View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

ELSEVIER
Available online at www.sciencedirect.com
SciVerse ScienceDirect
Procedia Computer Science 18 (2013) 270 – 279
Procedia
Computer Science

International Conference on Computational Science, ICCS 2013

EcoTM: Conflict-Aware Economical Unbounded Hardware Transactional Memory

Saša Tomić[a], Ege Akpinar[a], Adrian Cristál[a], Osman Unsal[a], Mateo Valero[a]

[a]*BSC-Microsoft Research Center*

Abstract

Transactional Memory (TM) is a promising paradigm for parallel programming. TM allows a thread to make a series of memory accesses as a single, atomic, transaction, while avoiding deadlocks, livelocks, and other problems commonly associated with lock-based programming. In this paper we explore Hardware support for TM (HTM). In particular, we explore how HTM can efficiently support transactions of nearly unlimited size.

For this purpose we propose EcoTM, an economical unbounded HTM that improves the efficiency of conflict detection between very large transactions by activating conflict-detection logic only for potentially-conflicting locations: shared and speculatively modified. EcoTM detects the potentially-conflicting locations automatically, without any program annotations.

We evaluate EcoTM performance by comparing it with ideal-lazy HTM, unbounded eager HTM with perfect signatures, and LogTM-SE. Our evaluations show that EcoTM has similar performance as the ideal-lazy HTM, 8.8% better than the eager-perfect HTM, and over 35.7% better than LogTM-SE, on the average.

Keywords: Hardware; Transactional Memory; HTM; TM; Parallel; Concurrent; High-Performance

1. Introduction

Writing thread-safe, efficient, deadlock-, livelock-, and data-race-free libraries and programs is still difficult, in spite of having multi-core processors on almost every desktop today. Hardware Transactional Memory (HTM) is often seen as a solution for writing efficient multi-threaded applications, since HTM does not have many of the the problems typically faced while writing lock-based libraries and programs. In this paper, we propose an HTM mechanism that can efficiently detect conflicts between large or small transactions.

Whereas commonly proposed HTMs are efficient, they typically work with transactions up to a few cache lines large (i.e. best-effort HTMs). To support larger transactions, some HTM proposals tend to either use per-cache-line metadata for tracking read-sets and write-sets, or they use Bloom-filter signatures to approximate a transaction's read-set and write-set in a finite-size data structure. Neither of these approaches is ideal. Per-cache-line metadata introduces substantial overheads in the caches, and storing this metadata for the lines evicted from private caches can be difficult. On the other hand, Bloom-filter signatures can introduce false-conflicts between unrelated transactions, by giving a false-positive hit in the signature, i.e., by indicating that a signature has the address even though the address has not been inserted. Furthermore, studies have shown that the false-conflict rates seen in practice are substantially higher than those anticipated by theoretical studies.

*Saša Tomić. Tel.: +34-9340-11841
E-mail address: sasa.tomic@bsc.es.

1877-0509 © 2013 The Authors. Published by Elsevier B.V. Open access under CC BY-NC-ND license.
Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.190

The biggest problem with the current HTMs is that they treat all cache lines in almost the same way – a cache line may create a conflict at any time. The consequence of such approach is that an HTM needs to track an impressively large number of cache lines. However, the truth is that most of the cache lines never create a conflict. Many of the lines are either shared between transactions but read-only, or thread-local, meaning that only one transactions uses the line at a time.

A location creates a conflict (is conflicting) if a location is written by a transaction, while the same location is read or written by other transactions. Other types of locations are considered non-conflicting, for example: (1) a locations read by multiple transactions, (2) a location that is exclusive to (or owned by) a single transaction (may be read and written), or (3) a location that changes the owner transaction over time. The mechanism classifies memory locations dynamically, and automatically, and transparently to the programmer, based on the run-time accesses. In addition, even if a conflict occurs on a previously non-conflicting location, the conflicting transactions are handled correctly.

This paper presents a novel and economical method to identify the non-conflicting cache lines, in order to improve the efficiency of conflict detection in HTMs. The key idea is to dynamically (during run-time) identify and separate the uncommon locations that create genuine conflicts between transactions, and to manage them by a small hardware table that detects conflicts precisely. Conversely, the most common (non-conflicting) locations are managed in a distributed manner, with only 2 bits of associated metadata.

This paper makes the following contributions:

- Presents EcoTM (**Eco**nomical**TM**), and a novel generic protocol for conflict detection that can be applied to HTMs with eager, lazy, and eager-lazy conflict management. The protocol adds only 2 bits metadata per cache line, whereas a naive unbounded lazy HTM adds 64 bits per line in a 32-core system, and a state-of-the-art unbounded HTM, TokenTM, adds 16 bits per line.
- EcoTM conflict-detection protocol avoids the false conflicts introduced by many other unbounded HTMs that employ Bloom-filter signatures. False conflicts in these signatures are much more likely with larger transactions, which we anticipate in future TM workloads.
- An evaluation indicates that EcoTM performs significantly better than the state-of-the-art HTMs. It performs $\sim$ 35.7% faster on average than LogTM-SE, a Bloom-filter state-of-the-art unbounded HTM, and $\sim$ 8.8% faster than TokenTM over all benchmark configurations. The performance advantage is more significant over configurations with larger transactions.

The paper is organized as follows. In Section 2, we present the mechanism that automatically and dynamically identifies and separates the conflicting from non-conflicting lines. In Section 3, we present the basic EcoTM architecture, which leverages the separation of conflicting and non-conflicting lines to provide a well-performing unbounded eager-lazy HTM. In Section 4, we describe how EcoTM handles the overflowed transactions and how the mechanism can be extended to the commodity CMP systems with limited directories (e.g., a directory only in L2 caches). In Section 5, we evaluate the performance of EcoTM and analyze its performance sensitivity. Finally, we discuss the related work in Section 6 and conclude in Section 7.

2. Detecting Conflicting Cache Lines

In this Section, we will first describe a naive implementation of an unbounded lazy HTM, that has good performance but high overheads. We will then present our work on eliminating these overheads.

In this paper, we consider a Chip-MultiProcessor (CMP) system with coherent caches and a memory directory. On such system, a naive implementation of unbounded lazy HTM can be straightforward and easy to understand. Each transactional (speculative) read and write is marked in the directory, in the special conflict-detection metadata. In the metadata, a (transactional) read of a processor core is marked by 1 bit, and a write by 1 other bit. This means that in a 32-core system, the transactional metadata of 64-bit is associated with each cache line.

In such system, a conflict is detected when a write bit of a core is set while other cores also have a write or read bit set for the same line. For example, a transaction (Tx) 1 has a write bit set, and after Tx 2 gets the read bit set; in this case, a conflict is detected between Tx 1 and Tx 2.

Although not complex and therefore being relatively easy to verify and implement in real hardware, such system would require a prohibitively large amount of additional metadata. In addition, that amount of metadata grows

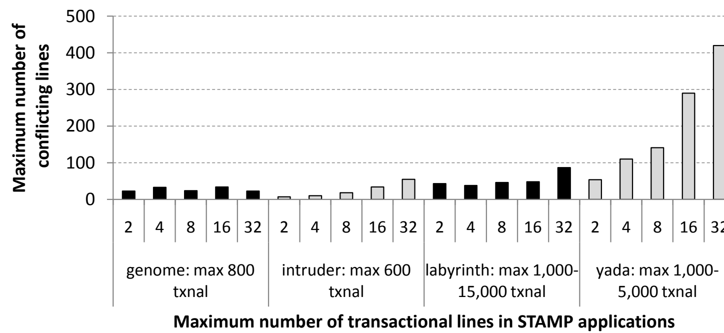


Fig. 1: Conflicting cache lines are very few (vertical axis), typically only 10s of lines, whereas there are many lines touched by transactions at some points of execution (horizontal axis), often more than 1000s of lines. Both the number of transactional and conflicting lines grow with more parallel threads, but the conflicting lines are always significantly fewer than transactional lines. The results do not show the total (cumulative), but the maximum number of lines at any point of execution. 51 other configurations have almost no conflicting lines and we do not show them here.

linearly with the number of processor cores, making the approach unsuitable for future processor generations that may have 100s or 1000s of cores.

To reduce the amount of metadata, we observed and characterized the behavior of the transactional applications in a common TM benchmark suite – STAMP [1]. The result of our findings is shown in Figure 1. We measured the number of conflicting and non-conflicting cache lines at every transaction commit and abort, and in the figure we plot the maximum number found in each workload. On horizontal axis we plot the number of all transactional (conflicting plus non-conflicting) lines, and on the vertical axis only the conflicting lines. In 51 out of 60 workloads, there are less than 30 conflicting lines at any instance of time, and we completely exclude them from the figure. From this figure, we can conclude that an HTM only *needs* to handle a modest number of conflicting lines, although many existing HTM proposals treat equally as many as 15,000 transactional lines.

To detect when a cache line becomes conflicting, each line tracks the history of transactional accesses, and encodes the state of the history it in the small metadata. A line may be in one of the following states: (1) non-transactional, or a transactional and one of the following (2) read-only, (3) read-write but exclusive, or (4) read-write and shared (i.e. conflicting). The four history states can be encoded with metadata of only two bits, named Quick Conflict Filter (QCF). This metadata is associated with each cache line (block) in the memory hierarchy. It is physically stored and updated similarly to the way that parity (ECC) bits, i.e. distributed, with regular cache blocks (lines).

Each cache line starts as non-transactional. On a transactional access, a line becomes (1) shared and `read-only` or (2) `exclusive`. The line will change to the `read-only` state if it is shared and read by several transactions. If no other transaction uses the line, i.e. if the line is accessed and used only by (this) one transaction, a line will become `exclusive`. On further transactional accesses, a line in the `read-only` or `exclusive` state may transition to the `conflicting` state – which indicates that a line is shared between transactions and read-write. A line returns to the initial non-transactional state, i.e. the history of transactional accesses (the metadata) is cleared, only after a regular (non-transactional) write to the line. A regular write also aborts all transactions that may have accessed the line, so it is completely safe to clear the metadata at this point.

A line in conflicting state needs to be treated differently from regular lines – we need to analyze the common line metadata (a list/bit-vector of line sharers) and to construct the extended conflict-detection metadata, that can be used to precisely detect conflicts between transactions. This construction of extended conflict-detection metadata, and the conflict-detection itself are explained later in this paper, in Section 3.1.

3. EcoTM Architecture

In this Section, we outline the baseline CMP architecture for EcoTM, the extensions to the processor cores, caches, and the directory.

Figure 2 presents an overview of the EcoTM architecture. EcoTM includes the common HTM hardware: (1) register file snapshot, (2) speculative R/W flags in L1 cache lines, and (3) TX state. A snapshot of register file is

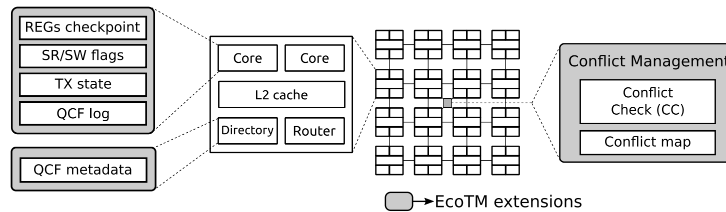


Fig. 2: Generic Chip Multi-Processor (CMP) architecture used as the baseline for EcoTM. The processor cores locally manage non-conflicting lines. The directory banks, using the QCF metadata, separate the common non-conflicting from the uncommon conflicting lines. Conflicting lines are managed by a dedicated hardware logic, reducing the conflict detection traffic on the interconnects.

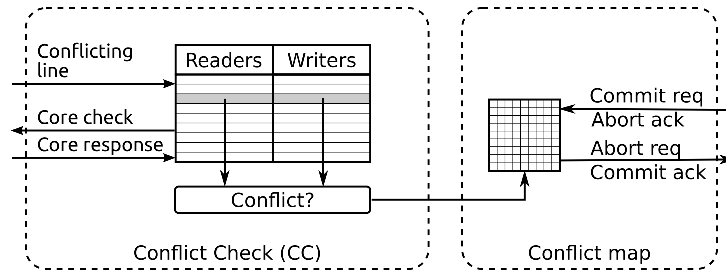


Fig. 3: Conflict-management hardware in EcoTM. CC precisely detects conflicts, and marks them in Conflict map.

made when a transaction begins, and restored on abort. The SR (Speculative-Read) and SW (Speculative-Write) flags mark speculative accesses, and avoid repeating conflict detection for the same line in the same transaction. Finally, the TX state marks whether a transactions is active or not, and whether it is overflowed (does not fit into L1 caches) or not.

EcoTM-specific extensions are: (1) the Quick Conflict Filter (QCF) metadata, (2) the optional QCF log, and (3) the conflict-management hardware logic in the directory. As explained in the previous section, QCF metadata and its associated hardware logic are handled in directory banks. The QCF logic detects when a line becomes conflicting, completely avoiding conflict detection for the majority of lines, which are non-conflicting. When a line used by a transaction is evicted from a private cache, a transaction becomes overflowed and future conflict detection much be done conservatively, assuming that a transaction accessed the line, which may create false conflicts. To prevent this, we can add QCF log to EcoTM. Using this structure, we can revert the QCF changes when transaction terminates, achieving precise conflict detection once again. The mechanism is explained in more details in Section 4.2.

For better efficiency and scalability, EcoTM performs the most common operations on-core, without communicating with the directory or with other cores. For example, the following operations are core-local: (1) L1 cache reads, and (2) L1 cache writes, if a line is in exclusive mode. In other cases, EcoTM informs the directory of the access. When the access message reaches the directory, the QCF checks for the common case, when the access is non-conflicting, and in that case immediately responds to the core. If QCF detects the uncommon case when the access creates a conflict, the request is forwarded to the conflict-management logic, named Conflict Check (CC).

While previous eager-lazy HTMs detect conflicts in a distributed manner [2, 3], this implies sending multi-cast messages for conflict detection. Instead, by using a dedicated conflict-management hardware logic at a directory level, we replace multi-cast with more efficient core-to-directory point-to-point messages. In our evaluations, we have not observed contention on the CC logic, since it handles only the uncommon conflicting lines.

The hardware logic for conflict-management is illustrated in Figure 3. CC is a fixed- and limited-size table that handles only conflicting lines. A CC entry for each core stores: 1 bit to mark a Speculative Read (SR) and 1 to mark a Speculative Write (SW). On a 32-core processor, each CC entry has 64 bits. Since a CC entry has a complete list of speculative accesses, EcoTM detects conflicts the same way a naive lazy HTM would, as we explained earlier, in Section 2. A speculative write bit marks a conflict with all other line accessors with SR or SW bit set. After CC detects a conflict, it is marked in the conflict map.

Conflict map is a $(N-1)^2$ bit-matrix (N is the number of processor cores) of conflicts between transactions on these cores. Each transaction has one bit-vector that represents the conflicts with other transactions. For example,

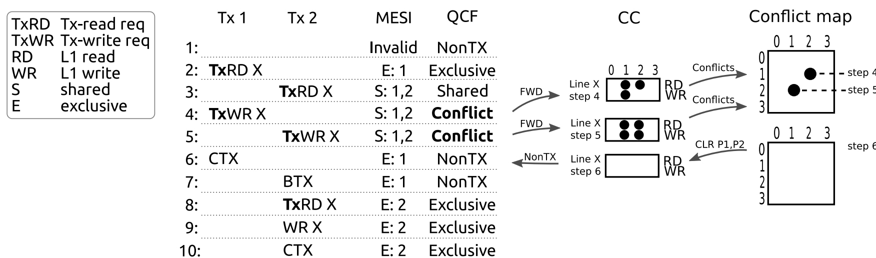


Fig. 4: Execution of conflicting bounded transactions. The conflicts are precisely detected in CC, and marked in the conflict map. On commit, the CC entry clears the conflicting state of the QCF entry.

if Tx 1 entry has set bit 2, this means that if Tx 1 wants to commit, it needs to abort the Tx 2.

3.1. Managing conflicting transactions

When a line does not have an entry in the CC table, an entry is initialized from (1) the directory information and (2) by querying line sharers. A CC entry may be invalidated at any time, for example, because of limited capacity of the CC table, or when a line stops being conflicting. A CC entry for the same line can be reconstructed (initialized) later, if necessary.

To initialize an entry, CC queries the line sharers (processor cores). If a sharer has overflowed transaction, it is assumed to have the line in both read and write set. Although this might create false conflicts, our evaluations indicate that it rarely happens, even with small L1 caches. If the common case, when a transaction is not overflowed, the entry value is initialized with the values SR and SW bits in the L1 cache of each sharer.

A transaction with conflicts needs to request a commit permission from the conflict map, i.e., such transaction cannot commit core-locally. The conflict map has simple functionality and responds fast and in constant time. On a commit request, the conflict map checks if the transaction is valid. If it is valid, the map requests aborts from all conflicting transactions of the committing transaction. The abort request is a "fire-and-forget", since it cannot be rejected. The map therefore immediately sends an acknowledgement to the committing core, and flash-clears all bits related to the committing transaction.

After commit, L1 cache write backs the committed data lazily, when it: (1) evicts the line (due to the capacity constraint), (2) reduces the access mode to shared or invalid (for example, other core requests line access), or (3) speculatively modifies the line in a new transaction. Functionally, commit behaves similarly to the memory fence [4, 5, 6]. That is, memory operations may be reordered before the commit, but not across the commit operation. The processor core stalls writing-back (retiring) instructions [7] that follow the commit, until it gets a response from the directory.

If a transaction with conflicts aborts, has to inform the conflict-management hardware. The conflict map clears all bits related to the aborted transaction, which prevents false (unnecessary) aborts of transactions that might execute later on the same core.

Example of conflict management. Figure 4 illustrates an example of execution of conflicting transactions Tx1 and Tx2. In step 4, the QCF marks the line as conflicting when Tx1 speculatively writes to the line. A CC entry is initialized for the line, and a conflict from Tx1 to Tx2 is marked in the conflict map. A second conflict is in step 5, when Tx2 also speculatively writes to the line. The CC entry is updated, and a conflict from Tx2 to Tx1 is marked. When Tx1 commits in step 6, the conflict map aborts Tx2, and flash-clears the Tx1 and Tx2 columns in CC. Since the CC detects that the line is no longer conflicting, it updates the QCF state of the line to NonTX. From step 7 to step 10, the Tx2 executes a non-conflicting transaction.

4. Precise Conflict Detection with Limited Hardware Resources

In this Section, we describe the execution of overflowed transactions, i.e. the transactions that do not fit private caches, and the EcoTM support for the commodity Chip-MultiProcessor (CMP) systems, which typically have limited directory size – i.e. with a directory only at the level of shared L2 caches.

4.1. Conflict management for overflowed transactions

To ensure that conflicts are detected even after a transactional line is evicted the L1 cache, the directory should keep the L1 cache in the list of line sharers. We apply the technique known as "silent evictions" to the transactional lines [8]. Silent evictions are currently employed by common MESI protocols to reduce the bandwidth overhead, where shared and exclusive lines do not trigger directory messages and updates.

After silent eviction, future queries on whether the line is in the L1 cache are conservatively answered positively, possibly introducing some false conflicts. For the TShared QCF state we assume that all sharers read the line, and for the TExclusive or TConflicting, we assume that the sharers both read and modified the line. However, if the line has an entry in the CC table, the correct information is preserved even upon the eviction from the L1 cache. In Section 5 we sensitivity of EcoTM performance to the number of entries in CC.

4.2. Logging QCF changes

QCF state converges to the correct state over time, as future execution re-checks sharers and updates the list of speculative accesses. However, when a line is evicted from a sharer, stale QCF states TExclusive or TConflicting may indicate a conflict of the overflowed sharer with other active transactions. A line entry in CC has the precise sharing information (even for overflowed lines), but CC entries may be evicted.

As a solution, we may use an undo-log of QCF, that will store only overflowed lines in a TExclusive and TConflicting state. The hardware appends to the log, and a software handler clears the log when overflowed transaction terminates, and resizes the log if necessary. The logs are efficient, they store only the line addresses (not the data) of overflowed lines, they are organized as stacks and are stored in a cacheable thread-private memory.

Note that the QCF undo-logs are more efficient than common LogTM [8] logs, since: (1) EcoTM logs only the overflowed lines, and (2) EcoTM does not log any data. By not logging data, EcoTM adds only 8 bytes per entry, instead of 64+8 bytes in LogTM log. EcoTM does not have to log data because it stores speculative values in another structure, overflow buffer, explained in Section 4.3.

4.3. Data management for overflowed transactions

Although EcoTM supports any commonly used mechanism for handling overflowed data, we here assume the use of a software-supported Overflow Buffer (OB), as proposed by Shriraman et al. [3]. The OB is organized as a simple per-thread hash table in virtual memory, and accessed by the OB controller that sits on the private cache miss path. On private cache misses, the request is redirected to the OB and handled in hardware. The commit-time write-backs are performed by the OB controller, and occur in parallel with other useful work by the processor. In Section 5 we show that the OB latency does not significantly affect the execution time.

4.4. Support for context switching and interrupts

Whereas an abort may be a costly operation in eager HTMs, EcoTM has fast aborts. While future workloads might call for a different approach, enabling migration of long-running transactions is likely to be extremely complex, particularly in systems combining the use of operating systems and virtual machine monitors. The execution of an aborted transaction from scratch is likely to be faster from many proposed transaction migration mechanisms, particularly for current workloads where transactions execute less than 10 microseconds, and the interrupts are typically generated every 10 milliseconds.

Despite the current design decision, EcoTM can easily be extended to fully support context switches and transaction migration. One approach could be to save the full speculative state before context switch in software, and to re-applying it after the transaction migrates to a different processor core.

4.5. EcoTM on Systems with Limited Directory Size

Modern Chip-MultiProcessors (CMPs) have limited directory size, e.g. a directory only in L2 or L3 caches. Because of this, the history of speculative accesses could be lost once a line is evicted from the top-level cache. This could cause future speculative accesses to the line to miss real conflicts. To prevent this, the QCF metadata (2 bits per line) can be saved to physical memory together with an evicted cache block, and simply restored when the block is requested again by the directory.

The conflict-detection metadata can be preserved either in the ECC area, as has been proposed by TokenTM [9], or by increasing the row size in DRAM chips – something already done between different DRAM generations.

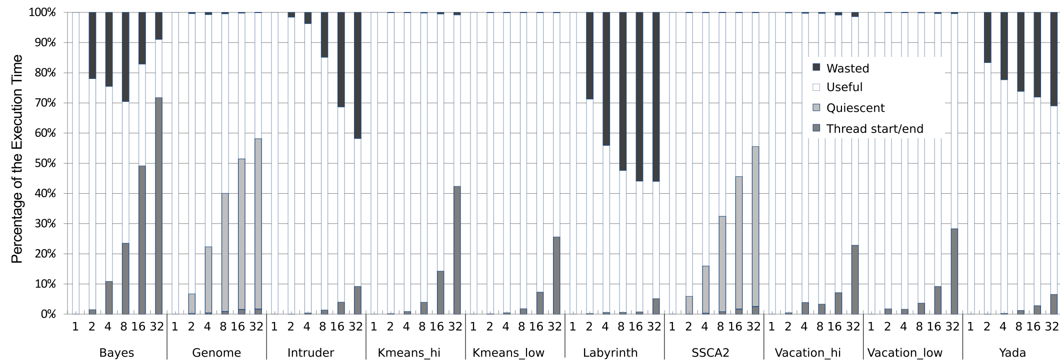


Fig. 5: A breakdown of the total execution time in STAMP applications, for 1-32 processor cores

5. Evaluation

We evaluated EcoTM using a full-system simulator M5, that we use to simulate a system with Alpha architecture [10]. We replaced the default bus-based cache coherency with a MESI directory-based cache coherency. We simulate in-order cores with a fixed 1 CPI for non-memory related instructions. The memory operations take 1 cycle plus a variable latency returned by the simulator of the memory subsystem. An overview of the simulated hardware is given in Table 1. For the workload, we use STAMP, the most commonly used TM benchmark suite for this type of evaluation.

For fair comparison, we on the same simulator infrastructure we also implemented (i) an eager HTM with Bloom-filter signatures, LogTM-SE [11], (ii) an unbounded eager HTM with perfect signatures, and (iii) an ideal-lazy HTM. We ported the LogTM-SE code to M5, from its original publicly available source code (implemented with Simics and GEMS), and verified that the performance is comparable with the original.

Processor	1-32 cores, single-issue, single-threaded, 1 CPI
L1 Cache	32KB private, 2-way, 64-byte blocks, 2 cycles, write-back
L2 Cache	8MB banked, 8-way, 64-byte blocks, 32 cycles, write-back
Directory	Bit-vector of sharers, 8-cycle access latency
Memory	4GB, 500 cycles latency
Interconnect	2D Mesh, 4-cycle latency
CC table	256 directly mapped entries (unless otherwise noted)
Conflict map	40-cycle access latency
Log latency	8-cycle latency for removing one entry

Table 1: The hardware configuration

Execution time breakdown. In Figure 5, we show the breakdown of the total execution time for EcoTM executing all STAMP applications over 1–32 processor cores. The total execution time is split into four categories: (1) *Thread start/end* – the time spent in thread synchronization during entering and leaving parallel sections, (2) *Quiescent* – the time spent in quiescent state, which usually occurs after being unable to enter a barrier after several successive retries, (3) *Useful* – the time spent outside of transactions, and in transactional code that successfully commits, and (4) *Wasted* – the time spent in transactional code that is rolled back due to abort.

The breakdown clearly indicates several parallelization problems with some STAMP applications. First, Bayes has unbalanced work between threads, since it uses a non-deterministic work queues. Because of that, we cannot make conclusions based on its execution time. Kmeans and Vacation also spend some time in thread synchronization, but not as much as Bayes. Second, Genome and SSCA2 rely on barriers for synchronizing thread progress, and the barrier synchronization takes more than 50% of the time with 32-core execution. We denote this time as quiescent. Barriers limit the scalability of these two applications. Since these applications have small transactions, and as can be seen in Figure 5, that the amount of wasted work (transaction aborts) is minimal in these applications, the reason for bad scalability is not in HTMs. Finally, wasted work becomes a significant factor for applications

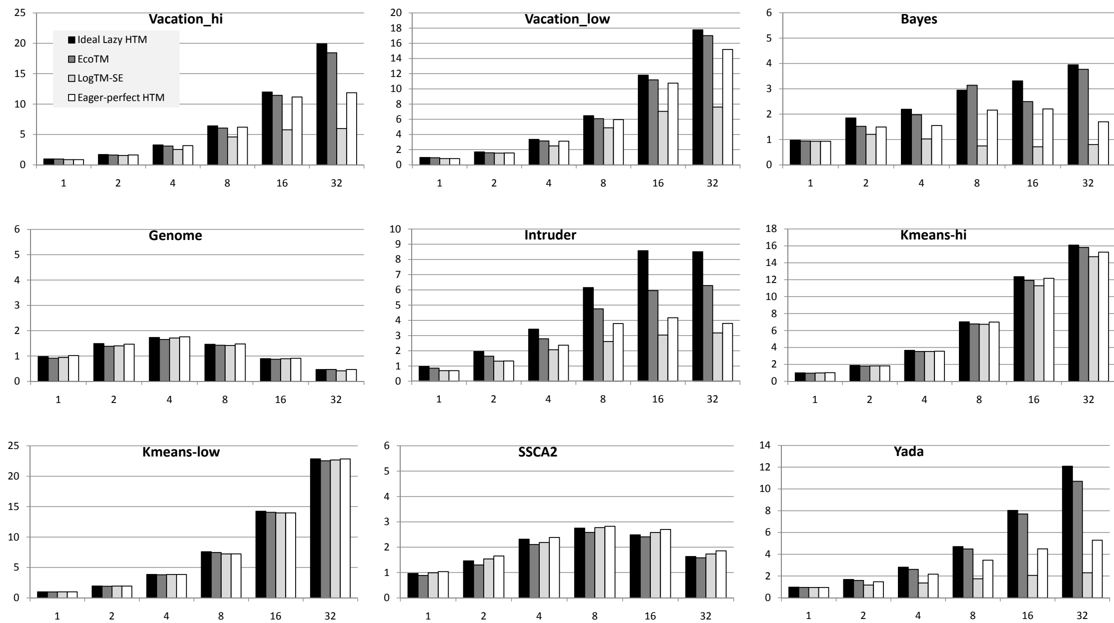


Fig. 6: The speedup of the STAMP TM benchmark suite applications normalized to the sequential execution (no threads or locks). The horizontal axis represents the number of processor cores. The overflow buffer latency is fixed to 100 cycles for EcoTM, and 0 cycles for the ideal-lazy HTM.

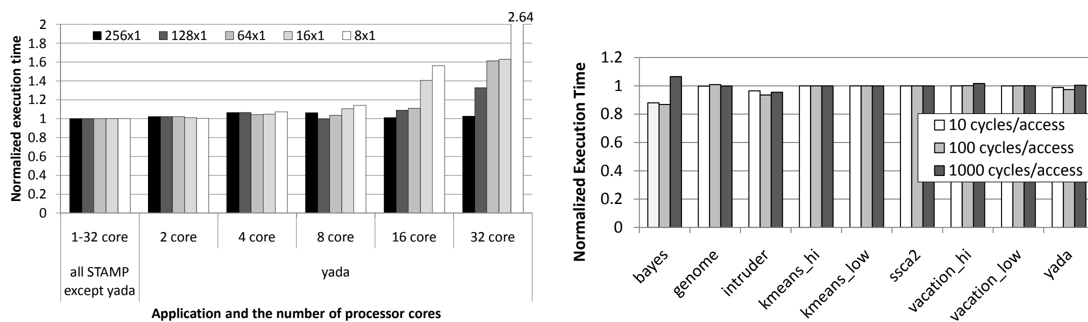
with medium and high contention – Bayes, Intruder, Labyrinth, and Yada, especially in highly-concurrent executions. Labyrinth particularly depends on early release to get any scalability over one [1], so we exclude it from the rest of evaluation since LogTM-SE does not support early release.

Performance and Scalability. Figure 6 presents an evaluation of speedup over the sequential execution of the same application (that does not use threads or locks). Our evaluation includes four HTMs. *Ideal-lazy* HTM has no latencies, i.e. all transactional operations are instant and without overheads. Speculative reads and writes are simple cache reads, and no transactional messages are sent during execution. When transaction commits, it magically detects and resolve conflicts with other transactions and then publishes the speculative changes. *LogTM-SE* is configured with exponential backoff and 2-Kbit Bloom filters with two parallel hash functions, as the authors proposed. Although more advanced signatures were proposed, they have an increased cost of verification and area/energy consumption. Instead of comparing with each of them, we evaluate *perfect or ideal* signatures in LogTM-SE, denoted as *Eager-perfect HTM* (and comparable with TokenTM performance).

In our evaluation, eager and lazy HTMs have comparable performance with small transactions. However, lazy HTMs perform better if: (1) the contention is high, or (2) the transactions are large, since they are better able to extract the small amount of parallelism available in such executions. As a consequence, EcoTM performs better than LogTM-SE and eager-perfect HTM. The geometric mean of the performance improvement of EcoTM for all STAMP applications is 35.7% over LogTM-SE with realistic Bloom filter signatures, and 8.8% over eager-perfect HTM. Over all STAMP configurations, EcoTM is within 7.1% of the ideal-lazy HTM execution time.

Yada has very large transactions and moderate contention, which makes it a good application for evaluating the deficiencies of unbounded HTMs. With 32 threads, EcoTM is only 11.4% slower than ideal-lazy HTM, and has 10.66*x* speedup over sequential execution. With the same configuration, eager-perfect HTMs is 2*x* slower than the ideal-lazy HTM. Similar performance difference between eager and lazy HTMs for this application was also reported by the STAMP authors.

EcoTM overheads. EcoTM overheads come from: (1) logging overflowed cache lines, and (2) conflict management. We evaluate these overheads by comparing the performance with a no-overhead, ideal implementation. EcoTM overall stands very close to the ideal-lazy HTM. The biggest difference is observed for the highly-conflicting Intruder, where the 32-core ideal-lazy HTM is approximately 40% faster than EcoTM, clearly indicating



(a) Even small CC table works well in almost all configurations. The execution time is normalized to the unbounded fully-associative CC table. Even a directly mapped 16-entry (1-way) CC table provides good results for current workloads.

(b) The latency of the Overflow Buffer (OB) barely affects the execution time. The execution time is normalized to the configuration with a 0 cycles per overflow buffer access. We evaluated 32-core EcoTM-Wr-Rd.

Fig. 7: Performance sensitivity of the EcoTM to the CC table configuration and the OB latency.

the conflict management overheads. For other applications, EcoTM is within 5.2% from the ideal-lazy HTM, indicating that the logging overhead is not significant.

Larger CC table (1) reduces the number of false conflicts introduced by overflowed transactions, and (2) reduces the traffic on the interconnects. This improves the execution time and reduces the power consumption. However, larger CC table has higher static power dissipation. Figure 7a shows the sensitivity of the EcoTM performance to the CC configuration. Except Yada, all STAMP applications have the same performance even with only 8 directly mapped CC entries. Although Yada needs 256 directly mapped CC entries, such hardware structure is still very simple and power-efficient.

Overflow buffer (OB) latency has very little impact on EcoTM performance, as can be seen in Figure 7b. We vary the latency of the OB from 0 to 10, 100, and 1000 cycles per OB access. The execution time is normalized to the 0-cycle OB. Although Bayes apparently benefits from a slightly slower OB, its non-deterministic execution prevents us from drawing conclusions from this particular application. The rest of the workloads have almost the same performance for all evaluated OB latencies.

6. Related Work

A common approach to support large transaction is to reduce the amount required metadata – however, some proposals achieve this by restricting the conflict-management policies. For example, TokenTM [9] and LiteTM [12] reduce the amount of metadata to 16 bits (TokenTM) and slightly over 2 bits per cache line (LiteTM), but they support only less performing eager conflict management, and LiteTM even invokes software routines to reconstruct some metadata. EcoTM demonstrates that it is possible to have only 2 bits of metadata per cache line, while supporting better performing lazy and eager-lazy conflict-management policies.

Another common approach for supporting large transactions is to detect conflicts using Bloom-filter signatures, e.g. as in LogTM-SE [11]. FlexTM [3] also uses Bloom filters to accelerate STM in hardware, and supports eager-lazy conflict management. DynTM [13] uses the Bloom filters only the evicted lines, and supports eager-lazy management for regular and eager-only for overflowed transactions. However, the use of Bloom filters hurts the performance with large transactions, while the eager conflict resolution does not perform well under medium and high contention workloads. However, we consider Bloom filters to be inappropriate for the tentative future transactional workloads, since the Bloom-filters need to have an appropriate size for a given number of entries, approximately 10 bits per entry. If the size of the Bloom filter is not appropriate, signature is more likely to falsely detect conflicts, which result in the aborts of actually non-conflicting transactions. With future potentially very large transactions, Bloom-filters are much more likely to falsely detect conflicts, and in this case the unnecessary aborts significantly reduce performance.

Some recent proposals focus on optimizing Bloom-filter signatures. For example, Quislant et al. [14] propose mapping nearby memory locations to the same bits of a signature, whereas Yen et al. [15] improve hash functions, and allow a programmer to define potentially conflicting locations, and insert only these locations into the signature.

In contrast with these proposals, EcoTM provides an automatic mechanism that does not need any effort from a programmer, and that will perform well with any future TM workload, even for extremely large transactions, for as long as the transactions have few real conflicts, which is a fairly realistic expectation.

7. Conclusions

In this paper we presented EcoTM, an unbounded-HTM system that provides precise conflict detection, while it uses a minimal amount of conflict-detection metadata. EcoTM achieves this by distinguishing the uncommon conflicting from the common non-conflicting cache lines automatically, without requiring any annotations from the programmer, and dynamically, during program execution. EcoTM's base hardware mechanisms support all current conflict management strategies: eager, lazy, and eager-lazy. This gives EcoTM both a performance advantage and better cost effectiveness over the alternative unbounded-HTM proposals. We compared the performance of EcoTM with LogTM-SE, a state-of-the-art unbounded HTM proposal, eager-perfect and ideal-lazy HTMs. EcoTM needs less metadata, and provides better performance than eager unbounded HTMs.

A conflict-detection mechanism other than using the Bloom-filter signatures will have an important advantage for future TM workloads, which are likely to have much larger transactions from the existing synthetic TM benchmarks.

References

[1] C. Cao Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization, 2008. doi:10.1109/IISWC.2008.4636089.

[2] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, M. Valero, EazyHTM: Eager-lazy hardware transactional memory, in: MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, ACM, New York, NY, USA, 2009, pp. 145–155. doi:10.1145/1669112.1669132.

[3] A. Shriraman, S. Dwarkadas, M. L. Scott, Flexible decoupled transactional memory support, in: Proceedings of the 35th Annual International Symposium on Computer Architecture, 2008.

[4] C. Intel, Intel IA-64 Architecture Software Developer's Manual, Itanium Processor Microarchitecture Reference for Software Optimization.

[5] C. May, E. Silha, R. Simpson, H. Warren, The PowerPC Architecture: A specification for a new family of RISC processors, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1994.

[6] D. Weaver, T. Germond, The SPARC architecture manual, Citeseer, 1994.

[7] J. E. Smith, A. R. Pleszkun, Implementation of precise interrupts in pipelined processors, in: Proceedings of the 12th annual international symposium on Computer architecture, ISCA '85, IEEE Computer Society Press, Los Alamitos, CA, USA, 1985, pp. 36–44. doi:10.1145/327010.327125.

[8] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, D. A. Wood, LogTM: Log-based transactional memory, in: In proceedings of the HPCA-12, 2006, pp. 254–265.

[9] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, D. A. Wood, TokenTM: Efficient execution of large transactions with hardware transactional memory, in: Proceedings of the 35th Annual International Symposium on Computer Architecture, 2008. doi:10.1109/ISCA.2008.24.

[10] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, S. Reinhardt, The M5 simulator: Modeling networked systems, IEEE Micro 26 (4) (2006) 52–60. doi:10.1109/MM.2006.82.

[11] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, D. A. Wood, LogTM-SE: Decoupling hardware transactional memory from caches, in: HPCA '07: Proc. 13th International Symposium on High-Performance Computer Architecture, 2007. doi:10.1109/HPCA.2007.346204.

[12] S. Ali Raza Jafri, M. Thottethodi, T. N. Vijaykumar, LiteTM: Reducing transactional state overhead, in: Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA-17), 2010.

[13] M. Lupon, G. Magklis, A. González, A dynamically adaptable hardware transactional memory, in: MICRO 43: Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture, 2010.

[14] R. Quislant, E. Gutierrez, O. Plata, E. L. Zapata, Improving signatures by locality exploitation for transactional memory, in: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, Washington, DC, USA, 2009, pp. 303–312. doi:10.1109/PACT.2009.25.
URL http://portal.acm.org/citation.cfm?id=1636712.1637769

[15] L. Yen, S. C. Draper, M. D. Hill, Notary: Hardware techniques to enhance signatures, in: MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2008, pp. 234–245. doi:10.1109/MICRO.2008.4771794.