOX (ICT-216852). Rubén Titos has a research grant from the Spanish MEC under the FPU National Plan (AP2006-04152) and was awarded a collaboration grant from HiPEAC to visit Chalmers.

6. REFERENCES

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proc. of the 11th Symp. on High-Performance Computer Architecture*, pages 316–327, Feb 2005.

[2] Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proc. of the 34th Int'l Symp. on Computer Architecture*, pages 81–91, Jun 2007.

[3] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the IEEE Intl. Symposium on Workload Characterization*, pages 35–46. Sept 2008.

[4] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd Int'l Symp. on Computer Architecture*, pages 227–238, Jun 2006.

[5] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *Proc. of the 13th Symp. on High-Performance Computer Architecture*, pages 97–108, 2007.

[6] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st Int'l Symp. on Computer Architecture*, pages 102–113, Jun 2004.

[7] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Int'l Symp. on Computer Architecture*, pages 289–300. May 1993.

[8] Marc Lupon, Grigorios Magklis, and Antonio González. A dynamically adaptable hardware transactional memory. In *Proc. of the 43rd Int'l Symp. on Microarchitecture*, Dec 2010.

[9] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[10] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, pages 92–99, Sept 2005.

[11] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proc. of the 12th Symp. on High-Performance Computer Architecture*, pages 254–265, Feb 2006.

[12] Anurag Negi, M.M. Waliullah, and Per Stenstrom. LV*: A low complexity lazy versioning HTM infrastructure. In *Proc. of the Intl. Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS 2010)*, pages 231–240, July 2010.

[13] Seth H. Pugsley, Manu Awasthi, Niti Madan, Naveen Muralimanohar, and Rajeev Balasubramonian. Scalable and reliable communication for hardware transactional memory. In *Proc. of the 17th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 144–154, Oct 2008.

[14] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. of the 10th Int'l Symposium on Architectural Support for Programming Language and Operating Systems*, pages 5–17, Oct 2002.

[15] Sutirtha Sanyal, Adrián Cristal, Osman S. Unsal, Mateo Valero, and Sourav Roy. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *HPCC '09: Proc. 11th Conference on High Performance Computing and Communications*, jun 2009.

[16] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proc. of the 35th Int'l Symp. on Computer Architecture*. Jun 2008.

[17] Sasa Tomic, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proc. of the 42nd Int'l Symp. on Microarchitecture*, 2009.

[18] M.M. Waliullah and P. Stenstrom, Classification and Elimination of Conflicts in Transactional Memory Systems. Tech. Report 2010:09, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, 2010.

[19] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of the 13th Symp. on High-Performance Computer Architecture*, pages 261–272, Feb 2007.