A Dynamically Adaptable Hardware Transactional Memory

Marc Lupon* mlupon@ac.upc.edu
Grigorios Magklis† grigorios.magklis@intel.com
Antonio González*† antonio.gonzalez@intel.com

* *Computer Architecture Department, Universitat Politècnica de Catalunya*
† *Intel Barcelona Research Center, Intel Labs Barcelona - UPC*

***Abstract*—Most Hardware Transactional Memory (HTM) implementations choose fixed version and conflict management policies at design time. While eager HTM systems store transactional state in-place in memory and resolve conflicts when they are produced, lazy HTM systems buffer the transactional state in specialized hardware and defer the resolution of conflicts until commit time. Each scheme has its strengths and weaknesses, but, unfortunately, both approaches are too inflexible in the way they manage data versioning and transactional contention. Thus, fixed HTM systems may result in a significant performance opportunity loss when they execute complex transactional applications.**

In this paper, we present DynTM (Dynamically Adaptable HTM), the first fully-flexible HTM system that permits the simultaneous execution of transactions using complementary version and conflict management strategies. In the heart of DynTM is a novel coherence protocol that allows tracking conflicts among eager and lazy transactions. Both the eager and the lazy execution modes of DynTM exhibit very high performance compared to modern HTM systems. For example, the DynTM lazy execution mode implements local commits to improve on previous proposals. In addition, lazy transactions share the majority of hardware support with eager transactions, reducing substantially the hardware cost compared to other lazy HTM systems. By utilizing a simple predictor to decide the best execution mode for each transaction at runtime, DynTM obtains an average speedup of 34% over HTM systems that employ fixed version and conflict management policies.

***Keywords*-hardware transactional memory; dynamically adaptable HTM; DynTM**

I. Introduction

Version and conflict management are the two most critical aspects of Transactional Memory (TM) systems that employ speculation to execute transactions concurrently. Version management defines where and how transactional modifications are kept, while conflict management details (a) when conflicts are detected, (b) how conflicts are resolved and (c) which actions are performed and by whom. These mechanisms can be implemented in software [8], [20], hardware [1], [7], [9], [16] or a combination of the two [6], [10], [19]. In this work, we focus on hardware TM (HTM) systems.

There are two main classes of version management schemes used in HTM systems: eager and lazy. Eager approaches keep new (speculative) values in-place in the memory hierarchy while old (pre-transactional) values are held *elsewhere*. The most popular eager systems buffer the old values in a different location in memory, using a software-managed log [14], [27]. On the other hand, lazy approaches keep the old values in the memory hierarchy while new values are invisible to the system until commit. Many lazy systems buffer the new values in-place in the processor's caches, but modify the coherence protocol to hide the modifications and to prohibit their propagation until commit time [9], [22].

Conflict management schemes are also classified as eager and lazy. Eager schemes detect and resolve conflicts at the moment that a load (store) instruction from an in-flight transaction accesses a memory location being written (read or written) by another transaction [1], [16]. Lazy schemes resolve conflicts at the time that a transaction wants to commit. In lazy schemes, conflict detection can take place early [22], [25] or it can be delayed until commit [7], [15]—after all, the conflict will not be resolved until commit time.

Nowadays, HTM systems implement fixed (either eager or lazy) version and conflict management mechanisms. The notable exception to this is FlexTM [22], a hybrid TM system that use software support to permit either eager or lazy conflict management. However, FlexTM still implements only lazy version management and it fixes the conflict management policy for the entire application execution.

Fixed-policy HTM systems are faced with several challenges that limit the concurrency of transactional workloads [3]. First, inflexible conflict management strategies have to prioritize between conflicting transactions. On the one hand, lazy HTM systems must abort all the transactions that conflict with the committing one, which (a) may result to starvation of the older transactions [2] and (b) it increases the amount of discarded transactional computation [21], [24]. On the other hand, eager HTM systems may abort a transaction multiple times, which may lead to different pathological situations [17]. Nevertheless, lazy transactions can avoid some read-after-write conflicts whereas eager transactions minimize discarded work in the case of write-after-

write violations by stalling conflicting requesters. Having a flexible version and conflict management scheme allows the system to select the policy that achieves the best performance on each situation.

Second, complex applications that combine small and large transactions with variable contention present a great challenge for HTM systems that fix the version and conflict management strategies for the whole program execution: while eager HTM systems can preserve the computation generated by long transactions in case of collision, lazy HTM systems are more effective in dealing with small, high-contention transactions. A truly flexible HTM that could select the ideal (eager or lazy) execution mode for each transaction at runtime would not be challenged by such complex situations.

In this paper, we propose the Dynamically Adaptable HTM (DynTM) system, an HTM implementation that tries to address the shortcomings of fixed HTM systems. By providing a fully-flexible version and conflict management implementation, DynTM allows the system to *dynamically* adapt its policies to best suit the application behavior. DynTM makes several contributions to the state of the art of HTM systems:

- DynTM is the first HTM system to allow the *simultaneous* execution of eager and lazy transactions, both in terms of version and conflict management. We propose a novel, unified transactional coherence protocol that, when tightly coupled with a new conflict resolution policy, enables safe execution of eager and lazy transactions.
- DynTM presents two high-performance execution modes for eager and lazy transactions. Especially, our proposal implements a new lazy execution mode that performs *local* commits and falls-back to eager mode when a transaction overflows the L1 cache.
- DynTM describes a runtime prediction scheme that decides, for each *dynamic* instance of a transaction, at what mode it should be executed according to its characteristics. The evaluation of DynTM in a cycle-accurate simulation environment shows that our proposal obtains a 34% average speedup compared to the fixed state-of-the-art HTM reference systems.

The remainder of the article is organized as follows. In Section II, we summarize related work on HTM. In Section III, we describe DynTM's hardware support and explain how the eager and lazy modes operate. In Section IV, we present the DynTM coherence protocol and show how eager- and lazy-mode transactions can be executed simultaneously, while in Section V we describe the DynTM execution mode predictor. In Section VI we evaluate our proposal and in Section VII we conclude the article.

II. Background in HTM systems

Many HTM implementations have been proposed since Herlihy and Moss introduced Transactional Memory as an alternative lock-free parallel programming model [9]. UTM [1] was the first eager HTM that supported unbounded transactions. By placing transactional modifications across the memory hierarchy and storing metadata on the side, it is able to execute transactions of any size or duration. LogTM [14] simplifies the version management mechanism by keeping the pre-transactional state in a software-managed log—which is restored on aborts—and using Read-Write cache bits [19] to eagerly detect conflicts. LogTM-SE [27] decouples the transactional state from caches by summarizing the memory accesses in signatures [4]. To accelerate the abort recovery phase and reduce the pressure on the write signature, FASTM [11] implements a hybrid version management mechanism.

Eager HTM systems present poor performance when they execute applications with a high number of conflicts. In these scenarios, eager designs may abort a transaction multiple times before it commits [2]. Moreover, most implementations require a backoff policy to avoid repetitive conflicts between aborting transactions [17]. On the other hand, eager approaches can preserve the computation generated on large transactions by stalling conflicting requesters, and they do not require additional commit actions [21].

Transactional Coherence and Consistency (TCC) [7] presents a novel consistency model based on *lazy* transactions. Committing transactions propagate their write set to the rest of the processors, which abort their transactions in case of conflict. In order to prevent the simultaneous commit of conflicting transactions, TCC requires a centralized arbiter. Chafi *et al.* proposed TCC-Scalable [5] to permit parallel commits. However, a TCC-Scalable implementation requires communication with all the directories in the read/write sets and a centralized agent to order the transactions. Pugsley *et al.* [15] improved the prior technique with a distributed arbitration mechanism based on the acquisition of directories. Eazy HTM [25] reduces even more the overheads of lazy commits by eliminating arbitration on non-conflicting transactions, but it still performs directory updates at commit time.

Most lazy HTM systems buffer the new values in-place in the processor's caches, but modify the coherence protocol to prohibit their propagation until commit. Hence, lazy approaches suffer considerable delays when hardware resources are overflowed—they fall-back to slow software solutions [6], [10] or require cumbersome hardware support [16], [22]. Moreover, lazy solutions must abort all the transactions that conflict with the committer, which (a) may result to starvation of older

transactions [20] and (b) increases the amount of transactional discarded computation [21]. Nevertheless, lazy HTM systems can avoid some read-after-write conflicts and can guarantee forward progress without applying a backoff policy. Therefore, lazy approaches are more effective when they deal with small, high-contention transactions [2].

Recent HTM designs have developed different alternatives to increase concurrency in the case of conflicts. DATM [18] bypasses transactional values to eliminate non-crossed WaW/RaW conflicts. Titos *et al.* [24] conceived an eager HTM that stalls younger committing transactions to avoid WaR conflicts. Unfortunately, the above proposals impose a strict order between conflicting transactions and can only eliminate *acyclic* dependences. Thus, these HTM systems experiment the same issues as conventional HTM systems when they execute transactions with crossed conflicts, which are common in typical transactional workloads [3].

In order to introduce some flexibility to TM systems, Shriraman *et al.* proposed FlexTM [22], a hybrid implementation [6], [10] that decouples conflict detection from conflict resolution by tracking transactional violations eagerly and delegating their resolution to the software. This dual-mode system permits the programmer to decide the conflict management scheme for the entire application (eager or lazy), but it requires (a) software decisions to resolve conflicts, (b) complex hardware to buffer transactional overflowed data (because it uses lazy version management) and (c) software commit arbitration for lazy transactions. What is more, FlexTM applies the same conflict management policy for the whole execution, being more restrictive than an HTM system that dynamically adapts the policy at the granularity of a transaction.

III. The DynTM Base System

DynTM offers two different execution modes: eager and lazy. The *eager* DynTM execution mode uses both eager version and conflict management. The *lazy* DynTM execution mode, on the other hand, uses both lazy version and conflict management. Moreover, DynTM permits eager- and lazy-mode transactions to execute *simultaneously* in the system. This is possible through UTCP, a novel unified transactional cache coherence protocol that is able to correctly track conflicts among transactions—independent of their execution mode—and it ensures the correct propagation of transactional modifications.

The DynTM eager execution follows a log-based approach. Transactional modifications are kept in-place in memory, where they are allowed to propagate to all levels of the hierarchy. The pre-transactional state is logged in a software structure [14]. In the eager mode, conflicts are resolved as soon as they are produced.

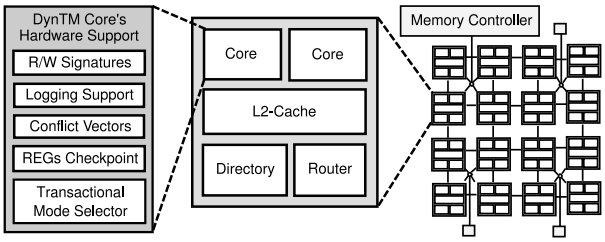

Figure 1. Base system configuration for DynTM

In contrast, the DynTM lazy execution mode resolves conflicts at the very end of a transaction. In the lazy mode, the speculative state is buffered in the L1 cache and is not made visible to the rest of the system until the transaction is committed. DynTM takes advantage of the built-in hardware support for eager version management in order to handle L1 cache overflows and context switches for lazy transactions. In such cases, the system will simply abort the lazy transaction and re-execute it in eager mode.

In order to further accelerate lazy transactions, DynTM implements *local* commits with core-to-core abort notification, avoiding expensive commit arbitration [15] and directory updates [25]. Moreover, DynTM employs a history-based hardware predictor to decide the execution mode (eager or lazy) on every new *dynamic* instance of a transaction. Predicting the application behavior at runtime permits the system to select the best-suited policy for each instance of a transaction, resulting in a significant performance improvement.

A. Hardware Support

In this work, we assume a CMP system with single-threaded cores and two levels of caches, where the L1 cache is private and the L2 cache is shared among all cores, as shown in Figure 1. Coherency is implemented using a blocking, distributed directory placed in the L2 cache. Besides the UTCP protocol, DynTM requires additional extensions to existing hardware components:

Logging Support: Like previous log-based HTM proposals [11], [14], [27], DynTM extends the core with software logging support to implement eager version management.

Signatures: DynTM requires Read and Write Signatures [4], [27] (Bloom filters) to track transactional accesses. While the Read Signature summarizes any transactional read, the Write Signature only contains adresses from eager transactional stores.

Conflict Vectors: Like in FlexTM [22] or Eazy HTM [25], DynTM introduces two bit-vectors per core to track conflicts among speculative transactions. While the Read Conflict Vector (RCV) tracks possible read violations, the Write Conflict Vector (WCV) tracks true write conflicts with remote transactions.

Transactional Mode Selector (TMS): Each core includes a TMS to decide the most profitable execution mode for each transaction. This hardware component uses past information of the *current* instance of a transaction and history from *previous* instances of the same transaction.

B. Eager Execution Mode

In DynTM, eager transactions follow the same hybrid data version management mechanism as the one presented in FASTM [11]. This mechanism guarantees that, if a transaction has not overflowed the L1 cache, the L2 cache will contain the correct pre-transactional state. This is done by writing-back a L1 *dirty* non-transactional cache line before overwriting it with transactional data. By keeping both the old and the new (transactional) state in-place in memory, DynTM offers a very fast abort recovery mechanism for transactions that do not overflow the L1 cache—it simply invalidates transactionally accessed lines.

Eager transactions also maintain the old state in a private, cacheable software log [14], which permits the safe eviction of consistent transactionally written lines. In case of overflow, the pre-transactional state can be recovered by a software routine (slow abort recovery mechanism). Moreover, transactional store operations always add their addresses in the Write Signature. Thus, the DynTM eager mode allows transactions to survive context switches and page faults by virtualizing the signatures and by using the software log for abort recovery [23] .

DynTM detects conflicts *early* with the help of the UTCP protocol. Conflicts are resolved using EE_{HP} [2], a high-performance conflict management policy that tries to avoid wasting computation by stalling transactions that issue conflicting requests. However, younger readers are aborted in order to minimize starvation for writing transactions. After aborting, an exponential backoff (based on the number of retries) is performed to guarantee the forward progress of eager transactions.

C. Lazy Execution Mode

Like other lazy HTM protocols, DynTM restricts transactional updates to the L1 cache only, maintaining pre-transactional values in the L2 cache. However, rather than requiring specialized hardware to handle L1 cache overflows [16], [22], DynTM aborts the offending transaction and re-executes it in eager mode. Moreover, DynTM implements *local* commits; a novel mechanism that avoids arbitration and communication at commit time [15], [25].

Lazy transactions also detect conflicts *early* via the UTCP protocol. Contrary to the eager execution mode, lazy transactions continue executing after detecting a conflict—conflicts are resolved *lazily* at commit time or

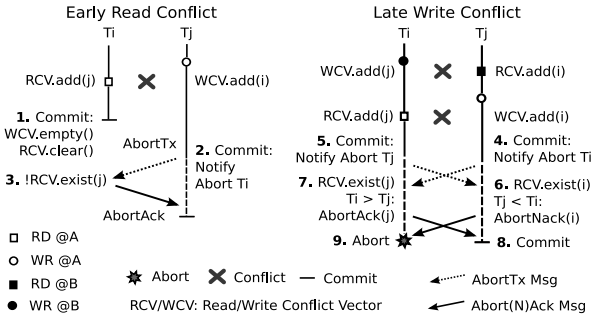

Figure 2. Local commits and abort notification in DynTM

until someone aborts the transaction. In order to track conflicts from their detection until their resolution time, DynTM transitions conflicting cache lines to special UTCP states, and marks conflicts among cores in the Read and Write Conflict Vectors (RCV and WCV).

In DynTM, when a lazy transaction attemps to commit, it probes its local WCV for conflicts with remote transactions. If the WCV is empty (*i.e.*, non-conflicting or read-only conflicting transactions), the core enters the commit phase. However, in case of conflict, the core enters the notification phase. In this phase, the core sends abort messages (*AbortTx*) to all the cores marked in its WCV and waits for their response. A core that receives an abort request must check both its RCV and WCV to verify that there is a conflict with the commiter. If so, the conflicting transaction is aborted and an *AbortAck* response is sent to the committer. Otherwise, the abort request is because of a conflict with a transaction that no longer executes on this core (either committed or aborted) and the request is ignored.

DynTM eliminates arbitration among lazy transactions, therefore two transactions may enter the notification phase at the same time. In order to prevent crossed conflicts, abort requests include a timestamp with the time a transaction started executing. When two transactions in the middle of their notification phase receive crossed abort requests, the younger transaction is aborted (it receives an *AbortNack* response). In the uncommon case that two transactions report the same timestamp, the transaction executed on the core with a higher CPU_{id} is aborted. Aborting committing transactions in their notification phase is safe to do because the memory state is not updated until the commit phase.

When all abort requests have been acknowledged, the notification phase ends. The core then enters the commit phase, when the core *locally* commits the transactional data in order to make it *globally* visible. This is done by transitioning all cache lines accessed transactionally to a non-speculative state (very cheap action with UTCP) and by clearing the local signatures and the Conflict Vectors. Unlike prior proposals, lazy DynTM does not require directory updates [5], [25] nor data movement [7], [15] at commit time.

	Lazy Reader	Lazy Writer
Eager Reader	No conflict	Speculate with the eager reader Abort eager if lazy commits first
Eager Writer	Abort lazy (immediately)	Abort lazy (immediately)

Table I
RESOLVING EAGER-LAZY CONFLICTS IN DYNTM

Figure 2 shows how DynTM executes lazy transactions. In the *Early Read Conflict* example, *Ti* is a read-only transaction that commits without conflict notification (step 1). When transaction *Tj* commits, it sends an *AbortTx* message to *Ti* (step 2). However, *Tj* does not appear in the Read Conflict Vector (RCV) of *Ti*, so *Ti* acknowledges the request and continues its execution (step 3). In the *Late Write Conflict* example, *Ti* and *Tj* are transactions with crossed conflicts that attempt to commit at the same time. Both transactions notify conflicts by sending abort messages (step 4-5), but only *Tj* successfully commits, because *Tj*'s timestamp is older than *Ti*'s (step 6-7). Both *Ti* and *Tj* wait until they collect all the replies from conflicting cores. *Tj* only receives *AbortAck* messages, therefore *Tj* moves to the commit phase, and *locally* commits the transaction (step 8). In contrast, *Ti* aborts as soon as it receives the *AbortNack* message from *Tj* (step 9).

IV. SIMULTANEOUS EXECUTION OF EAGER AND LAZY TRANSACTIONS

In order to simultaneously execute transactions with different version and conflict management schemes, DynTM uses a novel conflict resolution policy that preserves the consistency of eager transactions and, at the same time, shields lazy transactions from eager modifications that have overflowed the cache. We have decided to implement a conflict resolution policy that prioritizes eager transactions over lazy transactions. This policy favors large transactions that overflow the L1 cache and transactions with many lazy aborts.

In DynTM, lazy transactions cannot safely access the pre-transactional data of an eager transaction because, in the case of a transactional L1 cache eviction, eager transactions write-back the line in the L2 cache, poluting the pre-transactional values. For this reason, lazy transactions must abort when they access a memory location written by an eager transaction, since they cannot know if the L2 cache contains a pre-transactional or an evicted eager value.

Nonetheless, eager readers speculate when they conflict with lazy writers. When an eager transaction wants to read data that is written in a lazy transaction, the system will respond with the line from the L2 cache (lazy modifications are never evicted from the L1 cache, so the L2 cache always keeps the pre-transactional state) and mark a conflict in the eager transaction's RCV.

This policy avoids read-write conflicts if the eager transaction commits before the lazy transaction. If the lazy transaction commits first, then the eager transaction must abort. Notice that eager transactions only speculate with read data (the WCV remains empty), so abort notification at commit time is not required. Table I summarizes the conflict resolution policy between eager and lazy transactions.

A. The Unified Transactional Coherence Protocol

In the heart of DynTM lies a novel coherency protocol, the Unified Transactional Coherence Protocol (UTCP), that guarantees the correct propagation of transactional modifications, as well as the prompt detection of conflicts among transactions. The UTCP protocol distinguishes between coherent and speculative states. The coherent states include the four states of a typical MESI protocol, plus the T state. Cache lines in these states are either non-transactional or they are read inside a transaction and have no conflicts (M, E, S and I states), or they are written inside a transcation and they have no conflicts (T state). The two speculative R and W states keep transactionally read (R) or written (W) cache lines that have a conflict with one or more other transactions. Cache lines are transitioned to the R or W states only inside transactions that have a conflict with a lazy transaction—eager transactions are not allowed to speculate with their execution when they conflict with other eager transactions.

The coherent states T, M, E and S have a single owner or version in the system directory (multiple sharers are allowed, of course). On the other hand, speculative lines can have multiple active versions, therefore the directory must maintain a vector of owners. Conflicts among transactions are detected through the T and W states (for lazy transactional writers) and the Read and Write Signatures (for transactional readers and eager transactional writers, respectively). Figure 3 shows the UTCP states and transitions. The label of each transition shows the UTCP triggering message (before the slash) and the associated actions (after the slash). Following is a detailed description of the UTCP operations for same-mode transactions:

Non-conflicting Accesses [TLoad/TStore]: Non-conflicting eager or lazy memory accesses follow the TMESI protocol proposed in FASTM [11]. While transactional loads are performed as regular loads (adding the address in the Read Signature if they end successfully), transactional stores write-back the M-state lines to the L2 cache before transitioning to the T state. This guarantees that the L2 cache always has the correct pre-transactional value of the line.

Eager Conflicting Accesses [TLoad(E)/TStore(E)]: Assume two cores E_0 and E_1 executing eager transactions. When E_0 either misses in the L1 cache or

Request Message	Description
TLoad	Transactional (Tx) Load
TStore	Transactional Store
TGetS	Tx Read (Rd) Request (Req)
TGetX	Tx Write (Wr) Request
TGetSorX	Either Rd or Wr Tx Req
(T)GetS	Either Tx or Non-Tx Rd Req
(T)GetX	Either Tx or Non-Tx Wr Req
TGetS(E)	Conflicting Rd Req (Eager Tx)
TGetX(E)	Conflicting Wr Req (Eager Tx)
TGetS(L)	Conflicting Rd Req (Lazy Tx)
TGetX(L)	Conflicting Wr Req (Lazy Tx)
Local Signal	**Description**
Commit	Process Local Commit
Abort	Invalidate Tx Lines
Response Message	**Description**
Ack	Acknowledgement
Nack	Negative Acknowledgement (Eager Transaction)
Lack	Lazy Acknowledgement (Lazy Transaction)
Response Action	**Description**
Conflict	Lazy Conflict (Update Conflict Vectors)
AbortTx	Abort In-flight Transaction
OV	Notify Overflow (Abort Transaction if Lazy)
WB	L2 Cache Write-Back
Retry	Issue the Conflicting Request Again

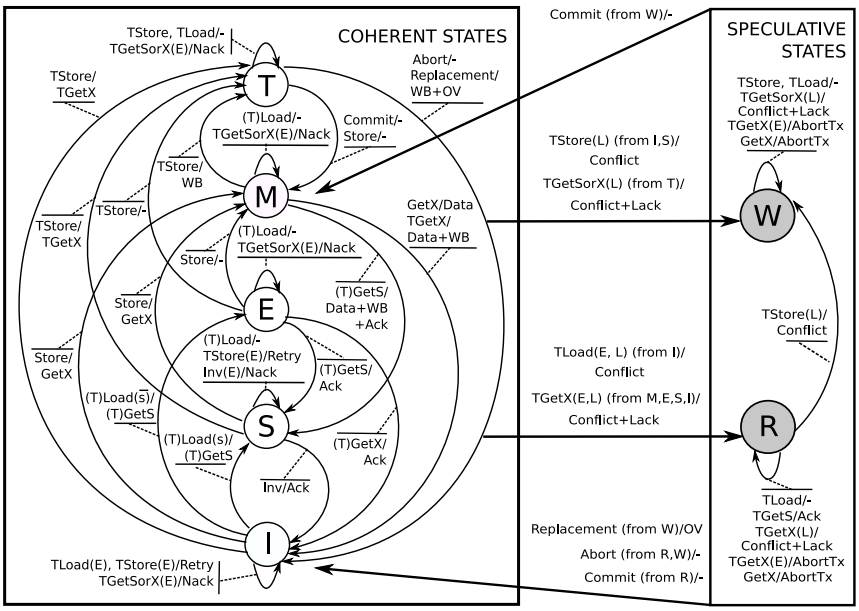

Figure 3. State-transition diagram of the unified transactional L1 cache coherence protocol

attemps to write shared data, it requests the line from the directory. The directory forwards the coherence request to the current owner (or sharers) of the line, which checks its signatures for conflicts. Assume that E_1 has accessed the line in its in-flight eager transaction. In this case, E_1 replies to E_0's conflicting request (*TGetS(E)* or *TGetX(E)*) with a *Nack* message. E_0 can either retry the memory access or abort the transaction.

Lazy Conflicting Accesses [TLoad(L)/TStore(L)]: Assume that core L_0, which is executing a lazy transaction, attemps to read (or write) a line that has been accessed by another lazy transaction executed on core L_1. In this scenario, L_0 requests the line to the directory, which (1) forwards the coherence request to the current owner(s) of the line and (2) sends a message containing the number of owners to L_0. When L_1 receives a lazy conflicting request (*TGetS(L)* or *TGetX(L)*), it replies to L_0 with a *Lack* message and moves the line to one of the speculative states. If L_1 has transactionally written the line before (*i.e.*, the line is in the T or W state), the line transitions to W and L_1 adds L_0 to its WCV. Similarly, if L_1 has transactionally read the line (*i.e.*, the line is in the M, E, S, I or R state), the line transitions to R and L_1 adds L_0 to its RCV.

When L_0 receives the *Lack* reply from L_1, it detects that there is a conflict. Hence, L_0's request is serviced by the L2 cache, which is guaranteed to have the correct pre-transactional values, and L_0 is added as a new line owner in the directory. L_0 puts the line in a speculative state (R for a *TLoad(L)*, W for a *TStore(L)*) and adds L_1 to its local RCV (*TLoad(L)*) or WCV (*TStore(L)*). This mechanism permits the system to identify inconsistent transactions that should be aborted before committing.

Local Commits and Aborts: Eager and lazy transactions employ *local* commits and aborts. When a core commits a transaction, it instantaneously transitions all the transactionally written lines (T or W) to the M state and invalidates all the R lines. When a core receives an abort notification, it invalidates all the T, W and R lines from its L1 cache. Notice that DynTM's commits or aborts do not require communication with the L2 cache or the directory.

Lazy Directory Updates: In DynTM, the directory is updated *lazily*, therefore committed lines may have multiple owners in the directory even though they may only exist in one L1 cache. Assume that L_0, L_1 and L_2 are cores that have written line A inside a lazy transaction. Eventually, L_2 commits, aborting L_0 and L_1. When L_0 re-executes, it requests line A from the directory. L_2 has not updated the directory at commit time, so the directory still maintains L_0, L_1 and L_2 as the owners of the line. Thus, the directory forwards the request to L_1 and L_2. While L_1 acknowledges the request (it has invalidated line A during its abort, so it does not own the line anymore), L_2 sends the committed data to L_0. After collecting all the responses, L_0 updates the directory by setting itself as a sharer (*TLoad*) or as the exclusive owner (*TStore*) of the line.

B. Combining Eager and Lazy Execution Modes

Now, assume that E_0 is a core executing an eager transaction and L_1 is a different core executing a lazy transaction. In order to explain how conflicts that involve eager and lazy transaction are resolved, we will describe the various situations in Figure 4.

Eager Early Write (Example 1): DynTM must prevent lazy transactions from reading or writing the modifi-

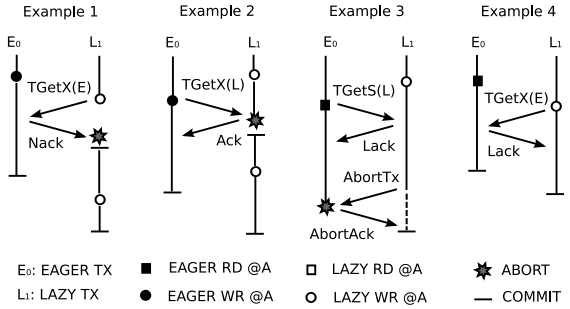

Figure 4. Resolving eager/lazy conflicts in DynTM

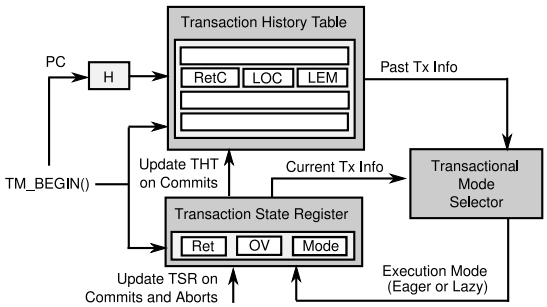

Figure 5. Hardware support for the Transactional Mode Selector

cations introduced by eager transactions. Thus, when L_1 attemps to access a line being modified by E_0, E_0 responds with a *Nack* message. After receiving the *Nack* response, L_1 aborts immediately.

Eager Late Write (Example 2): Similarly, upon a write request from E_0, L_1 acknowledges the request and aborts itself, permitting the eager transaction to obtain the pre-transactional data from the L2 cache. This is safe to do because lazy writes never leave the L1 cache. This approach reduces the amount of wasted computation on aborts and facilitates fast restarts, since lazy transactions do not require backoff cycles.

Eager Late Read (Example 3): When E_0 reads data that is written in L_1, L_1 responds with a *Lack* message. E_0 marks the conflict in its RCV, and L_1 marks the conflict in its WCV. E_0 receives the line data from the L2 cache and stores it in the R state. Since lazy modifications are never evicted from the L1 cache, E_0 gets the correct pre-transactional data. This policy avoids aborts from read-write conflicts when E_0 commits before L_1. Of course, if L_1 commits first, E_0 has to abort.

Eager Early Read (Example 4): Similarly, L_1 can continue its execution when it writes a memory location that has been read by E_0, tracking the conflict in its WCV. Hence, if L_1 commits before E_0, an *AbortTx* message is sent to E_0, which immediately aborts. Otherwise, if E_0 commits before L_1, no conflict is reported.

V. Transactional Mode Selector

In DynTM, each core includes a simple Transactional Mode Selector (TMS) to decide the most profitable execution mode for each transaction. The appropriate execution mode for a transaction is highly application-dependent. Lazy transactions usually manage contention more efficiently than eager transactions, especially when there are many small transactions with high contention. Nonetheless, eager transactions reduce the amount of discarded work due to aborts of large transactions. For this reason, the TMS decides to execute most of the transactions lazily, except in the case of multiple lazy-mode aborts or frequent overflows.

The TMS configuration shares similarities with typical two-level branch predictors [26]. As it can be seen in Figure 5, the TMS requires two hardware structures that store important information about past transactional executions. The first structure is the Transactional State Register (TSR), which collects information about the current *dynamic* instance of a *locally* executing transaction. The second structure is the Transactional History Table (THT), which records statistics from previously committed transactions on this core.

The TSR contains (a) the overflow bit (OV), which is asserted when the system aborts a lazy transaction due to an L1 cache overflow, (b) a 3-bit saturating counter (Ret) that counts how many aborts (*i.e.*, retries) the currently executing transaction has peformed, and (c) the Mode bit, which determines the execution mode of the current in-flight transaction. Each entry of the THT has two 2-bit saturating counters and a bit that contains the execution mode of the last committed instance of the transaction (LEM bit). The first counter (LOC) tracks if the transaction is prone to overflow while the latter (RetC) tracks if the transaction is prone to abort multiple times before committing.

At the beginning of a given transaction, the TMS decides the execution mode (eager or lazy) of the transaction and stores the decision in the Mode bit of the TSR. This decision is preserved until the transaction commits or aborts. Figure 6a shows how the execution mode is selected using the TMS. The TMS uses the TSR when the system re-executes an aborted transaction (Ret>0). In this case, DynTM changes the execution mode from lazy to eager when (a) the OV bit is asserted or (b) the number of transactional retries is above a threshold *T*. In our evaluation, the threshold *T* is a static parameter (the number of cores divided by eight). This technique permits our system to eliminate the *starvation of the older* pathology [2] and minimize the amount of discarded transactional computation [21].

When a new instance of a transacation starts (*i.e.*, not a re-execution), the TMS indexes the THT with the Program Counter (PC) of the transaction to decide the execution mode. If it hits in the THT, the TMS inspects the corresponding saturated counters. If previous instances of the same transaction have presented a reckognizable behavior (confident LOC or RetC counters), the TMS

(a) Execution Mode Predictor

```
if(Ret>0)
   if(OV == true || Ret > T || Mode == Eager)
      Mode = Eager
   else
      Mode = Lazy
else
   if(LOC == 3 || RetC == 3)
      Mode = Eager
   else if (LOC < 2 && RetC < 2)
      Mode = Lazy
   else
      Mode = LEM
```

(b) Transactional History Table Update

```
if(Ret > 2*T && RetC < 3){
      RetC++
else if(Ret < T/2 && RetC > 0)
      RetC--

if(OV == true && LOC < 3){
      LOC++
else if(OV == false && LOC > 0)
      LOC--

LEM = Mode
```

Figure 6. TMS selection (up) and THT update (down) algorithms

chooses between the eager (high counter values) or lazy (low counter values) execution modes. If the predictor is not confident on its decision, the TMS chooses the execution mode used in the last committed instance of the transaction (LEM bit). If there is a miss in the THT, the TMS executes the transaction lazily because lazy transactions usually obtain better performance than eager transactions. The THT is updated each time the core commits an instance of a transaction following the algorithm described in Figure 6b.

VI. Evaluation

For the evaluation of DynTM we assume a Chip Multiprocessor (CMP) with 32 cores, as shown in Figure 1. The system has a 16-node mesh interconnect with 64-byte links. Each node has two cores, a 1MB shared L2 cache and part of the directory. The system has four memory controllers to access 4GB of memory. Each core has 2Kbit Read and Write signatures, 32-bit Conflict Vectors (one bit per core) and a TMS with a 16-entry THT. Detailed system parameters are shown in Table II. The base system, the HTM support, and the UTCP coherence protocol have been simulated using the Simics [12] infrastructure from Virtutech and the GEMS [13] toolset from Wisconsin's Multifacet group. For our analysis, we execute applications from the STAMP benchmark suite [3] and two micro-benchmarks from the GEMS 2.0 distribution.

We have categorized these applications according to their characteristics. Low-contention applications (those with few conflicts) have been executed with 32 threads, whereas high-contention applications (those that scale poorly) have been executed with 16 threads. Table III

Core	32 cores, 1.2 GHz in-order, single issue, single-threaded
L1 cache	32 KB 4-way, 64-byte line, write-back, 2-cycle latency
L2 cache	16 MB 8-way, banked NUCA, write-back, 15-cycle latency
Memory	4 GB, 4 banks, 150-cycle latency
L2 directory	Bit vector of sharers/owners, 6-cycle latency
Interconnect	16-node Mesh, 64-byte links, 3-cycle latency
HTM Support	2 Kb Chuckoo-Bloom Signatures 32-bit CVs, TMS with 16-entry THT

Table II
Base system parameters

Bench	Suite	Input parameters	Category
Btree	μbench	25% insertions 100K Tx	Low Contention 32 threads
Deque		5K dummy work 100K Tx	
Genome	STAMP	32K seg. 512 gene. 32 len.	
Kmeans		15/15 clusters 16K points	
Ssca2		2^{14} nodes 9 edg. 9 len.	
Vacation		1M input, high contention	
Bayes		32 vars, 1024 records	High Contention 16 threads
Intruder		4K traf. 10 at. 16 pack	
Labyrinth		32*3*3 maze, 1024 routes	
Yada		20 angle, 633.2 mesh	

Table III
Input parameters of TM applications

presents the input parameters of the TM applications used in the evaluation. For our analysis, we have chosen to compare DynTM (labeled D in Figures 7 to 9) with four different HTM systems that require similar hardware support; two state-of-the-art HTM systems, and two DynTM alternatives. The four systems are the following:

Eager Fixed HTM (labeled E): This configuration corresponds to a state-of-the-art eager HTM system, which is similar to FASTM-Sig [11].

Lazy Fixed HTM (labeled L): This configuration shares similarities with state-of-the-art lazy HTM systems that use fixed version and conflict management policies [22], [25]. Our lazy HTM implements an infinite victim cache that "magically" buffers overflowed transactional data. Like the lazy mode of DynTM, it uses the Read/Write Signatures and the UTCP protocol to track conflicts, and it performs *local* commits.

Dynamic Overflow HTM (labeled O): This is the lazy mode of DynTM. In this configuration, we force DynTM to execute all transactions lazily. Only in case of L1 cache overflow a transaction switches to eager mode, but after committing, the next instance of the transaction will start again in lazy mode. We use this configuration to test the effectiveness of the non-adaptive lazy mode of DynTM.

Statically Programmed HTM (labeled S): Static alternative to DynTM where an expert programmer decides the execution mode of transactions. For our evaluation, we decided to execute all transactions lazily except those transactions that overflow the L1 cache or

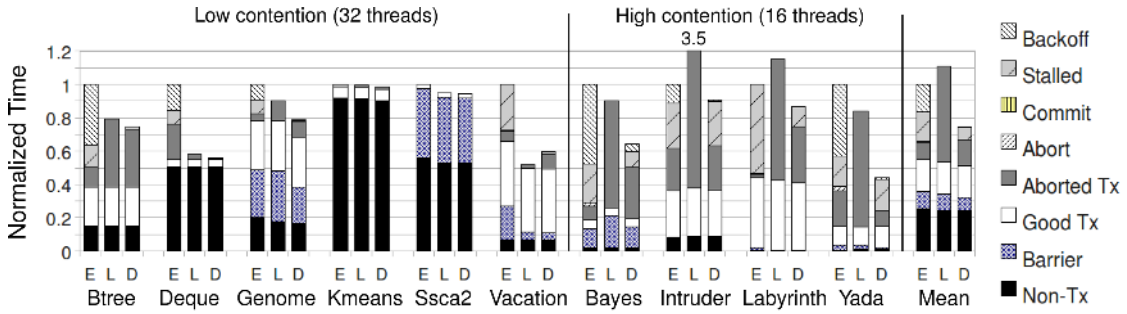

Figure 7. 32-threaded (left) and 16-threaded (right) normalized time of HTM systems

those transactions with many lazy aborts. We use this system to evaluate the performance benefit of the TMS predictor compared to a simpler adaptive method.

A. Performance Analysis of DynTM

Figure 7 presents the time distribution of the Eager Fixed (E), the Lazy Fixed (L), and the DynTM (D) HTM systems in their 32-threaded (low-contention applications, left side of the graphic) and 16-threaded (high-contention applications, right side of the graphic) executions. The execution time has been normalized to Eager Fixed and has been broken down to: non-transactional and barrier waiting cycles (labeled Non-Tx and Barrier), the time spent in committed transactions (labeled Good Tx) and in non-useful computation discarded on aborts (labeled Aborted Tx), the time spent in abort recovery and in *local* commits (labeled Abort and Commit), the time that eager transactions remain stalled after detecting a conflict (labeled Stalled) and the time that cores execute a backoff after aborting an eager transaction (labeled Backoff).

As it can be seen in Figure 7, DynTM outperforms both Eager and Lazy Fixed HTMs by (a) combining eager and lazy transactions in applications that execute heterogeneous transactions and (b) re-adapting the execution mode of the transactions at runtime. DynTM achieves, on average, a speedup of 34% over the Eager Fixed HTM and a speedup of 47% over the Lazy Fixed HTM. The reasons why DynTM outperforms state-of-the-art Fixed HTM executions are described in the following paragraphs.

On the one hand, eager HTMs—even the Eager Fixed HTM that implements high-performance conflict and version management policies—are not effective when collisions among threads are frequent. In our Eager Fixed HTM execution, transactions are stalled in case of conflict, which may lead to futile stalls (transactions that abort after being stalled for a long time) or cascades of stalls (transactions that are stalled by transactions that are waiting for other conflicts to be resolved). This behavior typically occurs in many-threaded applications with read-write conflicts, like *Btree* or *Vacation*, or in applications with long transactions like *Labyrinth*. On average, 18% of the eager execution time is spent in stalled transactions. Moreover, eager transactions utilize an exponential backoff that is based on the number of retries to spread the computation and avoid livelocks. Backoff is critical in high-contention applications with large transactions, like *Bayes* or *Yada*, or in applications with lots of aborts and small transactions, like *Deque* or *Genome*. On average, 16% of the eager execution time is spent in the backoff.

On the other hand, the Lazy Fixed HTM may abort older writers several times, which results to an important amount of discarded transactional work (5X more than the Eager Fixed HTM execution on average). This is critical in applications with large transactions, like *Intruder* or *Yada*. However, applications with small transactions and read-write conflicts, such as *Btree* or *Vacation*, improve their performance over the Eager Fixed HTM. This performance improvement is due to the speculative resolution policy that the Lazy Fixed HTM employs, which does not stall conflicting memory accesses nor requires backoff. Notice that our technique removes the commit phase from the critical path, spending less than 0.1% of the execution time in *local* commits.

DynTM uses the eager execution mode for transactions that commonly overflow the cache or for transactions that abort several times before committing. In the former case, DynTM can stall large transactions—those that modify lots of lines—to preserve useful work in case of conflict without requiring specialized lazy version management support. In the latter case, older transactions can commit faster and decrease the number of aborts because eager transactions have more priority than lazy transactions. Combining eager and lazy execution has a positive effect in applications with heterogeneous transactions like *Genome*, *Intruder* or *Yada*, which reduce the *Stalled* and *Backoff* cycles (with respect to the Eager Fixed HTM) and the *Aborted Tx* cycles (with respect to the Lazy Fixed HTM).

We want to point out that the Lazy Fixed HTM outperforms DynTM in *Vacation*. This workload suffers important performance issues when DynTM executes

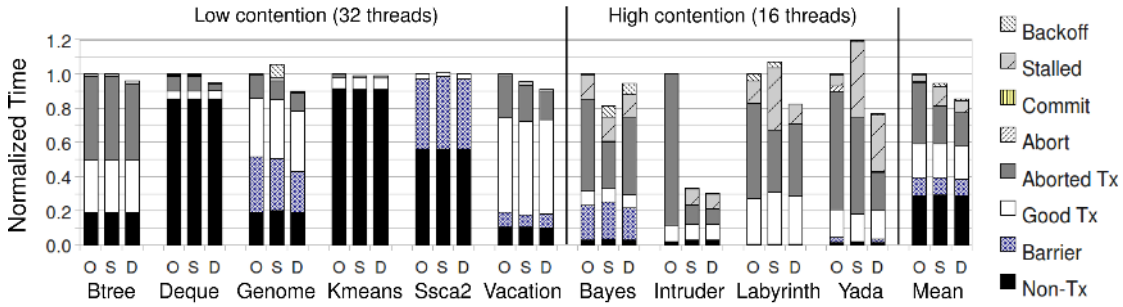

Figure 8. HTM systems normalized execution time

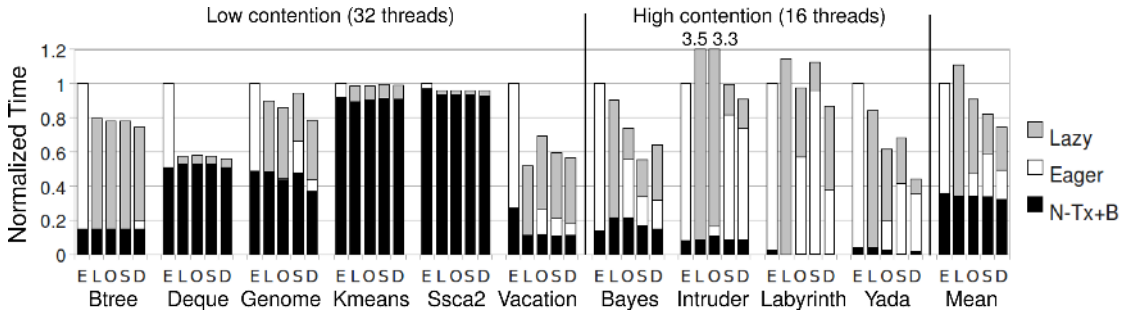

Figure 9. HTM systems time distributed by the execution mode

eagerly those transactions that overflow the L1 cache: eager transactions stall the requester in case of a read-write conflict, limiting the concurrency of *Vacation*'s transactions. Instead, the Lazy Fixed HTM can take advantage from the unbounded (non-realistic) transactional victim cache to keep executing *lazily* even in case of overflow, which permits the system to speculate with read-write conflicts.

B. Static and Dynamic Alternatives to DynTM

Figure 8 shows the distribution time of the Dynamic Overflow (O), the Statically Programmed (S) and the DynTM (D) HTM systems. As it can be seen in Figure 8, DynTM is the best of the three alternatives that combine eager and lazy transactions, achieving a 17% speedup over the Dynamic Overflow HTM and a 10% speedup over the Statically Programmed HTM.

The Dynamic Overflow HTM obtains good performance in applications with small transactions, like *Deque*, *Ssca2* or *Kmeans*, which do not suffer from cache overflows. Moreover, it also accelerates the execution of applications with huge transactions that overflow the L1 cache like *Bayes*, by restarting them in eager mode. However, the Dynamic Overflow HTM cannot combine execution modes on applications like *Intruder*, which starve older non-overflowed transactions. Moreover, overflowed transactions must abort before re-executing, which increases the amount of discarded transactional work in *Yada*.

The Statically Programmed HTM delegates the election of the execution mode to the programmer. The programmer has tried to minimize the impact of aborts caused by overflows (e.g., in *Bayes*) and to accelerate applications with multiple lazy aborts (e.g., in *Intruder*). However, applications that present a dynamic behavior (such as phase changes) may suffer considerable delays when we fix the execution mode of a transaction for the entire application. This happens in applications like *Genome*, *Labyrinth* or *Yada*, which present several overflows at the beginning of the execution and less overflows at their end.

Figure 8 shows the importance of having a dynamic execution mode selector. By re-adapting the system at runtime, DynTM can use the most profitable strategy at any time during the execution of a program. In contrast to Dynamic Overflow HTM, DynTM does not need to abort lazy overflowed transactions to restart them in eager mode, because it recognizes very quickly which transactions will probably overflow and decides to execute most of them eagerly right away. As opposed to the Statically Programmed HTM, DynTM executes eager transactions only when it is necessary (when lazy aborts are frequent), avoiding the use of conservative conflict management mechanisms for the entire application. The only scenario where the Statically Programmed HTM system performs better than DynTM is *Bayes*. This happens because *Bayes* executes transactions only a few times, which does not give enough time to our dynamic selector to learn the best execution mode for each transaction.

Figure 9 breaks down previous HTM execution times to the time spent in non-transactional code or barriers

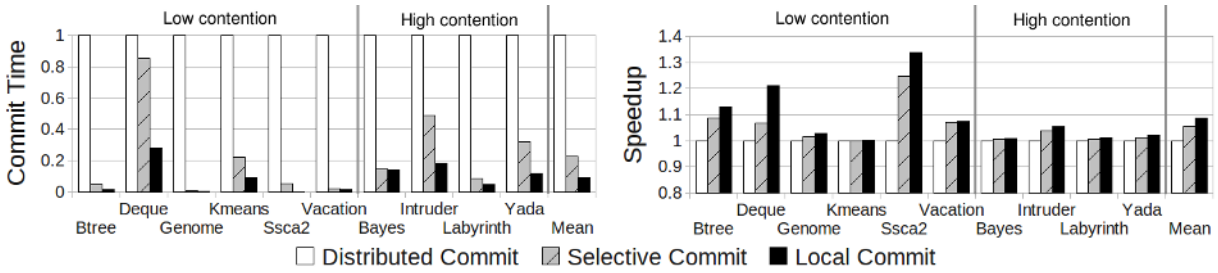

Figure 10. Commit time (left) and speedup (right) of Lazy Fixed HTM systems

(labeled N-Tx+B), the time spent in eager transactions (labeled Eager) and the time spent in lazy transactions (labeled Lazy). As it can be seen, DynTM executes most of the applications with small transactions lazily. This strategy is really useful because it eliminates read-write conflicts if the reader commits before the writer, does not require exponential backoff, and removes pathological behavior caused by stalled transactions. In contrast, DynTM chooses to run coarse-grain applications most of the time in eager mode, because (a) large transactions that overflow the cache do not support the lazy execution mode and (b) the DynTM eager conflict management policy reduces drastically the number of aborts and re-executions.

Figure 9 also shows the effectiveness of the adaptive mechanism that DynTM employs. While the Dynamic Overflow HTM spends most of the time in lazy transactions (only transactions that overflow the cache are executed eagerly) and the Statically Programmed HTM in eager transactions (the execution mode is designated by the programmer), DynTM can re-adjust its execution mode selection at runtime, distributing eager and lazy transactions in a more appropiate way in *Labyrinth*, *Vacation* or *Yada*.

C. Performance Analysis of Local Commits

Here, we compare the effectiveness of DynTM's *local* commits with two other mechanisms proposed for lazy transactions: *Distributed* and *Selective* Commits. Figure 10 shows the time spend on commits (left) and the speedup (right) of Lazy Fixed HTM systems that use different commit strategies normalized to the *Distributed* approach. In the *Distributed Commit* implementation [15], the system acquires the directory modules accessed during the transaction before making transactional writes globally visible. Hence, transactions that modify different directories can commit in parallel. In the *Selective Commit* implementation [25], the system only acquires directory modules when it commits a conflicting transaction. This fact permits a parallel commit on non-conflicting (or read-only conflicting) transactions. In our proposal (*Local Commit*), lazy transactions eliminate directory updates from the commit phase. All three HTM implementations use the abort notification mechanism to report remote conflicts.

DynTM performs *local* commits, a technique that eliminates the communication with shared resources at commit time. This is especially helpful in applications with read-only transactions, like *Btree* or *Vacation*, or in applications with tiny-size transactions, like *Deque* or *Ssca2*. Figure 10 shows that *local* commits accelerate the commit phase of the *Distributed* approach by a factor of 11 and of the *Selective* approach by a factor of 2.5. In environments that execute small transactions, the usage of local commits report significant benefit, improving up to a 20% the performance of the *Distributed Commit* implemention.

VII. Conclusions

In this paper we have presented DynTM, the first fully-flexible HTM that permits the simultaneous execution of transactions with distinct version and conflict management strategies, either eager or lazy. The tight coupling and correct functioning of the two exection modes is achieved by the UTCP protocol, a novel cache coherence protocol that ensures consistency among eager transactional modifications that are propagated in the memory hierarchy and lazy transactional modifications that are hidden in the processor's private caches. What is more, DynTM is the first HTM to offer a lazy execution mode that uses a log-based HTM as a fallback mechanism in order to survive cache overflows and context switches, with the added benefit of reduced hardware cost.

Both DynTM's eager and lazy mode implementations offer very good performance. Our simulations show that the eager DynTM mode is as fast as an accelerated log-based HTM system, while the lazy mode beats the performance of other lazy HTM implementations due to the new commit technique that it implements. Finally, when coupled with the Transactional Mode Selector, a history-based predictor that takes advantage of the flexibility offered by DynTM to decide the best execution mode for each transaction at runtime, DynTM obtains an average speedup of 34% over HTM systems that employ fixed version and conflict management mechanisms.

ACKNOWLEDGEMENTS

This work is partially supported by the Generalitat de Catalunya under grant 2009SGR1250, the Spanish Ministry of Education and Science under contracts TIN2007-61763 and TIN2010-18368, and Intel Corporation. Marc Lupon is supported by an UPC-Research grant.

REFERENCES

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *Procs. of the 11th Intl Symp on High-Performance Computer Architecture*, Feb. 2005.

[2] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance Pathologies in Hardware Transactional Memory," in *Procs. of the 34th Intl Symp on Computer Architecture*, Jun. 2007.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Procs. of The IEEE Intl Symp on Workload Characterization*, Sep. 2008.

[4] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *Procs. of the 33th Intl Symp on Computer Architecture*, Jun. 2006.

[5] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A Scalable, Non-blocking Approach to Transactional Memory," in *Procs. of the 13th Intl Symp on High-Performance Computer Architecture*, Feb. 2007.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid Transactional Memory," in *Procs. of the 12th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *Procs. of the 31st Intl Symp on Computer Architecture*, Jun. 2004.

[8] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software Transactional Memory for Dynamic-Sized Data Structures," in *Procs. of the 22nd ACM Symp on Principles of Distributed Computing*, Jul. 2003.

[9] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Procs. of the 20th Intl Symp on Computer Architecture*, May 1993.

[10] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid Transactional Memory," in *Procs. of the ACM SIGPLAN Symp on Principles and Practice of Parallel Programming*, Mar. 2006.

[11] M. Lupon, G. Magklis, and A. González, "FASTM: A log-based hardware transactional memory with fast abort recovery," in *Procs. of the 18th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep. 2009.

[12] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, 2002.

[13] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, 2005.

[14] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *Procs. of the 12th Intl Symp on High-Performance Computer Architecture*, Feb. 2006.

[15] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian, "Scalable and reliable communication for hardware transactional memory," in *Procs. of the 17th Intl Conf on Parallel Architectures and Compilation Techniques*, Oct. 2008.

[16] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," in *Procs. of the 32nd Intl Symp on Computer Architecture*, Jun. 2005.

[17] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "MetaTM/TxLinux: Transactional Memory for an Operating System," in *Procs. of the 34th Intl Symp on Computer Architecture*, Jun. 2007.

[18] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *Procs. of the 41st Annual Intl Symp on Microarchitecture*, Nov. 2008.

[19] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, "Architectural Support for Software Transactional Memory," in *Procs. of the 39th Annual IEEE/ACM Intl Symp on Microarchitecture*, Dec. 2006.

[20] W. N. Scherer III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Procs. of the 24th ACM Symp on Principles of Distributed Computing*, Jul. 2005.

[21] A. Shriraman and S. Dwarkadas, "Refereeing conflicts in hardware transactional memory," in *ICS '09: Procs. of the 23rd Intl Conference on Supercomputing*, Jun. 2009.

[22] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible Decoupled Transactional Memory Support," in *Procs. of the 35th Intl Symp on Computer Architecture*, Jun. 2008.

[23] M. M. Swift, H. Volos, N. Goyal, L. Yen, M. D. Hill, and D. A. Wood, "OS Support for Virtualizing Hardware Transactional Memory," in *Procs. of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2008.

[24] R. Titos, M. E. Acacio, and J. M. Garcia, "Speculation-based conflict resolution in hardware transactional memory," in *Procs. of the 23rd Intl Parallel and Distributed Processing Symposium*, May 2009.

[25] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "Eazyhtm, eager-lazy hardware transactional memory," in *Procs. of the 42nd Intl Symp on Microarchitecture*, Dec. 2009.

[26] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Procs. of the 25th Intl Symp on Computer Architecture*, Jun. 1998.

[27] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *Procs. of the 13th Intl Symp on High-Performance Computer Architecture*, Feb. 2007.