An Integrated Pseudo-Associativity and Relaxed-Order Approach to Hardware Transactional Memory

ZHICHAO YAN, Huazhong University of Science and Technology
HONG JIANG, University of Nebraska - Lincoln
YUJUAN TAN, Chongqing University
DAN FENG, Huazhong University of Science and Technology

Our experimental study and analysis reveal that the bottlenecks of existing hardware transactional memory systems are largely rooted in the extra data movements in version management and in the inefficient scheduling of conflicting transactions in conflict management, particularly in the presence of high-contention and coarse-grained applications. In order to address this problem, we propose an integrated Pseudo-Associativity and Relaxed-Order approach to hardware Transactional Memory, called PARO-TM. It exploits the extra pseudo-associative space in the data cache to hold the new value of each transactional modification, and maintains the mappings between the old and new versions via an implicit pseudo-associative hash algorithm (i.e., by inverting the specific bit of the SET index). PARO-TM can branch out the speculative version from the old version upon each transactional modification on demand without a dedicated hardware component to hold the uncommitted data. This means that it is able to automatically access the proper version upon the transaction's commit or abort. Moreover, PARO-TM augments multi-version support in a chained directory to schedule conflicting transactions in a relaxed-order manner to further reduce their overheads. We compare PARO-TM with the state-of-the-art LogTM-SE, TCC, DynTM, and SUV-TM systems and find that PARO-TM consistently outperforms these four representative HTMs. This performance advantage of PARO-TM is far more pronounced under the high-contention and coarse-grained applications in the STAMP benchmark suite, for which PARO-TM is motivated and designed.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Parallel processors

General Terms: Design, Performance

Additional Key Words and Phrases: chip multi-processor, hardware transactional memory, pseudo-associative cache, chained directory

ACM Reference Format:
Yan, Z., Jiang, H., Tan, Y., and Feng, D. 2013. An integrated pseudo-associativity and relaxed-order approach to hardware transactional memory. ACM Trans. Architec. Code Optim. 9, 4, Article 42 (January 2013), 26 pages.
DOI = 10.1145/2400682.2400701 http://doi.acm.org/10.1145/2400682.2400701

This work was supported by the National Basic Research 973 Program of China under Grant no. 2011CB302301, Central Universities Fundamental Research Foundation under grant no. 0903005203206, National Natural Science Foundation of China (NSFC) under grant no. 61025008, the US NSF under grants IIS-0916859, CCF-0937993, CNS-1016609, and CNS-1116606.
Authors' addresses: Z. Yan, Wuhan National Lab for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, China; email: zhichao-yan@mail.hust.edu.cn; H. Jiang, Department of Computer Science and Engineering, University of Nebraska-Lincoln; Y. Tan (corresponding author), College of Computer Science, Chongqing University, China; email: tanyujuan@gmail.com; D. Feng (corresponding author), Wuhan National Lab for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, China; email: dfeng@mail.hust.edu.en.

DOI 10.1145/2400682.2400701 http://doi.acm.org/10.1145/2400682.2400701

1. INTRODUCTION

With the emerging and strong trend of many-core computing systems, scaling the software performance at the pace of the New Moore's Law becomes an open and urgent problem, which means that software programmers must carefully design multithreaded programs competing for the shared data in a shared-memory environment. As a result, providing a simple interface of parallel programming to ease the burden on programmers becomes critically important and has attracted a great deal of attention from both the industry and academia. Transactional Memory (TM) has emerged as one of the most promising solutions that can deliver a significant parallel performance boost comparable to that based on the traditional hand-crafted fine-grained locks [Herlihy and Moss 1993]; [Hammond et al. 2004]; [Moore et al. 2006]; [Rossbach et al. 2010]. Recently, several leading IT companies, such as Azul [Click 2009], Sun Microsystems [Chaudhry 2008], IBM [Haring 2011], and Intel [2012], have planted TM in their processor chips, which represents a huge boost to TM in its real-world applicability. In particular, the recent Intel announcement [Intel 2012] to integrate transactional synchronization extensions in its Haswell processor and propose the transactional language constructs for C++ standard specification [Adl-Tabatabai et al. 2011] may have the potential of ushering in a new era of transactional memory.

TMs essentially wrap their computations as transactions that are executed atomically and in isolation, which enhances the programming abstraction. Any memory modifications during a transaction cannot be accepted until the transaction commits. When two or more memory accesses are issued to the same data, of which at least one access comes from a transaction and one of them is a write operation, a transactional conflict occurs. TM systems must detect and resolve this conflict to guarantee the consistency of the shared data. Meanwhile, TMs also must manage the old and new versions of the shared data upon each transactional write operation for abort and commit, respectively. As a result, two important design issues, conflict management and version management, must be carefully considered to obtain a cost-effective design [Harris et al. 2010].

Specifically, conflict management has two main tasks, to detect transactional conflicts during the execution time and to provide a schedule to avoid inconsistency caused by transactional conflicts. Current implementations of conflict management are usually decoupled from version management schemes, suffer from various characteristics of applications, and do not exploit the potentials of the interplay between conflict management and version management.

Version management also has two main tasks that organize transactional modifications alongside the old values until the end of a transaction and merge two versions upon each transactional modification with a consistent memory. Current implementations of version management either require a dedicated buffer/cache to hold the uncommitted data or organize different logs in different levels of the memory hierarchy. While the latter approach suffers from the serial accesses to the shared data and leads to an extended transaction time that can hurt thread-level parallelism, the former relies on a costly extra hardware component considering that it is exclusively used for transactional applications.

Along with conflict and version managements, TM systems incur static and dynamic overheads. The static overhead results from taking a checkpoint at the beginning of a transaction, detecting the transactional conflicts at runtime, maintaining an undo or redo log to hold the uncommitted modifications, and merging or discarding the transactional computation on commit or abort. The dynamic overhead, on the other hand, stems from stalling the conflicted transactions and wasted work spent on the aborted transactions. Recent studies [Bobba et al. 2007] show that the interplay between the static

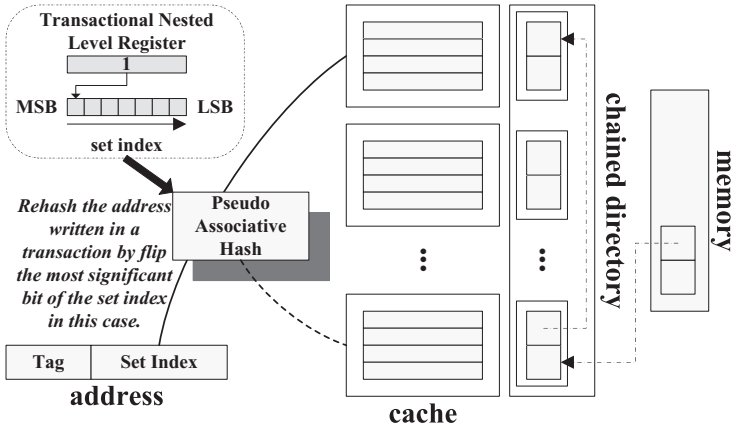


Fig. 1. The mechanism of pseudo-associative and relaxed-order approach.

and dynamic overheads can significantly constrain transactional applications' thread-level parallelism, especially under the high-contention and coarse-grained workloads. Specifically, we observe that the bottlenecks of the existing hardware transactional memory (HTM) systems are mainly rooted in the extra data movements in version management and in the inefficient scheduling of conflicting transactions in conflict management, an urgent and open problem that this article is motivated to address.

More specifically, we propose in this article an integrated pseudo-associativity and relaxed-order approach to HTM, called PARO-TM, as shown in Figure 1, and more details are presented in Figure 3. PARO-TM dynamically associates a pseudo-associative slot in the data cache as the new space to store the transactional modification through a pseudo-associative hash module. Moreover, it utilizes the information in a chained directory to support relaxed conflict management. As a result, PARO-TM can eliminate the extra memory-access latency associated with a conventional transaction due to either commit or abort and expose and exploit more thread-level parallelism in the conflicted transactions.

The rest of this article is organized as follows. We present the background and some experimental and analytical observations to motivate our study in Section 2. Section 3 describes the architecture of PARO-TM and presents its design and implementation details. Section 4 evaluates PARO-TM by comparing it with several state-of-the-art schemes. Finally we conclude the article in Section 5.

2. BACKGROUND AND MOTIVATION

In this section, we present the necessary background information on the state-of-the-art HTMs, with an emphasis on performance issues associated with version and conflict managements, along with some insightful observations to motivate our research.

2.1. Background

Since Herlihy [Herlihy and Moss 1993] introduced the notion of TM as an alternative to traditional synchronization techniques, TM has attracted a great deal of attention in the computer architecture, compiler, programming languages, and operating systems research communities, resulting in the design and implementation of various TM systems. These systems, including hardware TMs, software TMs, and hybrid TMs, aim to ease the difficulty of parallel programming while promoting performance of

multithreaded applications in the many-core era [Harris et al. 2010]; [Guerraoui and Kapalka 2010].

While TM's advantages over the traditional lock mechanism on managing the race blocks have been studied and demonstrated by a large body of research [Rajwar and Goodman 2002]; [Hammond et al. 2004]; [Moore et al. 2006], its scalability faces an increasing challenge brought on by the emerging high-contention and coarse-grained applications [Ansari et al. 2008]; [Minh et al. 2008]; [Kestor et al. 2009]. For example, the geometric mean speedup of the STAMP benchmark suite on a state-of-the-art HTM is roughly 3.6 when executed in a 16-core configuration [Blake et al. 2009]. It may be partly due to the fact that the STAMP benchmark suite is designed for future TM applications that spend most time on high-contention and coarse-grained transactions without handcrafted optimizations on their irregular parallel structures. It indicates that there is an ample room for improvement if the TM technology is to be considered a viable support for parallel programming in the future. Recently, general-purpose microprocessor vendors such as IBM and Intel have integrated limited transactional memory support in their products. In order for the HTM technology to become widely accepted in the many-core era, we must find effective ways to optimize TM's architecture.

2.2. Observation

TM systems organize possible race blocks in transactions and hold the old values and new uncommitted values associated with a transaction until it ends. Existing version management schemes explicitly organize these values in either an undo or redo log, where the lazy schemes [Hammond et al. 2004; Shriraman et al. 2007, 2008; [Minh et al. 2007]; [Dice et al. 2009] buffer the new values in a redo log and the eager schemes [Ananian et al. 2005]; [Moore et al. 2006]; [Yen et al. 2007]; [Blundell et al. 2007]; [Bobba et al. 2008]; [Lupon et al. 2009]] book the old values in an undo log. This is very similar to the prior works on version management for thread-level speculation to use their tricks to differentiate multiple speculative copies of the same cache block in the associative set of a cache [Gopal et al. 1998]; [Garzarán et al. 2003]; [Colohan et al. 2006]. Depending on the types of logs, either the lazy schemes redo the transactional updates on commit, or the eager schemes undo the transactional updates on abort. Once the transactional data overflows from the data cache, HTMs need to allocate pages to hold this overflowed data and usually deallocate the pages at the end of the transaction after merging the proper version of the data to the memory [Ananian et al. 2005]; [Chung et al. 2006b]; [Chuang et al. 2006]. Due to the different methods employed by version management, both the lazy and eager schemes have their own pros and cons.

A lazy scheme combines updating the transactional modifications in its buffer with transactional write operations on the same data path, similar to a write-back operation in the cache coherence protocol. This approach avoids the extra per-store access penalty to store the new value to the redo log, a penalty that is required of an eager scheme as it stores the old value to the undo log before in-place update. A lazy scheme will redo the transactional modifications, on commit, in the write buffer, which may slow down the commit operation especially in a transaction with a lot of transactional modifications. Moreover, if the transactional data overflows from the write buffer, more data movements will be needed, thus widening the isolation windows of the shared variables and hurting the performance. As coarse-grained and high-contention applications are poised to dominate future transactional applications, a small buffer will not be able to handle such coarse transactions with a lot of modifications. On the other hand, an eager scheme will undo all transactional modifications, on abort, in the undo log in a first-in-last-out order to restore to the state before the transaction execution, which makes the expected (i.e., more common) commit operations fast but slows down

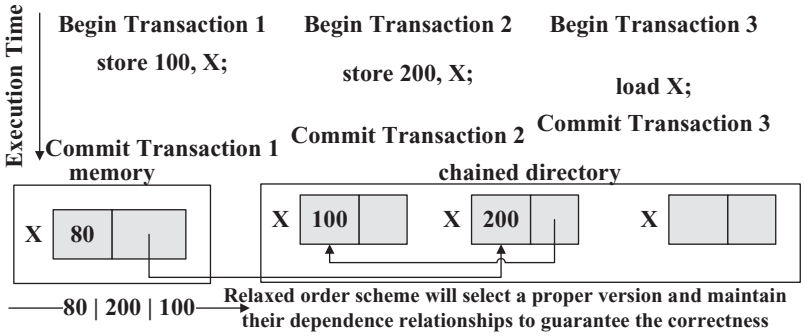


Fig. 2. An example of three conflicting transactions.

the abort operations that may conflict with other accesses to the shared data during its restore process. In any case, there exists a worst-case path in which data must be transferred from the undo/redo log at the end of the transaction, thus widening the exclusive window either on commit or abort. To make things worse, this case will likely induce more conflicted transactions under the high-contention and coarse-grained applications, possibly resulting in a vicious cycle.

Apart from the extra data movements in the version management schemes, the scheduling of the conflicted transactions can significantly affect the performance of HTMs. An inefficient scheduling, for example, will waste a great amount of computation by either aborting conflicted transactions that can actually commit successfully or continually executing conflicted transactions that are eventually aborted due to the cycle-dependent transactional conflicts. Existing methods [Carlstrom et al. 2007] usually rely on a two-phase locking algorithm to provide a fast response to the transactional conflicts but can easily limit the thread-level parallelism because they force the executions of many conflicted transactions to be serialized.

For example, Figure 2 shows 3 different transactions (TXs) concurrently accessing the shared data *X*. If a TM detects conflict eagerly with a requester-win policy, $TX2$ will abort $TX1$ and $TX3$ will abort $TX2$, while $TX1$ will stall $TX2$ and $TX3$ with a requestor-stall policy. Meanwhile, if a TM detects conflict lazily, $TX3$ can commit successfully and $TX2$ will abort $TX1$. The existing two-phase locking algorithms cannot execute without aborting or stalling the access operations. On the other hand, in a relaxed-order scheme, like the one in our proposed PARO-TM, conflicted transactions can continuously execute, provided that it is done efficiently and with minimum overhead.

2.3. Analysis

In pursuit of full exploitation of thread-level parallelism, programmers try to overlap as many transactions as possible, resulting in high-contention and coarse-grained applications. In light of HTM's static and dynamic overheads discussed in Section 1, the interplay between these two kinds of overheads will become more significant under these high-contention and coarse-grained applications. There is a rich body of research on minimizing these overheads. For example, while TCC, UTM, VTM, LogTM, PTM, XTM, RTM, Scalable-TCC, ObjectTM, FasTM, Reconfigurable-TM, SEL-TM and SUV-TM [Hammond et al. 2004]; [Ananian et al. 2005]; [Rajwar et al. 2005]; [Moore et al. 2006]; [Chuang et al. 2006]; [Chung et al. 2006b]; [Shriraman et al. 2007]; [Chafi et al. 2007]; [Khan et al. 2008]; [Lupon et al. 2008] [Lupon et al. 2009]; [Armejach et al. 2011]; [Zhao et al. 2012]; [Yan et al. 2012] focus on reducing static overheads on version management, OneTM, DATM, SBCRHTM, ProactiveTM, EasyTM, DynTM, SON-TM, BFGTS-TM,

and ZEBRA [Blundell et al. 2007]; [Ramadan et al. 2008]; [Titos et al. 2009]; [Blake et al. 2009]; [Tomic et al. 2009]; [Lupon et al. 2010]; [Aydonat and Abdelrahman 2010]; [Blake et al. 2011]; [Titos-Gil et al. 2011] propose different TM architectures to alleviate the dynamic overheads incurred by transactional conflicts. However, most of these methods decouple conflict management from version management and do not exploit opportunities availed by the interplay between conflict management and version management.

We believe that it is important to reduce *both* dynamic and static overheads to fully exploit performance benefits. In general, the reduction of dynamic overheads is correlated to that of static overheads. Our preliminary analysis on transactional applications reveals that each accessed variable owns an exclusive window during the execution, which induces a conflict with another access to the same variable if either access performs a store operation. This exclusive window includes two important time components involving data movements, namely, the time spent on maintaining transactional versions and the time spent on either restoring the old versions on abort or merging the new versions to memory on commit. Thus, under the high-contention and coarse-grained applications, more concurrent accesses lead to a lengthened exclusive window for each shared data due to data movements and cause unnecessary conflicts, which in turn increase the dynamic conflicts and result in more stalled or aborted transactions. Our integrated pseudo-associativity and relaxed-order approach to HTM is an attempt to effectively couple version and conflict managements to reduce data movements in each transaction and exploit the potentials in conflicted transactions.

From the preceding analysis, we conclude that: (1) existing HTMs face a serious scalability challenge especially under the high-contention and coarse-grained applications; (2) reducing data movements in version management not only reduces its static overheads but also reduces the chance for transactions to conflict; and (3) to further reduce the dynamic overheads, a less strict concurrency control algorithm is required to allow more transactions to run concurrently.

3. DESIGN AND IMPLEMENTATION

In this section, we first present the PARO-TM architecture, and then elaborate on its design and implementation from both the hardware and software viewpoints.

3.1. The PARO-TM Architecture

PARO-TM couples a pseudo-associative version management scheme with an eager conflict detection scheme. The latter is supported by a modified relaxed-ordering conflict management scheme instead of the policy that stalls the requester until a cycle is detected. Figure 3 depicts a complete system stack with the runtime environment, transactional library, and hardware components that support PARO-TM functions. While the runtime environment collects the dynamic information to schedule the transactions, the transactional library provides the interface for the basic transactional commands, and the hardware is added to the chip multiprocessor to provide hardware support. We highlight the integration of our pseudo-associative relaxed-order scheme into the existing LogTM-SE [Yen et al. 2007] HTM framework to realize PARO-TM.

PARO-TM's design goal is to maintain both old and new versions of transactional modifications in the same level of memory hierarchy to avoid extra cross-level data movements and schedule the conflicting transactions in a relaxed order. It achieves this goal by reusing the Bloom filter signature [Ceze et al. 2006] to detect transactional conflicts, holding different versions of the shared data in conflict in a cache set and its pseudo-associative counterpart, and employing our relaxed-order scheduling to resolve the transactional conflicts. Specifically, it uses a chained directory to allow multiple transactions to modify the shared data, hold multiple versions, along

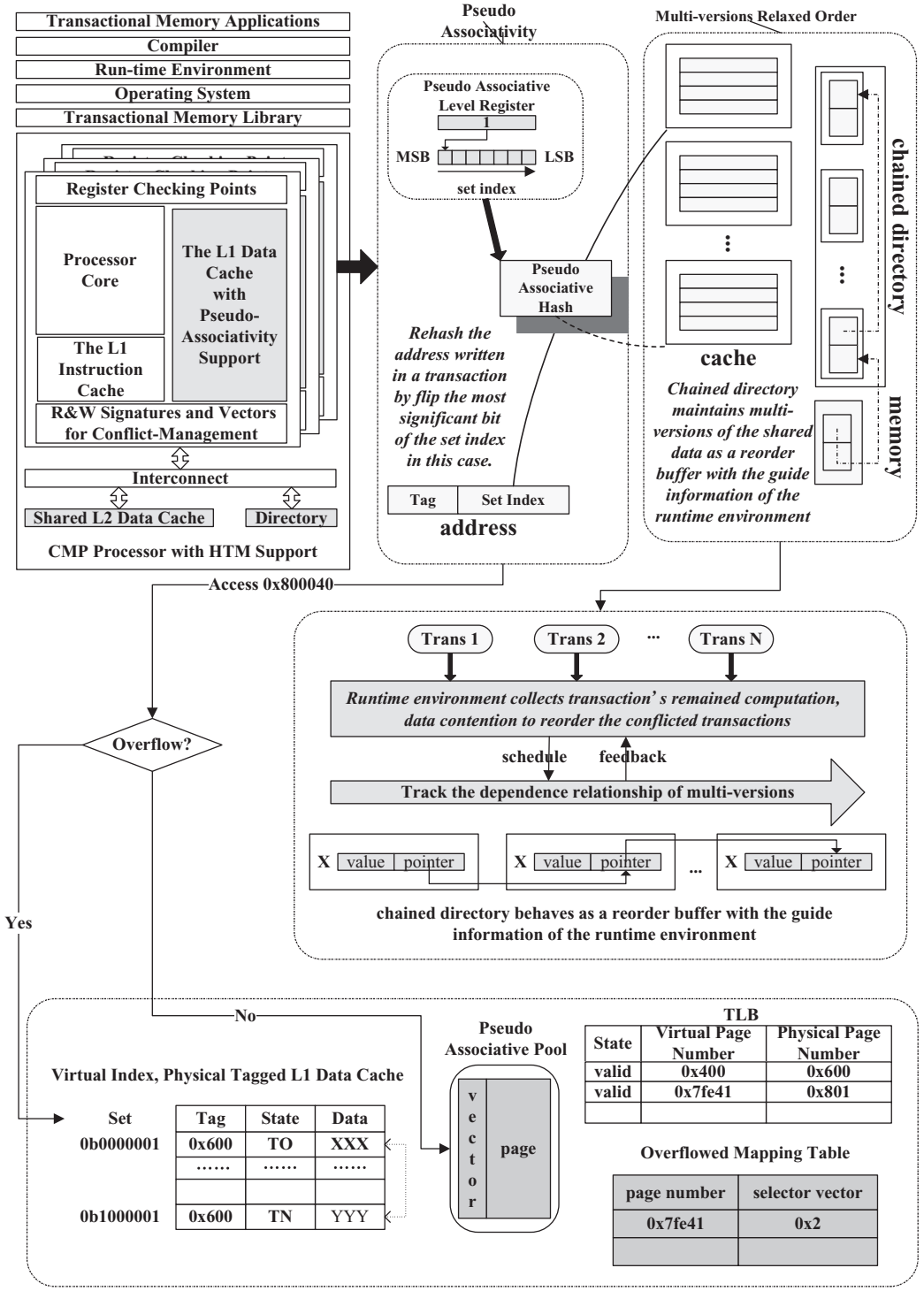
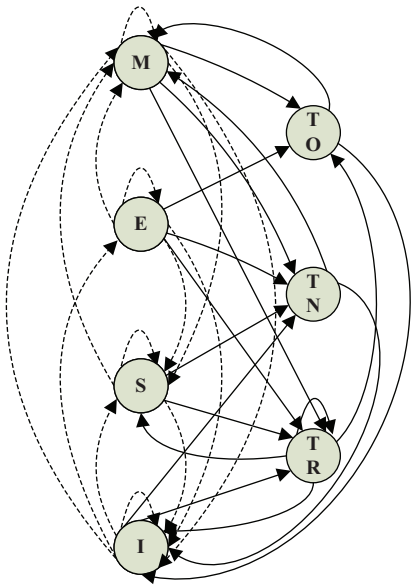


Fig. 3. Architectural overview of PARO-TM.



Main State Transitions

Current State	Event	Next State
M,E,S,I	PseudoBranch	TN
M,E,S,I	TLoad	TR
M,E	TStore	TO
TR	TStore	TO
TR	Abort	I
TO	Abort	M
TO	Commit	I
TN	Abort	I
TN	Commit	M

Main Messages

Message	Description
PseudoBranch	Branch a pseudo associative slot
TLoad	Transactional load
TStore	Transactional store
TGetS	Transactional read request
TGetX	Transactional write request
Load	Normal load
Store	Normal store
GetS	Read request
GetX	Write request
Commit	Commit the transaction
Abort	Abort the transaction

Main States

State	State Description
M	This line is modified that is read-writable, forwardable and dirty
E	This line is exclusive that is read-writable and clean
S	This line is shared among different processes which is read-only
I	Invalid means this line is in an uninitialized state
TR	This line is read by a load instruction in a transaction
TO	This line holds the old value for a transactional modification
TN	This line holds the new value for a transactional modification

Fig. 4. Main state-transition diagram of the PA-MESI Protocol.

with their dependence relationships, to guarantee data consistency in its conflict management.

3.2. Pseudo Associativity

The pseudo-associative cache design in PARO-TM is inspired by the structure of the column-associative cache [Agarwal and Pudar 1993]; [Calder et al. 1996]; [Powell et al. 2001] also known as pseudo-associative cache, that has the potential to provide more slots to handle the different versions in the transaction execution process. We modify the commonly used "MESI-CMP-Filter-Directory" cache coherence protocol in HTMs to handle the pseudo associativity (PA) in PARO-TM. This results in a PA-MESI-CMP-Filter-Directory (or PA-MESI for short) cache coherence protocol, which is integrated into the existing HTM framework.

As illustrated in Figure 4, the PA-MESI protocol includes 7 stable states, M, E, S, I, TR, TO, and TN, whose definitions are listed in the lower part of the figure. To reduce cluttering and increase the ease of understanding for our scheme, we have to omit some transient transitions that are needed to cope with the possible race conditions. Like most other cache coherence protocols, there are far more states (along with even more

transitions) than can be drawn within a figure in our PA-MESI protocol. In the state transition diagram of Figure 4, the 4 states on the left, M, E, S and I, are nontransactional and the 3 states on the right, TR, TO, and TN, are transactional. PARO-TM uses the "PseudoBranch" message to allocate the pseudo-associative slot and converts the original cache line in the M, E, S, or I state to the TN state on demand to hold the uncommitted new value, where the cache line in TN is then either transitioned to M on commit or to I on abort. When a nontransactional cache line (i.e., in the M, E, S, or I state) is read during the transaction, it is transitioned to TR that is in turn either transitioned to S on commit or to I on abort. The state is transitioned to TO with a transactional write to a cache line in the M, E, or TR state, where the TO state is then either transitioned to I on commit or to M on abort. These transitions usually need some transient transitions to avoid the race conditions. In the worst case, a transition such as switching from TO to I or a similar one needs up to 5 extra transitions. We have compared it with the traditional MESI protocol and verified that this latency does not impact the performance. PARO-TM also couples the conflict management with the PA-MESI protocol with the aid of the Bloom filter signatures to detect the conflict. It then piggybacks the NACK messages to negotiate with the competing transactions to solve the conflict in a relaxed order, where a conflict occurs either when any write request from another processor is issued to a cache line in the M, E, S, TR, TO or TN state, or when any read request from another processor is issued to a cache line in the M, TO, or TN state.

The basic function of the pseudo-associative algorithm is to invert the most significant bit of the SET index as indicated in the right part of Figure 3. This basic function can sufficiently support flattened nested transactions, in which it only needs to hold the old value of the outermost transaction and the new value of the current transaction. This mode works like the reconfigurable cache [Armejach et al. 2011] and SUV [Yan et al. 2012] but without changing the existing hardware structure to reduce the data movements. In order to expose and exploit more thread-level parallelism, we extend the basic function of the pseudo-associative algorithm to invert the Nth most significant bit of the SET index to provide more slots than existing TCC with multi-tracking or associativity-based support [**?**]; [McDonald et al. 2006] to hold the uncommitted data in the nested transactions, where N is the Nth uncommitted versions of a shared data. Throughout this article, we assume the L1 data cache to be of 32KB in size and 4-way associative with 64 bytes per cache line. It contains 7 bits in the SET index so as to support up to 7 levels of nested transactions, a sufficient depth of nested transactions for most transactional applications [Chung et al. 2006a]. Moreover, extended pseudo associativity with more slots can hold the uncommitted data if suspended transactions are scheduled in a relaxed order. This process is handled by the pseudo-associative hash module that is shown in Figure 3.

During execution, some lines may overflow from the small first-level data cache. Bloom filter signatures are used to represent transactional read and write sets for conflict detection that does not require all lines to reside in the first-level data cache. When transactional data overflows from the first-level data cache, the overflowed transactional data will be collected in a reserved pool whose size can be adjusted on demand and are held for the duration of the *whole application*, rather than per transaction as in the existing HTM proposals, along with the mapping information, in the main memory. This method provides an opportunity for other transactions of the same application to reuse the mapping information. More details on implementation and operation of this approach will be introduced in the following paragraphs.

3.3. Relaxed Ordering

Generally speaking, a transaction is composed of a series of operation code that takes some time to execute, which provides an opportunity to obtain a good relaxed-order

schedule. Traditional conflict management schemes usually adopt a two-phase locking algorithm to schedule the conflicted transactions, which is too strict and confined to expose and exploit the potential thread-level parallelism among the conflicted but serializable transactions. Our relaxed-order scheme utilizes the multi-version support from the pseudo-associativity scheme and schedules the conflicted transactions to allow as much thread-level parallelism to be exploited as possible. It achieves this goal by incorporating a chained directory to maintain multiple versions of the shared data and track their dependence relationships in a chained directory. Once the dependence relationships from the multiple versions of a shared data are determined, an efficient schedule of those transactions that have accessed the shared data can be obtained.

The chained directory scheme, unlike the conventional full-map and limited directory schemes, does not utilize the broadcast mechanism to realize a scalable directory solution, which is more suitable for many-core CMPs [Chaiken et al. 1990]. As shown in Figure 3, multiple transactions can modify the shared data. In the meantime, PARO-TM records the multi-version values in the chained directory and maintains the dependence relationships among these conflicted transactions in the runtime environment until it becomes impossible to maintain a serializable schedule of the conflicted transactions. The partial orders of transactions are determined while tracking the multi-version values of the shared data and stored in the run-time environment. And this partial-order information can guide the subsequent transactional access operations to maintain the dependence relationships of the multiple versions of the shared data. These dependence relationships are represented by a sequence of chained pointers in the chained directory. When a conflicted access from a new transaction arrives, this transaction will be inserted into a reorder schedule based on the characteristics of its peer (conflicting) transactions in the runtime environment. If a conflicted transaction cannot be inserted into the reorder schedule, it indicates that this transaction leads to a nonserializable schedule and PARO-TM will abort it and schedule it on a conflicted competitor thread. Otherwise, the new conflicted transaction's access operations will be assigned a proper version of the multi-version values based on the partial order of the existing transactions. And this transaction's modifications on the shared data will be inserted into the chained directory to maintain the dependence relationships among the concurrently conflicted transactions. We use the amount of residual work of the conflicting transactions as the main criteria to reorder them, if a new transaction is not in any partial-order dependence relationships of the concurrent transactions. For example, the runtime environment can calculate the amount of residual work of each transaction and compare it with a conflicting transaction to determine which one should precede another and it can obtain the uncommitted data forwarded from the preceding transactions. We assume the dynamically calculated mean length of each static transaction to be the length of the new dynamic transaction instance. This length minus the finished work is the residual work. As a result, PARO-TM stores each static transaction's mean length in the runtime, and for each dynamic transaction, there is a counter to record the work it has already completed. These metadata are stored in transaction's log, along with each transaction's other metadata, similar to LogTM-SE, and we allocate them in the L1 cache. When a transaction conflicts with another transaction, if this transaction still executes its transactional work, its counter will continually count its completed work; if this transaction is stalled by its competitor, its counter will not count this stalling time as its completed work until it restarts to execute its transactional work. Once a preceding transaction is finally aborted, the dependent transactions that access the forwarded and uncommitted data will also be aborted to maintain the data consistency.

PARO-TM will aggressively exploit the potential performance improvement in aborted and rescheduled transactions by adjusting the chained multi-version structure

in the chained directory to reuse their aborted computation. Thus, the structure of the chained directory is important to this exploitation. The chained directory can be implemented in either a single-directed or a double-directed chain, where the latter performs well in finding the predecessor versions because it maintains two pointers to point to both the predecessor and successor versions. Through our evaluation, however, we find that a single-directed chain provides comparable performance to a double-directed chain with much less space overheads. The main reasons are twofold. First, the number of concurrently running conflicted transactions is generally too small to incur significant traversal overhead on the chained multi-version entry. Second, the uncommon case of a large number of conflicted transactions usually leads to too many abort operations to obtain a correct schedule. So the single-directed chain is sufficient for our relaxed-order scheme.

In addition to maintaining the partial order of conflicted transactions, another concern is the complication brought to the chained directory scheme by the possible replacement operation of a cache block. Assuming the shared data X modified by N conflicted transactions, there will be $N + 1$ versions of X. If the ith version is mapped with a slot that is also mapped for variable Y in the cache map, when the thread reads Y, the ith version of X may be evicted from the data cache. Usually, the system can traverse the chain and splice the ith version from the directory or invalidate the ith version if we ignore the dependence among the conflicted transactions. Since the chain maintains the relationships, this operation may destroy this chain. Because we also record the transaction's dependence relationship in the runtime environment, we can avoid traversing the chain to find the relative order if it can be obtained from the runtime environment.

3.4. Other Design and Implementation Issues

In fact, the PA-MESI protocol is adequate in handling the bounded transactions that do not cause transactional data overflows and other exceptions (i.e., context switch, etc.) during execution. Otherwise, like most existing HTM proposals, PARO-TM will resort to the software to handle the unbounded transactions, which is implemented in the runtime environment and transactional interface.

Once the transactional data overflows from the data cache during execution, PARO-TM will collect the overflowed data (i.e., the cache line in the TN state) in the reserved pool, whose size can increase on demand to make room to hold the overflowed speculative data. In addition to storing the overflowed data, the mapping information between the old and new versions must be constructed to guide the subsequent executions. As shown in Figure 3, PARO-TM collects the new values to a page (i.e., with logical page number $0x7fe41$) to hold the speculative data overflowed from the pseudo-associative slot. It maps the old physical address $0xA00040$ to the new physical address $0x1002040$ and the mapping entry is stored in the overflowed mapping table. Because the overflowed data is associated with the old version, it is not necessary for PARO-TM to fetch the overflowed data from the main memory and merge it to the original address. Instead, the data is kept in the reserved pseudo-associative pool to avoid the extra data movements. To do this, a mechanism is required to guide the subsequent accesses to the original address to the actual address in the pseudo-associative pool. PARO-TM uses a Counting Bloom Filter to implement this mechanism by recording the address set of the already associated data and removing the address if the committed data is written back to the original address in the subsequent transactions. Here we use a 4-bit counter to avoid the possible overflow of the Counting Bloom Filter since the probability of a counter overflow with a 4-bit counter is less than $1.37 \times 10^{-15} \times m = 2.80 \times 10^{-12}$ (i.e., m is the vector size of the Bloom filter, and we choose 2048 as the vector size in this article) [Fan et al. 2000]. This is especially useful in high-contention applications. For example, when the new address, $0x1002040$, holds the new value in a committing

transaction, PARO-TM frees the old address, $0xA00040$, and uses it to hold the new value of the same shared data in subsequent transactions of the same application. This allows the existing mapping information to be reused and reduces the space for storing the mapping information without extra data movements. Once the Counting Bloom Filter overflows, the overflowed slot in the Counting Bloom Filter should always be set to "valid" to guarantee the correctness. Alternatively, we can increase the counter size or add a software handler to handle this case if the overflows happen frequently.

In order to support context switching and thread migration, PARO-TM adds several tag bits to hold the transaction ID. This enables PARO-TM to identify the suspended transaction or the current running transaction in the data cache and store/restore its state information, such as signatures, checkpoints, etc., to/from memory. The system automatically maintains the mappings between the old and new values, which can forward the proper value to the new thread.

3.5. Putting It All Together

In order to verify PARO-TM's functionality, we have implemented it on GEMS [Martin et al. 2005], a popular simulation platform based on the Simics simulator [Magnusson et al. 2002], which provides a standard "magic instruction" interface to researchers to add their customized function interface on the simulation platform. Programmers need to instrument the magic instructions to the target parallel programs to mark their critical sections as transactions. Once the magic instruction is executed, Simics will stop the simulation and invoke a user-defined profiling control program, from which PARO-TM can collect the simulation data of the system to profile its execution. In what follows, we will put the pseudo-associative and relaxed-order schemes together to show the key operations of PARO-TM.

To illustrate PARO-TM's operational mechanism clearly, for cases when no transactional data is overflowed from the first-level data cache, we list in Figure 5 three representative operations: (1) conflict detection, (2) flushing the committed pseudo-associative data, and (3) supporting a nested transaction. The case with overflowed transactional data is illustrated in Figure 6.

In Figure 5(a), a transactional store may branch a pseudo-associative slot and convert the line's state. When a remote read request reaches this "TO" line, a transactional conflict happens and PARO-TM can detect it eagerly. In our relaxed-order conflict management scheme, the transaction on core 2 can be scheduled before or after the transaction on core 1 based on their running behavior. Suppose that the transaction on core 1 is scheduled as the one preceding the transaction on core 2 in their partial order, as shown in Figure 5(b).

When the transaction on core 1 commits its computation (i.e., operation 1 in Figure 5(b)), the cache lines with state "TO" will transit to "I" and the cache lines with state "TN" will change to "M". Note that the "P" tag before the original lines indicates that this line is pseudo associative with another slot. This feature is especially useful after a transaction commits its work and leaves some pseudo-associative slots containing the latest version of the shared data. Back to Figure 5(b), this tag can assist subsequent access requests to obtain the proper data (steps 3 and 4). In order to avoid extra operations during steps 3 and 4, PARO-TM will flush the pseudo-associative slot to transfer the data back to the original address and remove the P tag (step 5).

In Figure 5(c), we show how PARO-TM works under a nested transaction. PARO-TM branches two pseudo-associative slots to hold the speculative data, and increases the P tag to indicate the pseudo-associative state, where P=2 means that subsequent accesses may be associative with two slots, and the latest speculative data is in a slot with its second most significant bit of SET index flipped (recall the pseudo-associative hash module in Figure 3). In step 6, the inner transaction commits its work, while PARO-TM

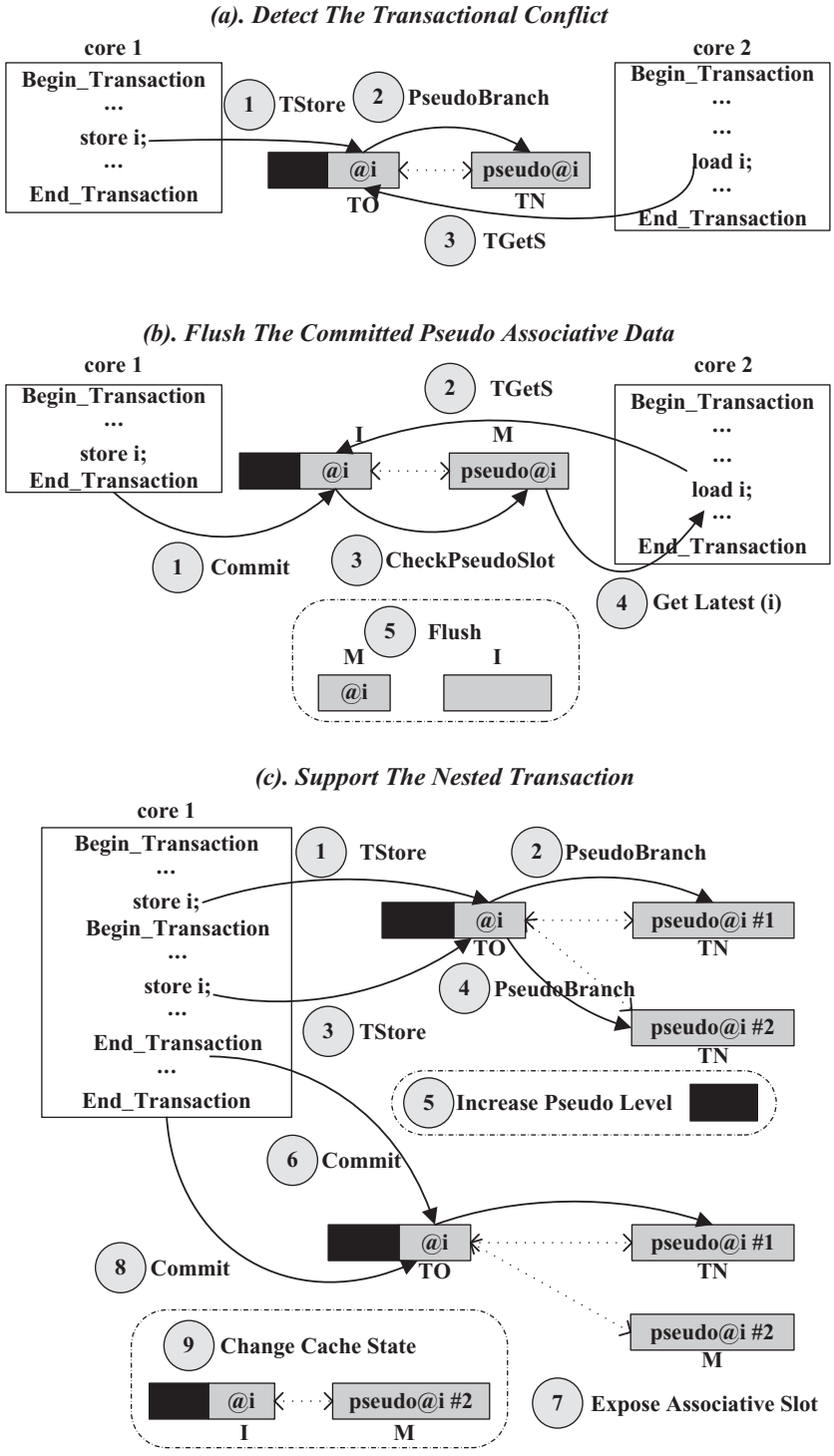


Fig. 5. Representative PARO-TM operations when no transactional data is overflowed from the L1 data cache.

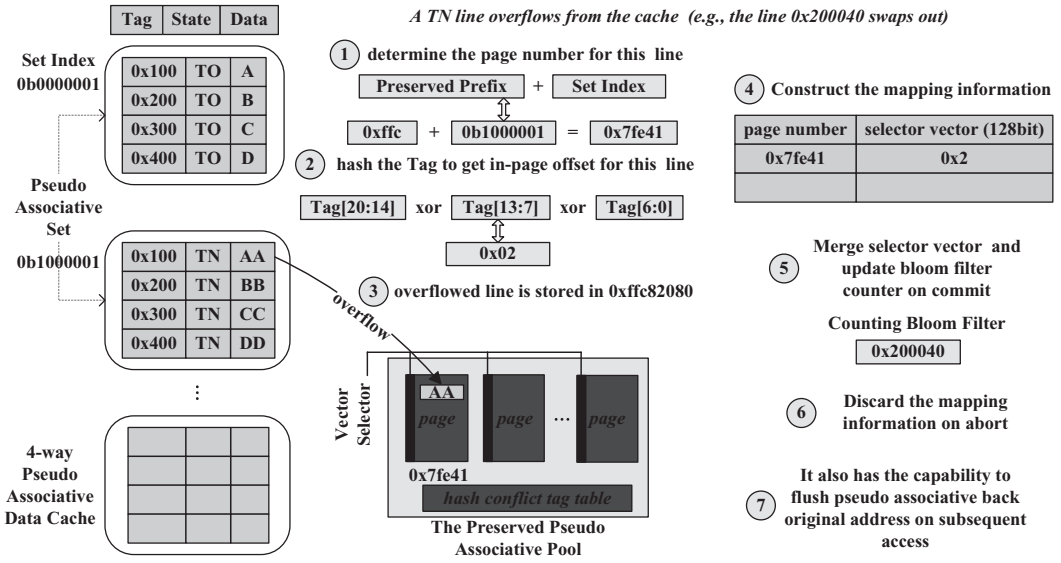


Fig. 6. Structures and operations when transactional data overflows from the L1 data cache.

exposes the pseudo-associative slot no. 2 in step 7. Because the outer transaction has not modified i since the inner transaction committed its work, the pseudo-associative slot no. 1 will be discarded when the outer transaction commits its work in step 8 while the P tag is still equal to 2. The subsequent accesses will be redirected to the pseudo-associative slot no. 2 because it is the latest version. The extra P tag needs 3 bits to label this information in the 32KB 4-way associative first-level data cache.

Once transactional data overflows from the first-level data cache, the speculative data in "TN"will be written to a reserved pseudo-associative pool and a mapping between the old and new values will be constructed because the pseudo-associative cache cannot maintain their relationship when data is overflowed.

Figure 6 shows that the pseudo-associative pool allocates new pages, called pseudo-associative pages in this article, on demand while the page number is determined by concatenating a reserved prefix and SET index bits of the overflowed line, and the in-page offset is calculated by hashing the tag bits. To handle the possible hash conflict problem, the preserved pseudo-associative pool constructs a hash conflict tag table to help indicate the conflicted block's in-page offset. In addition to storing the overflowed data to the pseudo-associative pool, PARO-TM constructs the overflowed mapping information in a mapping table that contains the pseudo-associative page number and its vector selector. Each bit of a pseudo-associative page's vector selector corresponds to a slot containing an overflowed data item associated with this page. Once the transaction commits its computation, it will merge the vector selector of the mapping table with its pseudo-associative page. By looking up the vector selector, a subsequent access to any overflowed data can determine whether to check the pseudo-associative page or directly go to the memory to fetch the data. A Counting Bloom Filter checking is incorporated to help PARO-TM make the right decision. While the number of the overflow entries in the table may grow quickly, the pseudo-associative data can in fact be flushed back to the original address, a common practice in transactional applications.

Now, back to Figure 2, $TX3$ has the least amount of residual work and is not in any partial-order dependence relationships with $TX1$ and $TX2$. We use the dynamically calculated mean length of each static transaction minus the finished work to estimate

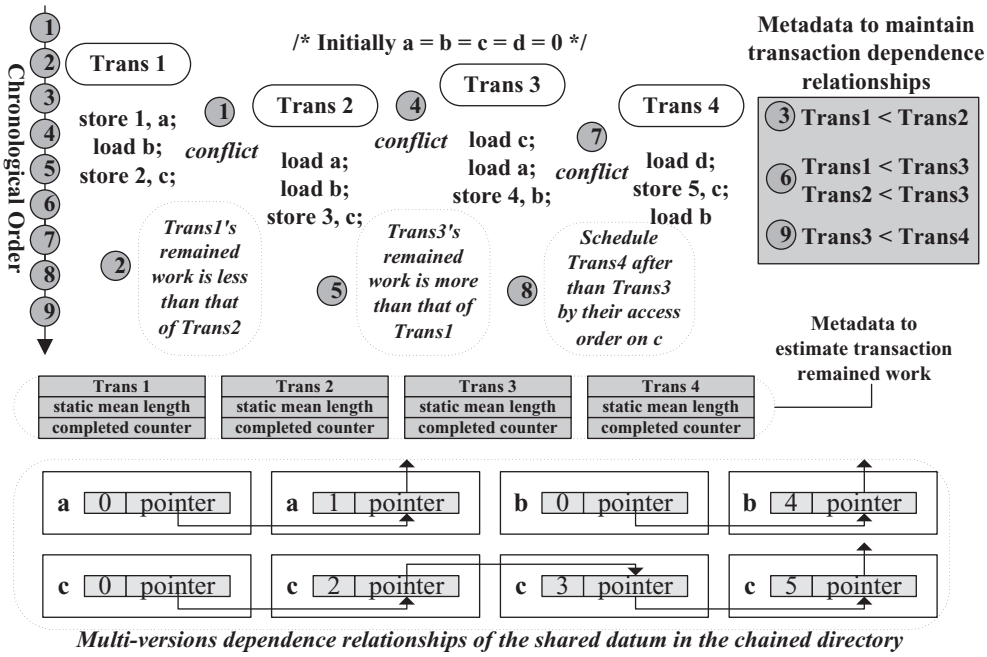


Fig. 7. A Complex example in relaxed order schedule.

the residual work for each dynamic transaction instance of the static transaction. Here we suppose this method can correctly gauge the residual work of the concurrent transactions. In our relaxed-order scheme, the oldest version 80 will be loaded by $TX3$ and $TX3$ should be scheduled to run ahead of $TX1$ and $TX2$.

Figure 7 shows a more complex example in relaxed-order scheduling. The operations labeled with numbers are ordered in chronological sequence. First, $TX1$ and $TX2$ are ordered by the amount of their residual work. Then, because $TX3$ has more residual work than $TX1$ and $TX2$, it loads the old version of b (i.e., 0), $TX3$ is scheduled to follow $TX2$. Finally, since $TX4$ tries to change c and $TX3$ has loaded the forwarded value of c (i.e., 3) from $TX2$, $TX4$ should commit after $TX3$. Different from the rule, which says "suppose A has forwarded a line to B, if C reads that line, then C should receive the line from B, not A" and is used in DATM [Ramadan et al. 2008] to track the dependence among the conflicting transactions, our multi-version-based relaxed-ordering method can schedule the conflicting transactions B and C in a more relaxed manner to exploit more thread-level parallelisms. Multi-version support lets PARO-TM have more freedom to select the proper version to better exploit the parallelism in conflicting transactions.

In summary, PARO-TM does not require any extra, dedicated hardware support, except for the extension of the cache coherence protocol and the Counting Bloom Filter signatures. It must be noted that most existing HTMs modify the cache coherence protocol to incorporate HTM functionalities. At the same time, it will construct some software structures to hold the dependence relationships among the concurrent dynamic transactions, the statistical mean length of each static transaction, the overflowed data and their mappings, and the other necessary transactional metadata used in LogTM-SE. Through the evaluation, which will be presented in the next section, it is demonstrated that PARO-TM's software overhead is reasonably low while it provides a notable performance gain over the existing state-of-the-art proposals.

Table I. Configuration of the simulated CMP System

Processor	1.2 GHz in-order, single issue
L1 Cache	32 KB 4-way, 64-byte line, write-back, 1-cycle latency, with 1-cycle latency to check the pseudo-associative slot
L2 Cache	8 MB 8-way, write-back, 15-cycle latency
Memory	4 GB, 4 banks, 150-cycle latency
Interconnect	Mesh, 2-cycle wire latency, 1-cycle route latency
Signature	2K-bit in vector size of the Bloom Filter signature, with 4-bit Counting Bloom Filter signature

Table II. Workload Characteristics of The STAMP Benchmark Suite

	Input Parameters	Length	Read-set		Write-set	
			avg	max	avg	max
kmeans (L&F)	40/40 clusters, 2048 rows	106	3.8	7	1.7	2
ssca2 (L&F)	8k nodes, 3 edges, 3 length	21	2.0	3	2.0	2
vacation(L&C)	4 queries, 4k transactions, 16k relations	2.1K	56.9	99	8.2	18
intruder(H&F)	10 attacks, 4 length, 2k flow	237	5.5	30	3.4	23
genome(H&C)	16k segs, 256 gene, 16 length	1.7K	24.1	234	5.9	151
labyrinth(H&C)	32x32x3 maze, 64 routes	317K	91.1	311	90.7	252
yada(H&C)	20 angle, 633.2 input mesh	6.8K	37.3	421	30.2	370

Here L, H, F, and C represent the low-contention, high-contention, fine-grained, and coarse-grained characteristics of the application, respectively.

4. EVALUATION AND ANALYSIS

The design and implementation of PARO-TM described in Section 3 qualitatively show its benefits. In this section, we assess PARO-TM's effectiveness quantitatively by executing the STAMP benchmark suite on an execution-driven CMP simulator.

4.1. Evaluation Environment and Workload

We configure a 16-core CMP with the HTM functionality on the GEMS 2.1 simulator [Martin et al. 2005], where each core has a private L1 data cache and shares an L2 data cache. Table I summarizes the configuration parameters of the simulated CMP system. In order to examine the performance impact of the high-contention and coarse-grained applications for which PARO-TM is designed, we choose the STAMP benchmark suite as the evaluation workload and list the input parameters in Table II. Note that we have excluded the bayes application from our evaluation because it has been shown to exhibit unpredictable behavior and high variability in its execution time [Minh et al. 2008; Dragojevic and Guerraoui 2010]. At the same time, we have adopted some optimizations to improve the scalability of the STAMP benchmark suite. These optimizations include: (1) the alignment of data at the cache line boundary by padding around the shared data, (2) the use of a 32-core machine to model a 16-core HTM configuration while leaving cpu0 for OS usage to avoid the interference from the operation system, (3) the labeling of the private data in transaction to alleviate the possible false conflict by the Bloom filter signature, and (4) the adoption of the hybrid policy to overcome the starving writer pathology in LogTM-SE. That is, if a transaction believes that it is a starving writer, it will force other readers to abort and use the requester-stall policy for everything else.

To assess PAROT-TM's performance benefit, we compare it against LogTM-SE, TCC, DynTM, and SUV-TM, four of the state-of-the-art HTM schemes. LogTM-SE is derived from GEMS and is configured to use the hybird policy to resolve transactional conflicts.

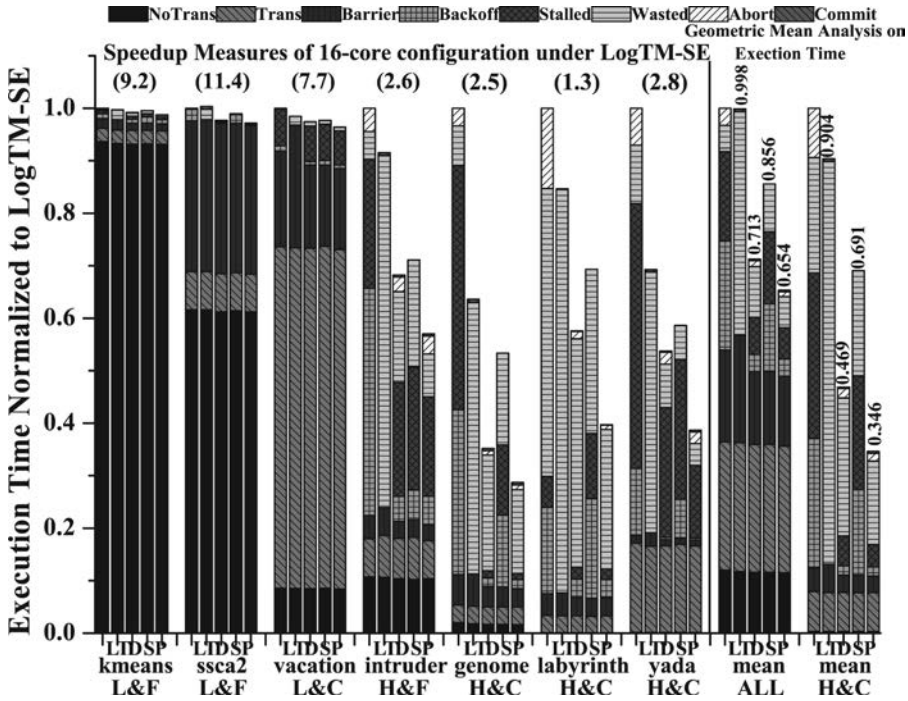


Fig. 8. The execution times of STAMP on various HTMs, where L, T, D, S, and P represent LogTM-SE, TCC, DynTM, SUV-TM, and PARO-TM, respectively and all execution times are normalized to that of LogTM-SE. The speedup measures obtained for the applications when running on a 16-core LogTM-SE configuration are included.

Meanwhile, we follow the instructions provided by GEMS to simulate the TCC scheme with row associativity [McDonald et al. 2006], which uses the committer-wins policy to resolve transactional conflicts. To emulate the side-effects caused by the redo log overflows, we have modified it appropriately by limiting the size of the write buffer and inserting some waiting times, which is estimated by the overflowed size and network's latency and congestion, to block the data access operations. Based on LogTM-SE, we have implemented DynTM and SUV-TM that represent the latest progress in HTM systems. While DynTM uses the hybrid policy in eager mode and the committer-wins policy in lazy mode to resolve transactional conflicts, SUV-TM uses the hybrid policy to resolve transactional conflicts. We run the simulation 10 times for each workload to obtain statistically meaningful results, where the same workload is used to warm the cache and obtain a checkpoint to rerun the whole application for each simulation.

4.2. Experimental Results and Analysis

In order to obtain a comprehensive understanding of the overheads incurred by various HTMs, we break down the execution time into these components: time due to nontransactional work (NoTrans), time due to unstalled transactional work (Trans), time due to waiting on a barrier (Barrier), time due to stall after an abort (Backoff), time due to stall to resolve the conflict (Stalled), time due to wasted work when a transaction is aborted (Wasted), time due to rolling back during abort (Abort), and time to make a speculative modification globally visible on commit (Commit).

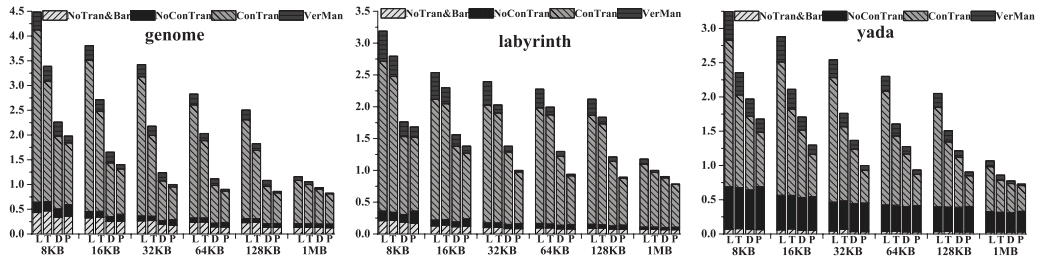
As shown in Figure 8, PARO-TM consistently achieves the best performance among the HTMs compared, outperforming the latest and the state-of-the-art LogTM-SE,

TCC, DynTM, and SUV-TM by the average performance gains of 34.6%, 34.5%, 8.3%, 23.6% respectively across all the 7 applications in the STAMP benchmark suite. PARO-TM's performance margins are widened to 65.4%, 61.7%, 26.2%, and 49.9% over the four schemes respectively under the 3 high-contention and coarse-grained applications in the STAMP benchmark suite. LogTM-SE suffers from the delays resulting from restoring the old values on abort and stalling some conflicting transactions to save the transactional computation. However, the cyclical dependency among the conflicting transactions must be resolved by aborting some transactions and providing a random back-off to prevent the same conflicts from happening again in the near future. This usually happens in high-contention applications with the extra data movements on aborted transactions, which can lead to a vicious cycle in the eager approach. TCC performs better than LogTM-SE, but its commit operations will waste a lot of time especially when the redo log overflows from the data cache. In this case, it will abort the competing transactions on commit, which wastes significant transactional computation. Through the evaluation, we find that PARO-TM outperforms TCC, because the former organizes different versions in the same L1 data cache by the PA-MESI protocol to avoid data movements in both transactional abort and commit operations, and the relaxed-order conflict management also contributes more to performance improvement than TCC's conflict resolution on commit. DynTM combines the EE (i.e., eager conflict detection and eager version management) and LL (i.e., lazy conflict detection and lazy version management) approaches. This helps DynTM reduce the time spent on Backoff, Stalled, Wasted, Commit, and Abort only when the selector makes the correct choice. PARO-TM does not rely on a predictor to select the proper execution mode. Rather, it associates a pseudoslot to hold the speculative update of the transaction, switches to the proper version with the help of the PA-MESI protocol and schedules the conflicting transactions in a relaxed order. This allows PARO-TM to reduce the overhead due to the Stalled, Wasted, and Abort components and enables it to consistently perform the best. PARO-TM also avoids the potential data movements across the memory hierarchy in DynTM when the dirty data already exists in the data cache before the transaction begins or restarts the aborted transaction to load the shared data. SUV-TM avoids extra data movements via a redirection table, but it fails to reduce the dynamic overheads on conflicting transactions. So it performs better than LogTM-SE and TCC but worse than DynTM and PARO-TM. In summary, PARO-TM exploits the most amount of thread-level parallelism of the concurrent transactions via the pseudo-associativity and relaxed-ordering schemes to reduce both the static and dynamic transactional overheads among the state-of-the-art HTM schemes.

One main performance concern of the pseudo-associativity version management scheme in PARO-TM is the possibility for the PA-MESI protocol to slow fast hits down to slow hits if they frequently hit in the pseudo-associative slot, which may lengthen the hit time. We collected and analyzed the access operations to the shared data and summarized the hit rates of these access operations in Table III. Here, we have found that the PA-MESI protocol not only avoids the PARO-TM going to the next (lower) level of the memory hierarchy to access the shared data, but is also able to dynamically reverse the roles of the pseudo-associative slot and the original address to minimize the number of slow hits by flushing the committed data, whose operation is shown in Figure 5(b). From the statistics in Table III, PA-MSEI slows down the memory access no more than 30% from the fast hit in the L1 cache, which is much better than the schemes that store logs in the L2 cache or the main memory that will access the lower level of the hierarchy to fetch the data. We believe that this behavior is very dependent on the characteristics of transactional workloads, but the concurrent accesses to the shared data from multiple transactions will provide a lot of opportunity to exploit this feature.

Table III. The Hit Rates in the PA-MESI Protocol

	Hit in The Data Cache		Hit in The Main Memory	
	Fast Hit	Pseudo Hit	Fast Hit	Pseudo Hit
kmeans	70.1%	29.7%	0.1%	0.1%
ssca2	69.3%	30.5	0.1%	0.1%
vacation	73.2%	26.5%	0.1%	0.2%
intruder	72.6%	26.8%	0.1%	0.5%
genome	65.2%	22.2%	4.7%	7.9%
labyrinth	57.1%	20.7%	4.7%	17.5%
yada	56.8%	22.3%	4.2%	16.7%



(a) Study on Genome (b) Study on Labyrinth (c) Study on Yada

Fig. 9. Sensitivity study on the size of the L1 data cache on various HTMs, where L, T, D, and P represent LogTM-SE, TCC, DynTM, and PARO-TM respectively and the execution times are normalized to that of PARO-TM with a 32kb L1 data cache.

To better understand the performance impact of the size of the L1 pseudo-associative data cache and verify the benefit of PARO-TM on managing transactional overflowed data, we carry out a sensitivity study on the size of the L1 data cache. We divide the execution time into four parts: time due to nontransactional work with barrier waiting time (NoTran&Bar), time due to nonconflicting transactional work without versioning time (NoConTran), time due to conflicting transactional work without versioning time (ConTran), and time due to version management on organizing different versions, merging data on commit, and repairing data on abort (VerMan). From Figure 9, we find that PARO-TM is consistently the best performer among the competing schemes. LogTM-SE does not work well under the high-contention and coarse-grained transactional workload because it will block the conflicting transactions eagerly and incur a high cost on aborting the conflicting transactions. TCC may waste the computations on the conflicting transactions that are deferred to solve on commit, and this problem will worsen under the coarse-grained workload with many transactional overflows. DynTM will suffer the same pathology when transactional data overflows the first-level data cache, although it can mitigate much of this overhead by its transactional mode selector. Moreover, when the cache size shrinks down to $8KB$, we find that PARO-TM's extra time spent on $NoTran\&Bar$ is comparable to that by DynTM, which verifies that the extra space overhead in PARO-TM is not a problem. At the same time, these results also demonstrate PARO-TM's performance benefits of avoiding the data movements across the memory hierarchy with a small data cache in the pseudo-associative pages by effectively managing the overflowed transactional modifications. When we scale the cache size to 1MB, we find that PARO-TM can obtain more benefits than other schemes, which suggests that our design is more suitable for designs aimed at handling emerging applications with higher contention and coarser granularity. So we believe that the pseudo-associativity scheme provides more performance benefits than existing schemes.

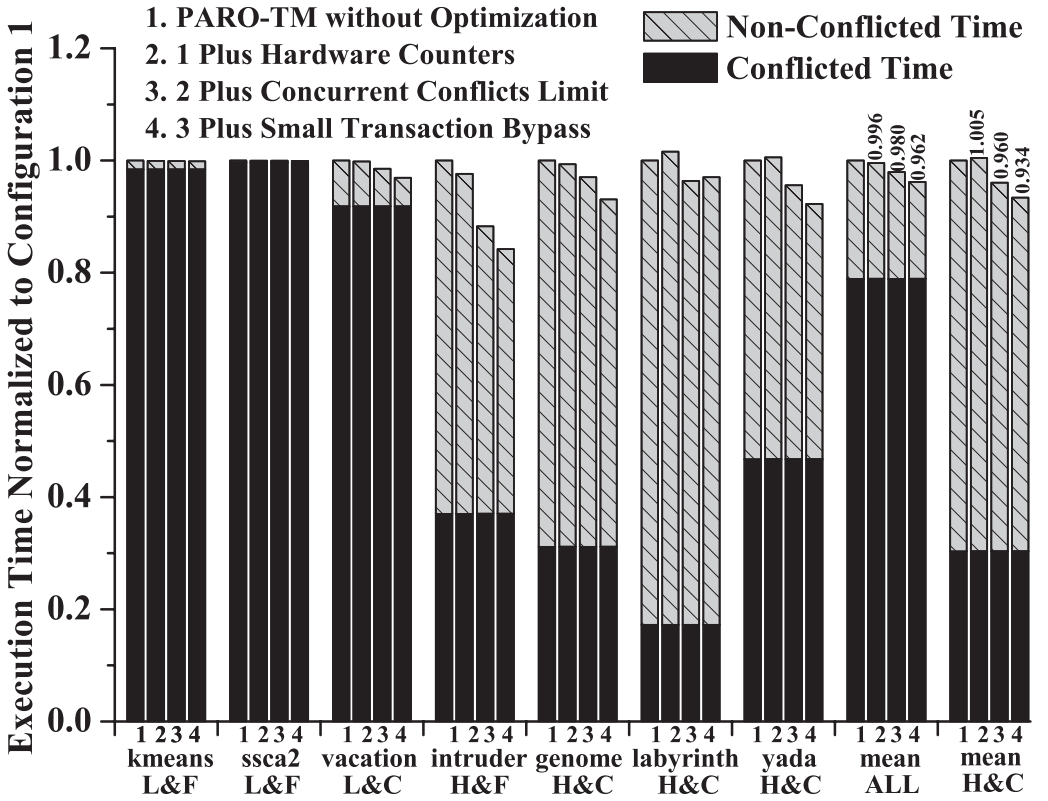


Fig. 10. A case study of four conflict management policies with the pseudo-associativity version management scheme, where all execution times are normalized to that of PARO-TM.

From Figure 8, we find that reducing the data movements induced by version management alone, like SUV-TM, cannot effectively and efficiently exploit the potential thread-level parallelism, especially under the coarse-grained and high-contention workloads. DynTM and PARO-TM have partially addressed this problem by adopting new conflict management schemes to better schedule and resolve conflicted transactions. To evaluate the performance benefits of the relaxed-order conflict management scheme proposed in PARO-TM, we design an experiment to combine the pseudo-associativity version management scheme with several commonly used conflict management schemes other than PARO-TM's multi-version-based relaxed-order method for a fair comparison. Because the pseudo-associativity version management scheme maintains both the old and new versions in the same memory hierarchy, it can be implemented in either eager or lazy mode. Here we compare our relaxed-order method with three policies: (1) detect conflict eagerly and the requester wins, (2) detect conflict eagerly and the requester stalls until a cyclic dependence happens to abort the requester, (3) detect conflict lazily and the committer wins. As shown in Figure 10, we have divided the whole execution time into two parts: one is due to the conflicted time (i.e., including Backoff, Stalled, Wasted, and Abort) and another is due to non-conflicted time (i.e., including NoTrans, Trans, Barrier, and Commit), from which we can see that PARO-TM is able to exploit more conflicted parallelism especially under the high-contention and coarse-grained applications. The relaxed-order method performs well in most cases except for the low-contention and fine-grained ssca2, where

the relaxed-order method's overheads overshadow its benefit and make it underperform the existing policies. Overall, PARO-TM's relaxed-order conflict management only incurs 1% extra overheads.

To better understand impact of the software runtime overheads on PARO-TM's performance, we conduct a sensitivity study on this impact by scaling the software overheads incurred in the relaxed-order schedule by incorporating a hardware accelerator to speed up the software operations. Software's overheads stem from the dynamic calculation of the mean length of each static transaction, the residual work of conflicted dynamic transaction instances, and maintenance and lookup of the dependence relationships of the concurrent uncommitted transactions. We argue that the calculation of the length and residual work can be easily handled by adding several hardware counters at a very low hardware cost. Meanwhile, the system can avoid calculating the residual work when the conflicted transaction has already been put in the existing partial-order dependence relationship. Most overheads, therefore, come from managing the partial-order dependency graph to find a circle in the graph. When the multiversion-supported hardware directory detects a conflict, it will invoke the software runtime to update the partial-order dependency graph. In our evaluation, we find that in most cases when more than 4 concurrent conflicted transactions access the same shared data, there is usually no way to guarantee a searializable schedule even in our partial-order approach. One obvious optimization is to stall the new conflicted transaction until the number of the concurrent conflicted transactions accessing the same shared data is less than 4. So it is easy to utilize this feature to limit the software overheads on managing the partial-order dependency graph. Moreover, in some very high-contention transactional workloads such as labyrinth and yada, spending more time on software to schedule the conflicted transactions usually does not significantly degrade the performance because this extra time can help PARO-TM to serialize the conflicted transactions more accurately. At the same time, to reduce the software overheads on small transactions, we dynamically bypass the partial-order schedule when the size of the transaction is less than 1k instructions. Figure 11 presents the execution time with the different runtime overheads, where we make PARO-TM run in these four configurations: (1) the basic PARO-TM without any software overhead optimization, (2) PARO-TM assisted by hardware counters, (3) PARO-TM with limited concurrent conflicts and assisted by hardware counters, and (4) PARO-TM with limited concurrent conflicts, assisted by hardware counters and bypassing small transactions. From Figure 11, we find that the hardware counters do not improve the performance significantly, while limiting the number of concurrent conflicted transactions and bypassing small transactions can boost the performance significantly. The reasons are twofold. First, a smaller number of conflicted transactions requires significantly lower overhead on maintaining the partial-order relationships. Second, limiting conflicted transactions alleviates the high contention to reduce the conflicts while bypassing small transactions works well under the workloads with many contended and small transactions, such as intruder in our evaluation. In summary, although the runtime overheads may impact the performance, we argue that it is acceptable considering it can help exploit more thread-level parallelism in the conflicted transactions.

4.3. Complexity Analysis

PARO-TM has incorporated the pseudo-associative cache and multi-version chained directory to better exploit the thread-level parallelism of the transactional workloads. Throughout our evaluation, these two optimizations are shown to have great potentials for improving HTM's performance. These performance benefits of PARO-TM, however, come at some hardware cost as it complicates the hardware structure of the existing data cache and directory to some extent. Here we will analyze the complexity introduced

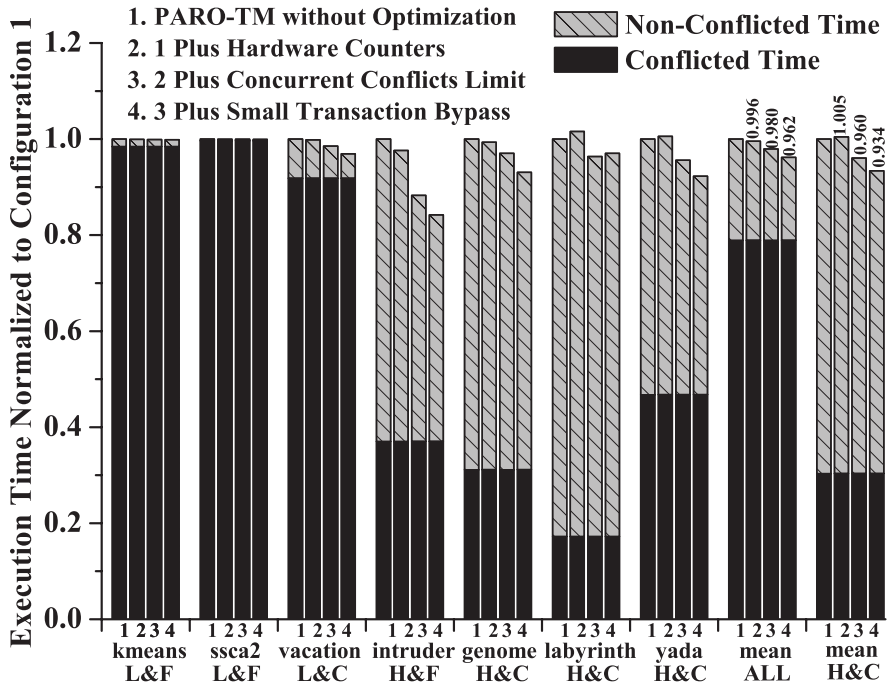


Fig. 11. A Case study of overheads on runtime, where all execution times are normalized to that of basic PARO-TM without any optimization.

Table IV. Estimated Overheads on Area and Power

Component	Area (um^2)	Power (mW)
Signature's Overhead per Core	22983.67	8.87
Pseudo Associativity's Overhead per Core	74292.52	25.46
Chained Directory's Overhead per Core	58652.21	18.52
Total Overheads per Core	155928.40	52.85
Overhead Normalized to That of Rock	1.1%	0.53%

by PARO-TM and discuss whether it is worthwhile to add these hardware components into the existing hardware transactional memory systems.

PARO-TM needs to add the countering Bloom filter signatures, extend the cache coherence protocol to support pseudo associativity, and modify the directory's structure to add the pointers to point to the next slots. However, in the current GEMS simulator, it is difficult to accurately estimate the area and power overheads of the PA-MESI protocol and multi-version chained directory. As a result, we have implemented PARO-TM's pseudo-associative cache and multi-version chained directory in Verilog HDL and use Synopsys Design Compiler to estimate their area and power overheads. Table IV summarizes the area and power estimations based on Sun's Rock processor, which is fabricated in the 65nm technology and each of the 16 cores has an area of $14mm^2$ and an estimated power consumption of $10W$ [Tremblay and Chaudhry 2008]. We target at the 65nm cell library and set the clock frequency to 1.2 GHz to constrain the synthesis process to obtain the results. The results in Table 4.3 indicate that PARO-TM is able to obtain significant performance benefits with a very small area and power overheads.

In addition to the hardware components added, PARO-TM will construct a software structure to guide the subsequent accesses to the overflowed transactional data in

the pseudo-associative pool. The space overhead of the software structure designed to hold the mapping information is no more than 29KB throughout our evaluation. At the same time, the space requirement of the pseudo-associative pool is about 440KB, which can be totally placed in the L2 data cache. On the other hand, the relaxed ordering schedule needs less than 1KB space for each core to store the metadata. Moreover, we find that the overhead incurred by our relaxed-order schedule may increase the access latency. But in our evaluation we find that the overhead incurred by this extra latency is less than 3.9% on average when an average of more than 8.1% performance gain on execution time is achieved by this method. Adding some hardware components may further reduce this overhead, a research topic we plan to study in our future work.

In summary, we believe that the design philosophy of PARO-TM to reduce the data movements across the memory hierarchy and exploit the parallelism in the conflicted transactions is in the right direction to design a better HTM system. Through our evaluation and analysis, the overhead of PARO-TM is reasonably low and thus is acceptable and practical in our opinions.

5. CONCLUSION

In this article, we propose an integrated pseudo-associativity and relaxed-order approach to hardware transactional memory, called PARO-TM, which provides a new framework to couple conflict management with version management to simultaneously reduce the data movement overheads and exploit the potential parallelism among conflicting transactions by means of a pseudo-associative cache and chained directory.

We evaluate PARO-TM by comparing it with LogTM-SE, TCC, DynTM, and SUV-TM. Through extensive execution-driven experiments under the STAMP benchmark suite that represents a wide spectrum of coarse-grained and high-contention applications, PARO-TM is shown to achieve average performance gains of 34.6%, 34.5% 8.3%, 23.6% across the 7 selected applications in the STAMP benchmark suite over LogT-SE, TCC, DynTM, and SUV-TM, respectively. Its performance margins over these competing HTMs are widened to 65.4%, 61.7%, 26.2%, and 49.9%, respectively, under the 3 high-contention and coarse-grained applications in the STAMP benchmark suite that represent the likely future workloads on the TM systems. The area and power overheads of PARO-TM are estimated to be 1.1% and 0.53% of Sun's Rock processor.

REFERENCES

ADL-TABATBAI, A.-R., SHPEISMAN, T., AND GOTTSCLICH, J. 2011. Draft specification of transactional language constructs for c++. http://www.open-std.org/Jtc1/sc22/wg14/www/docs/n1613.pdf.

AGARWAL, A. AND PUDAR, S. D. 1993. Column-Associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. 179–190.

ANANIAN, C., ASANOVIC, K., KUSZMAVI, B. C., LEISERSON, C. E., AND LIE, S. 2005. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA'05)*. 316–327.

ANSARI, M., KOTSELIDIS, C., WATSON, I., KIRKHAM, C., LUJAN, M., AND JARVIS, K. 2008. Lee-TM: A non-trivial benchmark for transactional memory. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*. 196–207.

ARMEJACH, A., SEYEDI, A., TITOS-GIL, R., HUR, I., CRISTAL, A., ET AL. 2011. Using a reconfigurable L1 data cache for efficient version management in hardware transactional memory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 361–371.

AYDONAT, U. AND ABDELRAHMAN, T. S. 2010. Hardware support for relaxed concurrency control in transactional memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MIRCO'10)*. 15–26.

BLAKE, G., DRESLINSKI, R., AND MUDGE, T. 2009. Proactive transactional scheduling for contention management. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 156–167.

Blake, G., Dreslinski, R., and Mudge, T. 2011. Bloom filter guided transaction scheduling. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*. 75–86.

Blundel, C., Devietti, J., Lewis, E. C., and Martin, M. M. K. 2007. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. 24–34.

Bobba, J., Goyal, N., Hill, M., Swift, M., and Wood, D. 2008. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. 127–138.

Bobba, J., Moore, K., Volos, H., Yen, L., Hill, M. D. et.al. 2007. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*. 81–91.

Calder, B., Grunwald, D., and Emer, J. 1996. Predictive sequential associative cache. In *Proceedings of the 2nd IEEE Symposium on High Performance Computer Architecture (HPCA'96)*. 244–253.

Carlstrom, B. D., McDonald, A., Carbin, M., Kozyrakis, C., and Olukotun, K. 2007. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*. 56–67.

Ceze, L., Tuck, J., and Torrellas, J. 2006. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. 227–238.

Chafi, H., Casper, J., Carlstrom, B. D., McDonald, A., Cao, C., et al. 2007. A scalable, non-blocking approach to transactional memory. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. 97–108.

Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A. 1990. Directory-Based cache coherence in large scale multiprocessors. *Compt. 23*, 6, 49-58.

Chaudhry, S. 2008. Rock: A third generation 65nm, 16-core, 32 thread + 32 scout-threads cmt sparc processor. In 20th *HotChips Conference*.

Chuang, W., Narayanasamy, S., Venkatesh, G., Sampson, J., Van Biesbrouck, M., et al. 2006. Unbounded page-based transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. 347–358.

Chung, J., Chafi, H., Minh, C.C., McDonald, A., Carlstrom, B. D., et al. 2006a. The common case transactional behavior of multithreaded programs. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA'06)*. 266–277.

Chung, J., Minh, C. C., McDonald, A., Skare, T., Chafi, H., et al. 2006b. Tradeoffs in transactional memory visualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. 371–381.

Click, C. 2009. Azul's experiences with hardware transactional memory. In *Transactional Memory Workshop.*

Colohan, C. B., Ailamaki, A., Steffan, J. G., and Mowry, T. C. 2006. Tolerating dependences between large speculative threads via sub-threads. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. 216-226.

Dice, D., Lev, Y., Moir, M., Nussbaum, D., and Olszewski, M. 2009. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. 157–168.

Dragojevic, A. and Guerraoui, R. 2010. Predicting the scalability of an stm a pragmatic approach. In *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*.

Fan, L., Cao, P., Almeida, J., and Broder, A. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw. 8*, 3, 281–293.

Garzaran, M. J., Prvulovic, M., Llaberia, J. M., Vinals, V., Rauchwerger, L., et al. 2003. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA'03)*. 191-202.

Gopal, S., Vijaykumar, T., Smith, J., and Sohi, G. 1998. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA'98)*. 195–206.

Guerraoui, R. and Kapalka, M. 2010. *Principles of Transactional Memory*. Morgan and Claypool.

Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., et al. 2004. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. 102–113.

Haring, R. 2011. The IBM blue gene/q compute chip+simd floating-point unit. In *Proceedings of the 23rd IEEE International Symposium on High Performance Chips (HotChips'11)*.

Harris, T., Larus, J., and Rajwar, R. 2010. *Transactional Memory, 2nd ed. Morgan and Claypool*.

HERLIHY, M. AND MOSS, J. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. 289–300.

INTEL. 2012. Intel architecture instruction set extensions programming reference. http://software.intel.com/sites/default/files/m/a/b/3/4/d/41604-319433-012a.pdf.

KESTOR, G., STIPIC, S., UNSAL, O., CRISTAL, A., VALERO, M. 2009. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *the 4th Workshop on Transactional Computing (TRANSACT'09)*.

KHAN, B., HORSNELL, M., ROGERS, I., LUJAN, M., DINN, A., ET AL. 2008. An object-aware hardware transactional memory system. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC'08)*. 93–102.

LUPON, M., MAGKLIS, G., AND GONZALEZ, A. 2008. Version management alternatives for hardware transactional memory. In *Proceeding of the 9th Workshop on Memory Performance : Dealing with Applications, Systems and Architecture*. 69–76.

LUPON, M., MAGKLIS, G., AND GONZALEZ, A. 2009. FasTM: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 293–302.

LUPON, M., MAGKLIS, G., AND GONZALEZ, A. 2010. A dynamically adaptable hardware transactional memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. 27–38.

MAGNUSSON, P., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HAILBERG, G., ET AL. 2002. Simics: A full system simulation platform. *IEEE Comput. 35*, 50–58.

MARTIN, M., SORIN, D., BECKMANN, B. M., MARTY, M. R., XU, M., ET AL. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News 33*, 92–99.

MCDONALD, A., CHUNG, J., CARLSTROM, B. D., MINH, C. C., CHAFI, H., ET AL. 2006. Architectural semantics for practical transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'06)*. 53–65.

MINH, C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the 4th IEEE International Symposium on Workload Characteristics (IISWC'08)*. 35–46.

MINH, C., TRAUTMANN, M., CHUNG, J. W., MCDONALD, A., BRONSON, N., ET AL. 2007. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. 69–80.

MOORE, K., BOBBA, J., MORAVAN, M. J., HILL, M. D., AND WOOD, D. A. 2006. LogTM: Log-Based transactional memory. In *Proceedings of the 12th IEEE Symposium on High Performance Computer Architecture (HPCA'06)*. 254-265.

POWELL, M. D., AGARWAL, A., VIJAYKUMAR, T. N., FALSAFI, B., AND ROY, K. 2001. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'01)*. 54–65.

RAJWAR, R. AND GOODMAN, J. 2002. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. 5–17.

RAJWAR, R., HERLIHY, M., AND LAI, K. 2005. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. 494–505.

RAMADAN, H. E. ROSSBACH, C. J., AND WITCHEL, E. 2008. Dependence-Aware transactional memory for increased concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*. 246–257.

ROSSBACH, C., HOFMANN, O., AND WITCHEL, E. 2010. Is transactional programming really easier. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. 47–56.

SHRIRAMAN, A., DWARKADAS, S., AND SCOTT, M. 2008. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. 139–150.

SHRIRAMAN, A., SPEAR, M., HOSSAIN, H., MARATHE, V. J., DWARKADAS, S., ET AL. 2007. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. 104–115.

TITOS, R., ACACIO, M. E., AND GARCIA, J. M. 2009. Speculation-Based conflict resolution in hardware transactional memory. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*. 1–12.

Titos-Gil, R., Negi, A., Acacio, M. E., Garcia, J. M., and Stenstrom, P. 2011. ZEBRA: A data-centric hybrid-policy hardware transactional memory design. In *Proceedings of the 25th International Conference on Supercomputing (ICS'11)*. 53–62.

Tomic, S., Perfumo, C., Kulkarni, C., Armejach, A., Cristal, A., et al. 2009. Eazyhtm: Eager-lazy hardware transactional memory. In *Proceedings the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 145–155.

Tremblay, N. and Chaudhry, S. 2008. A third-generation 65nm 16-core 32-thread + 32-scout-thread cmt sparc processor. In *Digest of Technical Papers of IEEE International Solid-State Circuits Conference (ISSCC'08)*. 82–83.

Yan, Z., Jiang, H., Feng, D., Tian, L., and Tan, Y. 2012. SUV:A novel single update version-management scheme for hardware transactional memory systems. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'12)*. 131–143.

Yen, L., Bobba, J., Marty, M. R., Moore, K. E., Volos, H., et al. 2007. LogTM-SE:Decoupling hardware transactional memory from caches. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. 261–272.

Zhao, L., Choi, W., and Drapper, J. 2012. SEL-TM: Selective eager-lazy management for improved concurrency in transactional memory. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'12)*. 95–106.

Received June 2012; revised September 2012; accepted November 2012