ZEBRA: Data-Centric Contention Management in Hardware Transactional Memory

Rubén Titos-Gil, *Member, IEEE*, Anurag Negi, *Member, IEEE*, Manuel E. Acacio, *Member, IEEE*, José M. García, *Member, IEEE*, and Per Stenstrom, *Fellow, IEEE*

Abstract—Transactional contention management policies show considerable variation in relative performance with changing workload characteristics. Consequently, incorporation of fixed-policy Transactional Memory (TM) in general purpose computing systems is suboptimal by design and renders such systems susceptible to pathologies. Of particular concern are Hardware TM (HTM) systems where traditional designs have hardwired policies in silicon. Adaptive HTMs hold promise, but pose major challenges in terms of design and verification costs. In this paper, we present the ZEBRA HTM design, which lays down a simple yet high-performance approach to implement adaptive contention management in hardware. Prior work in this area has associated contention with transactional code blocks. However, we discover that by associating contention with data (cache blocks) accessed by transactional code rather than the code block itself, we achieve a neat match in granularity with that of the cache coherence protocol. This leads to 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. ZEBRA, therefore, brings together the inherent benefits of traditional eager HTMs—parallel commits—and lazy HTMs—good optimistic concurrency without deadlock avoidance mechanisms—, combining them into a low-complexity design.

Index Terms—Multicore architectures, transactional memory, parallel programming, cache coherence protocols

1 Introduction and Motivation

Most microprocessor road-maps today project rapid growth in the number of cores integrated on chip in an attempt to provide increasing performance through thread level parallelism. This has brought the problem of effective concurrent programming of such systems to the forefront of computing research, presenting both immense opportunities and enormous challenges. In the context of shared memory architectures where concurrent tasks co-operatively process shared data, guaranteeing correctness while maintaining efficiency and productivity is one key challenge. Traditional multi-threaded programming models use low-level primitives such as locks to guarantee mutual exclusion and protect shared data. Unfortunately, the complexity of lock-based synchronization makes parallel programming an error prone task, particularly when fine-grained locks are used to extract more performance.

Transactional Memory (TM) [8], [10] has been proposed as a conceptually simpler programming model that can help boost developer productivity [9] by eliminating the complex task of reasoning about the intricacies of safe fine-grained locking. At a high level, the programmer or compiler annotates sections of the code as atomic blocks or *transactions*. The underlying system executes these transactions atomically and in isolation from other any other concurrently running code, while exploiting as much parallelism as possible. By using transactions to safely access shared data, programmers need not reason about the safety of interleavings or the possibility of deadlocks to write correct multi-threaded code. Hence, TM addresses the performance-productivity trade-off by not discouraging programmers from using coarse-grain synchronization, since the underlying system can potentially achieve performance comparable to fine-grained locks by executing transactions speculatively. The TM system attempts to make best use of available concurrency in the application while guaranteeing correctness.

Fast implementations of transactional programming constructs that provide optimistic concurrency control with stringent guarantees of atomicity and isolation are necessary for TM to gain widespread usage. Software TM [21] (STM) implementations impose too high an overhead and do not fare well against traditional lock based approaches when performance is important. Hardware TM (HTM) [1], [4], [5], [7], [16], [25] systems show much greater promise. Yet, within the design space of HTM systems, there are trade-offs to be made among various pertinent metrics like design complexity, speed and scalability. Early work on HTM proposals [7], [25] 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 within that framework.

Results in research so far do not show a clear winner or an optimal design point. Lazy HTMs, which confine speculative updates locally and run past data races until a

- *R. Titos-Gil, A. Negi, and P. Stenstrom are with the Department of Computer Science and Engineering, Chalmers University of Technology, 41296 Gothenburg, Sweden. E-mail: {ruben.titos, negi, per.stenstrom}@chalmers.se.*
- *M. Acacio and J. García are with the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, 30100 Murcia, Spain. E-mail: {meacacio, jmgarcia}@ditec.um.es.*

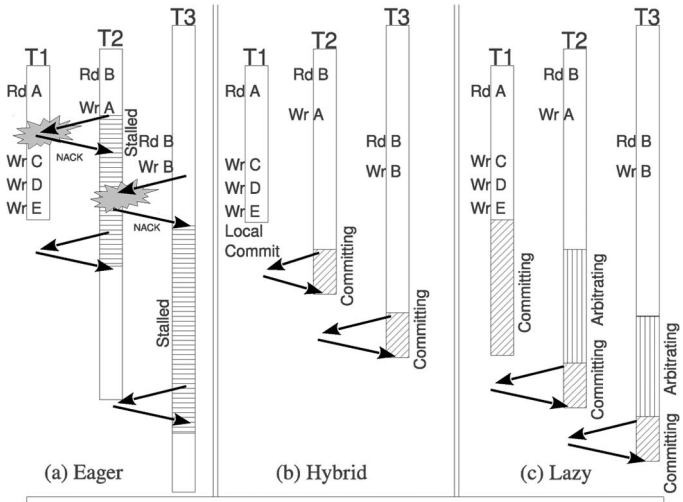
Manuscript received 24 May 2013; revised 19 Sept. 2013; accepted 27 Sept. 2013. Date of publication 9 Oct. 2013; date of current version 21 Mar. 2014. Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.262

transaction ends, do seem to be more efficient at extracting parallelism [22] but require elaborate schemes [5], [19], [24] to make race free publication of speculative updates (i.e. transaction commit) scalable. Eager HTMs, which 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 resolve 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 hybrid HTM design, which selects the best performing policy (eager or lazy) depending on workload characteristics, would be close to the optimal point in the HTM design space. A key factor would then be the complexity involved in realizing such a design in hardware. The first attempt to provide a hybrid-policy HTM design was DynTM [12], at which heart lies a cache coherence protocol (UTCP) that allows transactions in a multi-threaded application run either eagerly or lazily based on some heuristics like prior behavior of transactions. Although it lays down an interesting approach, UTCP is a significant departure from existing cache coherence designs, as we show in this paper, and its additional complexity involved for just supporting TM represents too high a design cost.

In this work we propose a different 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 shared but not actively contended [23]. 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 versioning and contention management at which minimal changes are required to support both eager and lazy policies in existing cache coherence protocols: the cache line. Our design annotates cache lines as being either contended or not. Contended lines are managed lazily, thus permitting greatest concurrency among transactions. All non-contended lines are versioned eagerly and thus only contended lines need to be published on transaction commit. We call this data-centric, hybrid-policy HTM protocol *ZEBRA*.[1]

1. 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).



A, B are contended variables, while C ,D and E are not. The eager HTM (a) stalls T2 when it faces a conflict with T1. T3 is then stalled by T2 causing a chain of conflicts that spans T1, T2 and T3.
The hybrid approach (b), like the lazy system (c) sidesteps this issue by managing contended lines lazily. Moreover, in the hybrid case T1 is able to perform an instant commit since its write set {C,D,E} is managed eagerly, thus leading to shorter commit arbitration delays experienced by T2 and T3.

Fig. 1. Behavioral differences between HTM designs.

Fig. 1 depicts an interleaving of three concurrent transactions and highlights some important behavioural aspects of our proposal, when compared to fixed-policy approaches. In the eager case (Fig. 1a), 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 (Fig. 1b) or the purely lazy case (Fig. 1c) and in the example shown all three transactions commit without conflicts. The figure shows how eager systems disallow the reader-writer concurrency that naturally occurs in lazy systems if the reader commits before the writer. It should be noted here that in ZEBRA 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 (i.e. when a simple global commit token is used for serializing commits [2]). This in turn delays $T3$'s commit. With ZEBRA, $T1$ 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 $T1$.

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 our evaluation shall show, substantial gains over existing fixed-policy HTM designs are achieved. The incremental cost of implementing this approach is minimal since only very modest behavioural changes are required in the cache coherence protocol, as demonstrated in Section 4.

There are also certain other benefits that stem from using such a data-centric approach. First, deadlock avoidance mechanisms such as that used by LogTM [17] are not

required since contended lines are eventually managed lazily, thereby guaranteeing forward progress. Also, significant reductions in transaction commit delays result in a major contraction of the window of contention for concurrent transactions. Furthermore, the pressure on lazy versioning mechanisms is considerably reduced, making cache evictions of lazy speculatively modified data less likely and hence enabling much larger transactions to run without resorting to safety nets. Finally, since the design does not lock policy and is able to revert the contention annotations, it can adapt to changing workload conditions and is resistant to pathologies that fixed policy HTMs suffer from. Overall, this proposal touches upon a sweet spot in the HTM design space that offers both simplicity of design and robust performance.

A first implementation of ZEBRA was presented in [23]. Here, we extend that work in the following ways:

- We have incorporated a comparative complexity analysis of our approach, describing in detail the coherence protocol extensions that ZEBRA entails over a standard MESI, in contrast to the UTCP protocol that forms the basis of DynTM [12].
- We have incorporated prediction and reversion mechanisms to dynamically switch between eager and lazy management of cache lines based on changes in their contention characteristics.
- The design has been evaluated more comprehensively with the inclusion of DynTM and other new design points. A new comparison against idealized eager and lazy designs provides a better measure of the performance potential of our protocol.

The rest of this paper is organized as follows. Section 2 puts our work here in perspective of other work in HTM systems on related issues. Section 3 describes the salient architectural and behavioural features of ZEBRA. In Section 4 we present our complexity analysis versus DynTM. Section 5 describes the experimental methodology adopted, and then Section 6 presents our results. Section 7 concludes the paper with our take away message about low-complexity hybrid-policy HTM design.

2 Background and Related Work

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 bus [7] or global commit token [2] 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 [19] and early discovery of conflicts [24]. 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.

Lupon *et al.* proposed DynTM [12], the first hybrid-policy HTM design capable of dynamically adapting its choice of policy to the workload characteristics. DynTM deserves further discussion since it constitutes a different solution to the same challenge addressed by our work, i.e. finding a point in the HTM design space which combines the advantages of eager and lazy policies. The key difference between DynTM and ZEBRA is that the former chooses a different dimension when combining eager and lazy: DynTM selects policies at the level of transactions, while ZEBRA is a data-centric design which works at the granularity of cache lines. DynTM's choice of granularity does not match that of the underlying coherence infrastructure, which works at the granularity of cache lines. The result, as further discussed in Section 4, is increased complexity, which will be a significant criterion in any decision to incorporate TM in silicon. In regards to adaptability, DynTM adapts based on a history of past transactions, trying to figure out the best policy for each transaction, which is an inherently slow process. Switching entire transactions from eager to lazy is cumbersome as well. ZEBRA, on the other hand, adapts seamlessly. If a transactional reader exists for an eager line when a conflicting write is issued, policy switch to lazy occurs without need for either stall or abort. A stall can only occur if an eager writer exists and briefly lasts while the writer aborts. Moreover, unlike DynTM, after a policy switch the behaviour of a transaction does not change drastically. As ZEBRA discovers contention for shared data a gradual shift in behaviour occurs permitting fine-grained adaptability. The reader is referred to the online supplemental material (Sections 2 and 4) for further detail about previous works.

3 The ZEBRA Design

We choose a tiled chip multiprocessor as the baseline architecture upon which our design is built. Each tile comprises a processing core and a slice of a shared inclusive L2 cache and corresponding directory entries. Each processing core has private Level 1 instruction and data caches. The directory keeps private caches coherent using a MESI protocol. The tiles are interconnected by a mesh-based routing network. ZEBRA uses eager versioning by default, and thus it inherits features such as the logging logic and the read and write signatures from LogTM-SE [25]. Fig. 1 of the supplemental material available online shows a high-level overview of the hardware components required by the ZEBRA design. Read and write set signatures [4] are used to track speculatively read as well as eagerly-managed speculatively written lines, and they allow such lines to be evicted from the private cache level. In order to perform lazy versioning in private caches, each cache entry is augmented with a *speculatively modified* (SM) bit, which also enables gang-invalidation of all SM lines on abort, similarly to prior HTM proposals [7]. A transaction with no lazy updates can commit without delay, permitting true commit parallelism in such a case. If there are some lazy updates, they must be validated and

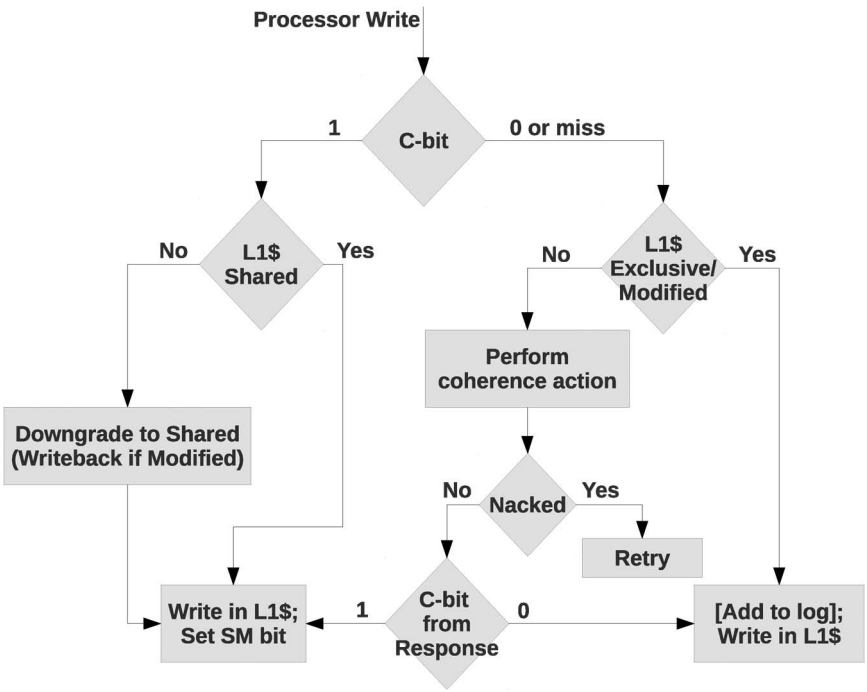


Fig. 2. Write handling at the L1 cache.

made globally visible. We adopt the simplest possible approach to do so by having the committer acquire a global commit token [2] to achieve a global serial order. While more scalable lazy commit schemes [5] would further enhance our proposal, the design choice is orthogonal to the key ideas described in this work. Once commit is granted, a simple state machine issues coherence requests to obtain exclusive ownership for each address in the *lazy address table*. Thus, all lines in the shared (S) state in cache with SM indicator set are upgraded to modified (M) state. The invalidations forwarded by the directory cause the abort of any concurrent transactional reader or lazy writer.

3.1 Policy Selection Based on Contention Meta-Data

In order to track contention, ZEBRA extends per-cache line meta-data at the private cache and directory levels with just one additional bit—''*contended bit*''—hereafter referred to as the *C-bit*. The C-bit is transported with all coherence messages. A C-bit value of ''1'' indicates that the line has experienced contention in the past. C-bits at the directory are set by unblock messages upon completion of in-flight coherence operations which discovered contention. 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 behaviour 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. Apart from the C-bit, all coherence messages generated in response to speculative accesses by the core are distinguished from ordinary ones by setting a special *transactional* status 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.

Fig. 2 shows how writes from the processor are dealt with by the private cache controller. 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. This ensures that 1) consistent values are preserved in the shared L2 level, 2) gang-invalidations of SM lines on abort simply appear as silent S replacements, and 3) multiple speculative readers and writers are allowed (tracked as sharers by the directory protocol). Updates to lines that are either non-contended or have unknown C-bit status (misses) are treated as usual, issuing exclusive coherence requests to the directory if L1 line-fill or write permissions are required. If the coherence operation succeeds but indicates that the line is contended (i.e. there are concurrent readers), the line is allocated in the cache (if not already present) with shared permissions, and both SM and C bits are set. If the C-bit in the response is not set, the update happens ''in place'' and the old contents of the line are written to a thread-private log in virtual memory. This aspect of eager behaviour is similar to that of LogTM [25]. In the event that a load or store miss finds a concurrent eager writer, the latter will set the C-bit, trigger abort and respond with negative acknowledgements (*nack*) as long as the write signature signals the conflict. Upon reception of a nack, the requester simply retries after a few cycles. The request will eventually succeed, once the aborting writer has unrolled log (restoring the consistent value) and the write signature has been cleared. 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 either livelocks or deadlocks. Note that ZEBRA provides the same forward progress

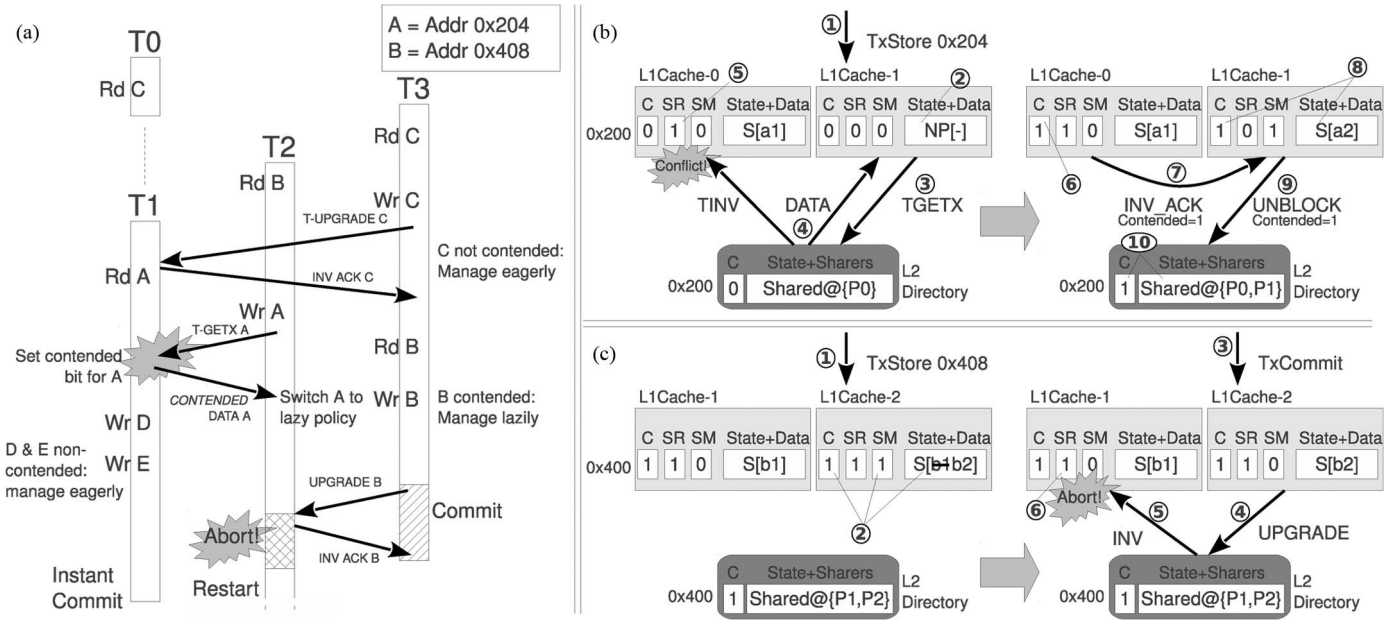


Fig. 3. ZEBRA—Key protocol actions.

guarantees of a lazy system, and thus renders LogTM's conservative deadlock avoidance mechanism based on TLR-like timestamps [20] unnecessary.

3.2 Protocol Behaviour

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. Fig. 3 depicts key protocol actions that occur when contended lines are accessed. All cache lines are managed eagerly by default.

Fig. 3b 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 Fig. 3a. 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 an exclusive coherence request with transactional status (*TGETX*) sent to the directory (step 3). The line is marked as shared at the L2 directory, which forwards transactional invalidations (*TINV*) to the sharers (step 4), in this case Core 0, and responds with clean data and the ack count (number of acknowledgements to expect) to Core 1, following its usual behaviour. When Core 0 receives the invalidation, it checks its read and write signatures, as well as the SM bit for the line. It finds that it has only read the line transactionally (step 5), thus detecting a conflict (note that in Fig. 3 we use a speculatively read—SR—bit to indicate that the line address hits in the read signature, for clarity). Core 0 then marks the line as contended in its private cache (step 6) causing any future write from Core 0 to be managed lazily. An acknowledgement with *contended status* is sent to Core 1 (step 7). When Core 1 receives such a response, places the line in shared state (rather than exclusive) in its cache, sets the local C bit and performs the lazy write in cache (step 8). Finally, it indicates completion of the coherence operation by sending an unblock message with contended status to the directory (step 9). The directory sets the C-bit for the line and instead of marking Core 1 as exclusive owner, it adds it to the *sharers* bit-vector, along with Core 0 (step 10). The line will now be managed lazily by all accessors until a non-transactional access causes a C-bit reset or the reversion logic described in Section 3 of the supplemental material determines that contention has dissipated.

Fig. 3c shows protocol actions that occur when lazily managed lines are published upon commit. The details correspond to interactions between transactions $T2$ and $T3$ in Fig. 3a. Core 2 (running $T3$) initiates a write to line B (address 0x408, step 1). The line is found in cache with its C-bit already set and in shared state, and so the store performs lazily in cache and the SM bit is set (step 2). When $T3$ commits (step 3), it first acquires a global commit token and then issues a non-transactional request to the directory for upgrading from shared to exclusive permissions over line B (step 5). The directory responds by forwarding invalidations to Core 1, which checks its read signature (step 6) and proceeds to abort $T2$ upon detecting a conflict with a non-transactional invalidation. 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 exclusive 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.

4 Analysis of Protocol Complexity

Designing and validating the coherence protocol, arguably the most complex part in the memory hierarchy of chip multiprocessor, is not an easy task [14]. As part of this

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
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

study, we show that the ZEBRA protocol is considerably simpler than UTCP, the coherence protocol of DynTM [12]. The reader is referred to Section 4 of the supplemental material for a detailed description of both protocols. Our in-depth comparative analysis of both approaches to hybrid-policy HTM design shows that the root cause behind their different levels of complexity is the dimension chosen for selecting the management policy (transactions versus cache lines). By switching TM policy at the transaction level, DynTM treats all data the same way regardless of the contention experienced. To make up for this suboptimal decision, DynTM then places strong demands on the protocol, which is significantly modified to allow both good lazy commit scalability with no contention, and low rollback overheads during contention. Two key features are responsible for this departure in terms of complexity. The first of such features is the support for fast abort of eager transactions inherited from FASTM [11] (i.e. eager transactions restore consistent values without the need to unroll the undo log in software). The second feature is the support for *local* commits of lazy transactions (i.e. lazy transactions make their updates globally visible at commit time without communication at that point). As our performance evaluation will reveal, the ZEBRA design represents a more cost-effective solution to hybrid-policy HTM systems.

5 Experimental Methodology

Full-system execution-driven simulation, based on the Wisconsin GEMS tool-set [15] and Simics [13], is used for our evaluation. GEMS provides a detailed timing model for the memory subsystem, which is connected to the Simics in-order processor model. Simics provides functional simulation of the SPARC-V9 ISA and boots an unmodified Solaris 10 operating system. Experiments were performed on a 16-core tiled CMP system, detailed in Table 1. Each tile contains private L1 caches and a slice of the logically-shared physically-distributed L2 cache.

The STAMP transactional benchmarks with recommended inputs are used as workloads [3]. *Bayes* was excluded since it exhibits unpredictable behaviour and high variability in its execution time [6]. For kmeans and vacation, only results for the high contention input are shown, as these benchmarks exhibit barely no remarkable performance variations between eager and lazy systems, for both high and low contention configurations. Small input sizes were used for all workloads. Medium length runs (denoted by '+') were also included for five applications that show widely varying transactional characteristics—from vacation and ssca2 at the low contention end to yada and intruder at the high contention end. For each workload-configuration pair we gathered average statistics over 10 randomized runs.

Table 2 lists all the HTM design points evaluated in this work. The reader is referred to the supplemental material available online (Section 5) for more details about each design.

6 Performance Evaluation

All the results shown in this section are normalized to the EE system. Averages for small and medium size (+) inputs are shown separately. Execution time is broken into disjoint components, so that we can better understand the roles that each source of overhead plays on HTM performance: rollback and stalls in the eager case, arbitration and commit in the lazy case, and aborted transactional execution in both cases. Fig. 4 shows the relative performance of ZEBRA, compared to both realistic and ideal flavours of HTM designs that cannot adapt their policy (see Table 2). In it we see that ZEBRA provides marked gain in overall performance (18 percent over EE and 5-8 percent over LL-GCT). Our hybrid design closely tracks the performance of the best policy for each workload and excels when applications show mixed behaviour—having both contended and non-contended phases, e.g. genome. This confirms that ZEBRA is able to combine the best of both eager and lazy, to achieve consistent performance across a variety of workloads.

Evaluating ZEBRA against idealized fixed-policy designs further reveals the clear advantages of our approach. On the one hand, the comparison with LL-IdealCommit highlights how ZEBRA can achieve performance at par with that of a lazy design with truly parallel commits, despite the fact that ZEBRA uses a very simplistic commit scheme that precludes any lazy commit parallelism. The reason behind this is the data-centricity of our hybrid approach: As shown in our previous study [23], contended lines represent a very small fraction of the transaction's write set, and thus the publication phase of transactions

TABLE 2
HTM Configurations Evaluated in Section 6

Configuration	Description
Eager	
EE	The LogTM-SE design[25]
EE-IdealAbort	LogTM-SE with ideal instant rollbacks
DynTM-E	The FASTM design [11]
Lazy	
LL-GCT	Lazy-Lazy with global commit token [2]
LL-IdealCommit	Lazy-Lazy with ideal parallel commits
Hybrid	
ZEBRA	The ZEBRA design
DynTM-O	DynTM-overflow mode [12]
DynTM-D	The DynTM design [12]

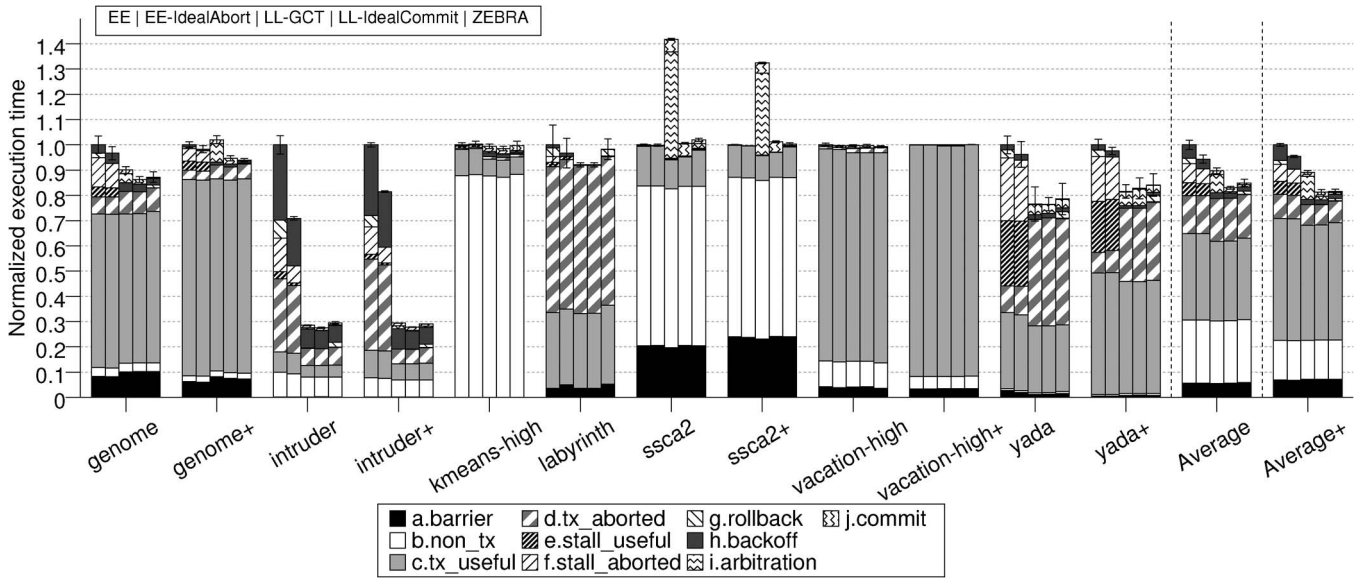


Fig. 4. Execution time breakdown of ZEBRA vs. fixed-policy HTM designs.

that write to contended data is considerably shortened. Most written data (if not all, when no contention) is managed eagerly and thus needs no further coherence actions at commit time.

On the other hand, by comparing against EE-IdealAbort, we observe in Fig. 4 that rollback overhead is not the main reason for the performance loss seen in the EE systems, a fact which favours a mostly-eager system like ZEBRA—since the majority of the data is managed eagerly, aborts are often done in software. We see that overall EE performance is worse than lazy even when rollback penalty is completely removed. This is particularly noticeable in applications with moderate to high levels of contention like genome, intruder or yada. Even in contended workloads with large transactions (like yada or labyrinth), rollback penalty is not that important, as transaction size in itself leads to low abort counts. Only benchmarks with high contention over small transactions (intruder) find significant gains in reduced rollback overhead, because even a small increase in abort latency translates into even more contention.

Table 3 shows key statistics that offer a closer look into the behaviour of our proposal. First of all, speed-ups with respect to single-thread runs obtained for our 16-thread configuration appear below each benchmark name, validating the scalability of the design. The table then shows detailed number of commits ($\#Tx$), aborts ($\#Abr$) and average write set size (WS, in cache lines), for each static transaction $TxID\text{-}n$ found in the source code. For benchmarks with more than three atomic blocks (e.g. genome, yada), only the first three are shown, due to space constraints. The numbers in brackets found below the commit count and write set size represent, respectively, the fraction of lazy commits and lazily managed lines in the write set. Finally, the last two columns illustrate the use of the policy predictor (commits with lazy default prediction) and the safety net mechanism (commits after lazy overflow).

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

Genome

This benchmark comprises several phases with varying levels of contention. Reader-writer conflicts frequently appear during hash table insertions in the initial phase, creating contended scenarios in which lazy approaches inherently allow greater concurrency and thus outperform EE. ZEBRA quickly switches the management of contended cache lines to lazy and also avoids the futile stalls suffered

TABLE 3
Hybrid, Data-Centric Management in ZEBRA

Benchm (Scalab)	TxID-0 #Tx (%L)	TxID-0 WS (%L)	TxID-0 #Abr	TxID-1 #Tx (%L)	TxID-1 WS (%L)	TxID-1 #Abr	TxID-2 #Tx (%L)	TxID-2 WS (%L)	TxID-2 #Abr	Lazy Pred (%)	Ov er flw
genome	1376	1	379	241	1	0	3615	3	494	139	0
(10,9)	(7)	(27)		(0)	(0)		(31)	(11)		(2)	
genome+	2736	1	407	481	1	0	14911	3	835	149	0
(13,4)	(5)	(25)		(0)	(0)		(13)	(5)		(1)	
intruder	3769	1	2188	3753	6	6599	3753	2	304	2374	0
(6,2)	(100)	(100)		(89)	(72)		(55)	(35)		(21)	
intruder+	18322	1	30382	18306	6	12283	18306	1	1050	4126	0
(6,6)	(100)	(100)		(58)	(39)		(22)	(18)		(8)	
kmeans	6144	2	1223	2046	1	247	48	1	0	0	0
(9,9)	(99)	(97)		(98)	(98)		(100)	(100)		(0)	
labyrinth	112	1	46	96	217	149	16	4	1	48	0
(4,4)	(84)	(98)		(75)	(35)		(100)	(27)		(21)	
ssca2	16	1	51	16	1	51	47267	2	693	0	0
(8,4)	(100)	(100)		(100)	(100)		(29)	(15)		(0)	
ssca2+	16	1	15	16	1	55	93693	2	594	0	0
(8,1)	(100)	(100)		(100)	(100)		(12)	(6)		(0)	
vacation	3688	7	105	204	6	1	204	4	5	2	0
(14,1)	(2,0)	(0,3)		(2,5)	(0,5)		(3,4)	(0,9)		(0)	
vacation+	3688	7	4	204	5	0	204	4	1	0	0
(14,7)	(0,1)	(0,0)		(0,0)	(0,0)		(1,0)	(0,2)		(0)	
yada	1352	2	278	1336	0	0	948	59	2049	871	25
(6,2)	(99)	(100)		(0)	(0)		(94)	(87)		(16)	
yada+	3213	2	1013	3197	0	0	2667	70	2603	2002	69
(10,2)	(100)	(99)		(0)	(0)		(73)	(55)		(14)	

by EE (*stall_aborted* in Fig. 4). Subsequent phases are dominated by transactions with accesses to mostly non-contended data, where the eager approach proves to be the quickest. As shown in Table 3, despite the aborts seen in TxID-0 and TxID-2, only a small fraction of all committed writes are managed lazily by ZEBRA: 25-27 percent in TxID-0, and 5-11 percent in TxID-2. Because ZEBRA runs most of these transactions in a fully eager way (for instance, in genome+ only 13 percent of TxID-2 commits have a lazy portion), it does not suffer from the arbitration and commit overheads seen in LL-GCT.

Intruder

This workload shows high contention in two of its three atomic blocks. TxID-0 extracts elements from a highly contended queue of packets, causing the EE systems to experience a large number of aborts due to conflicts on a single cache line (pointer to the head of the queue). ZEBRA immediately turns this line to lazy (roughly 100 percent of TxID-0 commits are completely lazy) to emulate the LL systems, reducing the number of aborts dramatically. For such heavily-contended small transactions that read-modify-write a single cache line, the lazy approach to conflict detection and resolution proves to be more efficient, since requests for exclusive ownership are only issued once the transaction has acquired permission to commit. Eager detection performs badly because it chokes the directory with futile exclusive requests from transactions that end up aborting, obstructing the processing of other requests from higher-priority transactions, thus reducing commit throughput. Lazy detection naturally combines in one message the tasks of invalidating remote copies (and aborting conflicting transactions) and publication of the committed value. It is interesting to note how in intruder+, despite the high contention seen on its main transaction TxID-1 (around 40 percent of all attempted instances have to abort), most of the write-set is still managed eagerly (61 percent). With the small input, contention on TxID-1 is so high (65 percent aborts) that the default policy prediction switches to lazy (21 percent of all commits used lazy prediction), explaining why 72 percent of the write set is managed lazily.

SSCA2

In its main phase (TxID-2), it has a large number of tiny transactions that demand high commit bandwidth. Inherently parallel commits in eager approaches serve this requirement very well. The LL-GCT approach suffers, as is clearly evident in Fig. 4, where commit delays represent the primary overhead. ZEBRA is able to manage the entire write-set eagerly for most transactions (71-88 percent), matching the performance of EE.

Labyrinth

The dominant transaction is TxID-1, with a write set of over 200 lines. Although a significant fraction is not contended (a thread-local copy of the grid where the routing is computed) and thus managed eagerly (65 percent), the sum of the logging and rollback overheads makes both ZEBRA and EE perform slightly worse than the lazy systems. In spite of the huge write set, the latter can take full advantage of the lazy versioning capabilities of the private cache without resorting to safety nets.

Yada

The dominant transaction (TxID-2) exhibits high contention over a large write set (60-70 lines). In EE, 30 to 50 percent of the accumulated cycle count corresponds to stalls, half being futile. Though EE resolves some conflicts through stalls rather than the wasteful aborts (note the larger *tx_aborted* component in Fig. 4 for LL and ZEBRA), stalling produces unnecessary thread serialization where lazy approaches are better at exploiting parallelism. ZEBRA performs slightly worse than LL because the predictor takes a few repeated aborts to discover that contention is spread across a large portion of the shared data, and change the default policy to lazy.

Kmeans/Vacation

These applications are highly concurrent and do not show major differences in execution times with changes in policies.

6.1 ZEBRA vs. DynTM

Fig. 5 shows the relative performance of ZEBRA, compared to the different flavours of DynTM described in Section 5. The first observation that we make out of this figure is that DynTM-E obtains in most cases performance levels similar to EE, baseline of the normalization. While benchmarks that exhibit moderate contention (such as genome and labyrinth) show minor improvements, the protocol support for local aborts in UTCP only brings a substantial benefit in those workloads with many small and highly contended transactions (the case of intruder). Though DynTM-E achieves a reduction in execution time of roughly 40 percent over EE, it is still far from the performance of ZEBRA. As we mentioned earlier, rollback overheads are not the primary cause of the poor performance achieved by eager approaches but rather their tendency to collapse the directory with requests—since conflicting requests are retried—for those few contended lines with read-modify-write behaviour.

DynTM-O, which in benchmarks without L1 cache overflows (all, except yada) executes all transactions in lazy mode, does not seem to handle high contention better than DynTM-E. The reason behind this unexpected pathological behaviour is that, in order to enable local commits of lazy mode transactions, DynTM leverages the underlying UTCP protocol for eager conflict detection, as it was discussed in Section 4. As shown in previous studies [18], the eager-detection lazy-resolution of conflicts in DynTM exposes a weakness not found in lazy-lazy designs or ZEBRA. The requirement to detect all conflicts before a lazy-mode transaction can commit brings the latency for doing so in the critical path. This latency grows longer and longer as the level of contention increases for these ''hot'' cache lines. In the case of intruder, the directory is permanently choked with requests to these lines, which often belong to doomed transactions (i.e. those that have received an abort message from a racing transaction while waiting for their pending cache miss to complete). To complicate things even further, the decision to remove back-off

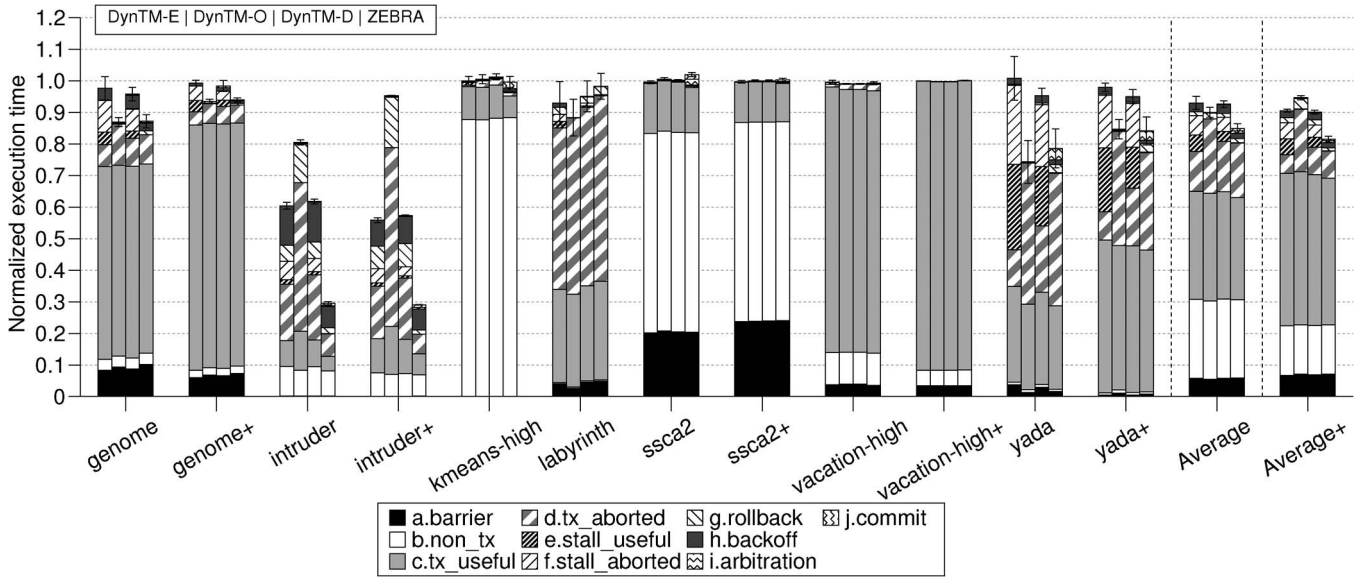


Fig. 5. Execution time breakdown of ZEBRA vs. the different DynTM flavours.

upon lazy mode aborts proposed in DynTM creates even more contention for the conflicting cache lines at the directory, explaining the performance drop in comparison to DynTM-E. In contrast, the lazy conflict detection approach of ZEBRA works well in this case since it can combine conflict detection with write-set publication.

Apart from the pathological behaviour in intruder, the lazy-mode of DynTM allows more concurrency than its eager counterpart in workloads with moderate contention and larger transactions. DynTM-O achieves roughly the performance of ZEBRA in genome and yada, and it obtains a relative reduction of 10 percent in execution time for labyrinth, thanks to its ability to exploit the versioning capabilities of the private cache to a larger degree than our proposal (ZEBRA still relies on logging for the most part). It should be noted that for the small input size of labyrinth, DynTM-O is able to lazily version all speculative writes in L1 without resorting to the overflow mode (abort and restart in eager mode). As problem size increases (larger grids), the performance gap between ZEBRA and DynTM is expected to close, since L1 cache will no longer be able to lazily version all the speculative updates.

An interesting observation is that DynTM-O allows good concurrency in moderate to high contention, and it does so without introducing any penalty in low-contended scenarios that require good commit bandwidth (e.g. ssca2), thanks to the fancy mechanisms implemented by the UTCP protocol to provide local commits of lazy transactions. Overall, Fig. 5 shows that DynTM-O achieves an average improvement over the EE baseline of 5-10 percent, whereas our proposal reaches 15-19 percent on average due to its more robust performance across all evaluated workloads.

Let us now analyse the case of DynTM-D, which dynamically chooses the most profitable execution mode based on the transactional history and two heuristics. On the one hand, DynTM-D keeps track of transactions that are prone to suffer overflow of lazy speculative updates in L1 cache, and the predictor determines that those transactions should execute in eager mode. As we have mentioned earlier in this section, yada is the only benchmark for which the prioritized retention of SM lines done by the replacement algorithm does not suffice in avoiding transactional overflows. Therefore, the transactional mode selector of DynTM-D often predicts that TxID-2 in yada is prone to overflow, and a majority of its instances are executed in eager mode. This leads to the aforementioned stalls that DynTM-D exhibits in yada, seen in Fig. 5. Whether futile or useful, these stalls limit available concurrency and are responsible for a performance degradation with respect to DynTM-O. On the other hand, DynTM-D also tracks the number of retries (aborts) suffered by each transaction. In DynTM-D, transactions that are predicted to suffer multiple lazy-mode aborts are executed in eager mode. This heuristic is based on the fact that DynTM-D gives priority to eager writers whenever they conflict with lazy transactions (*eager early write* and *eager late write* [12]). Nonetheless, we find this heuristic counterintuitive, as lazy conflict resolution inherently allows more concurrency in high contention situations, as we have seen in Fig. 4. In benchmarks with moderate to high contention, the predictor adapts the behaviour of DynTM-D for contended transactions to resemble DynTM-E. This is positive in intruder, where lazy-mode transactions suffer due to the increased congestion at the directory, but it results in a performance drop in genome, compared to DynTM-O. Finally, for low contention benchmarks that do not suffer overflows (vacation or ssca2), DynTM-D basically behaves like DynTM-O, as it chooses to execute all or most transactions lazily.

Overall, we find that the heuristics used by DynTM-D to adapt its behaviour depending on the past history are not particularly robust, and in certain cases the predictor may unintendedly opt for a less profitable execution mode. Thus, we believe that embedding these heuristics in silicon is not sound because it may open up the door to unexpected pathologies.

7 Conclusion

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 process, our proposal—the ZEBRA HTM system—manages to bring together the good aspects of both eager and lazy designs with very modest 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, as well as lazy systems when contention dominates. ZEBRA 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. Furthermore, in comparison with state-of-the-art hybrid-policy designs, this paper quantitatively shows that our data-centric approach not only entails significantly less complexity than previous proposals but also surpases them in overall performance. Hence, our study demonstrates that the ZEBRA HTM design is the most cost-effective solution in the area of low complexity hybrid-policy HTM systems.

Acknowledgment

This work has been supported by the Swedish Foundation for Strategic Research under grant RIT10-0033, as well as by the Spanish MINECO under grant TIN2012-38341-C04-03.

References

[1] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie, ''Unbounded Transactional Memory,'' in *Proc. 11th Symp. High-Perform. Comput. Architect.*, 2005, pp. 316-327.

[2] J. Bobba, K.E. Moore, L. Yen, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood, ''Performance Pathologies in Hardware Transactional Memory,'' in *Proc. 34th Int'l Symp. Comput. Architect.*, 2007, pp. 81-91.

[3] C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, ''STAMP: Stanford Transactional Applications for Multi-Processing,'' in *Proc. IEEE Intl. Symp. Workload Characterization*, 2008, pp. 35-46.

[4] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, ''Bulk Disambiguation of Speculative Threads in Multiprocessors,'' in *Proc. 33rd Int'l Symp. Comput. Architect.*, 2006, pp. 227-238.

[5] H. Chafi, J. Casper, B.D. Carlstrom, A. McDonald, C.C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, ''A Scalable, Non-Blocking Approach to Transactional Memory,'' in *Proc. 13th Symp. High-Perform. Comput. Architect.*, 2007, pp. 97-108.

[6] A. Dragojevic and R. Guerraoui, ''Predicting the Scalability of an STM,'' in *Proc. 5th ACM SIGPLAN Workshop TRANSACT*, 2010.

[7] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, ''Transactional Memory Coherence and Consistency,'' in *Proc. 31st Int'l Symp. Comput. Architect.*, 2004, pp. 102-113.

[8] T. Harris, J.R. Larus, and R. Rajwar, *Transactional Memory* 2nd., San Rafael, CA, USA: Morgan & Claypool, 2010.

[9] M. Herlihy and E. Koskinen, ''Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects,'' in *Proc. 13th ACM SIGPLAN Symp. Principles Pract. Parallel Programm.*, 2008, pp. 207-216.

[10] M. Herlihy, J. Eliot, and B. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' in *Proc. 20th Int'l Symp. Comput. Architect.*, 1993, pp. 289-300.

[11] M. Lupon, G. Magklis, and A. González, ''FASTM: A Log-Based Hardware Transactional Memory with Fast Abort Recovery,'' in *Proc. 18th Int'l Conf. Parallel Architect. Compilation Techn.*, 2009, pp. 293-302.

[12] M. Lupon, G. Magklis, and A. González, ''A Dynamically Adaptable Hardware Transactional Memory,'' in *Proc. 43rd Int'l Symp. Microarchitect.*, 2010, pp. 27-38.

[13] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, ''Simics: A Full System Simulation Platform,'' *Computer*, vol. 35, no. 2, pp. 50-58, Feb. 2002.

[14] M. Martin, M. Hill, and D. Sorin, ''Why on-Chip Cache Coherence is Here to Stay,'' *Commun. ACM*, vol. 55, no. 7, pp. 78-89, July 2012.

[15] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood, ''Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset,'' *Comput. Architect. News*, vol. 33, no. 4, pp. 92-99, Nov. 2005.

[16] A. McDonald, J. Chung, D.C. Brian, C.C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, ''Architectural Semantics for Practical Transactional Memory,'' in *Proc. 33rd Int'l Symp. Comput. Architect.*, 2006, pp. 53-65.

[17] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood, ''LogTM: Log-Based Transactional Memory,'' in *Proc. 12th Symp. High-Perform. Comput. Architect.*, 2006, pp. 254-265.

[18] A. Negi, R. Titos-Gil, M.E. Acacio, J.M. Garcia, and P. Stenstrom, ''π-TM: Pessimistic Invalidation for Scalable Lazy Hardware Transactional Memory,'' in *Proc. 18th Symp. High-Perform. Comput. Architect.*, 2012, pp. 1-12.

[19] S.H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian, ''Scalable and Reliable Communication for Hardware Transactional Memory,'' in *Proc. 17th Int'l Conf. Parallel Architect. Compilation Techn.*, 2008, pp. 144-154.

[20] R. Rajwar and J.R. Goodman, ''Transactional Lock-Free Execution of Lock-Based Programs,'' in *Proc. 10th Int'l Symp. Architect. Support Programm. Lang. Oper. Syst.*, 2002, pp. 5-17.

[21] N. Shavit and D. Touitou, ''Software Transactional Memory,'' in *Proc. 14th ACM Symp. Principles Distrib. Comput.*, 1995, pp. 204-213.

[22] A. Shriraman, S. Dwarkadas, and M.L. Scott, ''Flexible Decoupled Transactional Memory Support,'' in *Proc. 35th Int'l Symp. Comput. Architect.*, 2008, pp. 139-150.

[23] R. Titos-Gil, A. Negi, M.E. Acacio, J.M. Garcia, and P. Stenstrom, ''ZEBRA: A Data-Centric, Hybrid-Policy Hardware Transactional Memory Design,'' in *Proc. 25th Int'l Conf. Supercomput.*, 2011, pp. 53-62.

[24] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, ''EazyHTM: Eager-Lazy Hardware Transactional Memory,'' in *Proc. 42nd Int'l Symp. Microarchitect.*, 2009, pp. 145-155.

[25] L. Yen, J. Bobba, M.R. 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. 13th Symp. High-Perform. Comput. Architect.*, 2007, pp. 261-272.

Rubén Titos-Gil received the MS and PhD degrees in computer science from the University of Murcia, Spain, in 2006 and 2011, respectively. He is currently Postdoctoral Research Associate at Chalmers University of Technology, Sweden. His research interests lay on the fields of parallel computer architecture and programming models, including synchronization, coherence protocols and memory systems. He was a Spanish MEC-FPU Fellowship recipient from 2007 to 2011. He is a member of the IEEE.

Anurag Negi is a PhD candidate at Chalmers University of Technology. His research interests include computer architecture, multicore memory hierarchies and support for transactional memory in hardware. In the past he has worked for Intel and Conexant Systems in India. He has bachelor's and master's degrees in electrical engineering from the Indian Institute of Technology, Madras, India. He is a member of the IEEE.

Manuel E. Acacio is an Associate Professor of computer architecture and technology at the University of Murcia, Spain, where he leads the Computer Architecture & Parallel Systems (CAPS) research group. He has served as a committee member of important conferences, ICPP and IPDPS among others, and is currently an Associate Editor of IEEE Transactions on Parallel and Distributed Systems (TPDS). Dr. Acacio is actively working on prediction and speculation in multiprocessor memory systems, synchronization in CMPs, power-aware cache-coherence protocols for CMPs, fault tolerance, and hardware transactional memory systems. He is a member of the IEEE.

José M. García received the MS degree in electrical engineering and a PhD degree in computer engineering both from the Technical University of Valencia in 1987 and 1991, respectively. He is professor of Computer Architecture at the Department of Computer Engineering, and also Head of the Parallel Computer Architecture Research Group. His current research interests lie in high-performance power-efficiency coherence protocols for Chip Multiprocessors (CMPs), interconnection networks and general-purpose computing computing on GPUs. He has published more than 150 refereed papers in different journals and conferences in these fields. Prof. García is member of HiPEAC, the European Network of Excellence on High Performance and Embedded Architecture and Compilation. He is also member of several international associations such as the ACM and IEEE.

Per Stenstrom is professor at Chalmers University of Technology. His research interests are in high-performance computer architecture. He has authored or co-authored three textbooks and more than a hundred publications. He has been program chairman of the IEEE/ACM Symposium on Computer Architecture, the IEEE High-Performance Computer Architecture Symposium, and the IEEE Parallel and Distributed Processing Symposium. He is an editor of ACM TACO and Associate Editor-in-Chief of JPDC. He is a Fellow of the ACM and the IEEE and a member of Academia Europaea and the Royal Swedish Academy of Engineering Sciences.