EazyHTM: Eager-Lazy Hardware Transactional Memory

Saša Tomić[†‡] Cristian Perfumo[†‡] Chinmay Kulkarni[†⋆] Adrià Armejach[†‡]
Adrián Cristal[†] Osman Unsal[†] Tim Harris[⋆] Mateo Valero[†]
[†]BSC-Microsoft Research Centre [‡]Universitat Politècnica de Catalunya
[⋆]BITS Pilani [⋆]Microsoft Research Cambridge

ABSTRACT

Transactional Memory aims to provide a programming model that makes parallel programming easier. Hardware implementations of transactional memory (HTM) suffer from fewer overheads than implementations in software, and refinements in conflict management strategies for HTM allow for even larger improvements. In particular, lazy conflict management has been shown to deliver better performance, but it has hitherto required complex protocols and implementations.

In this paper we show a new scalable HTM architecture that performs comparably to the state-of-the-art and can be implemented by minor modifications to the MESI protocol rather than re-engineering it from the ground up. Our approach detects conflicts eagerly while a transaction is running, but defers the resolution lazily until commit time. We evaluate this EAger-laZY system, EazyHTM, by comparing it with the Scalable-TCC-like approach and a system employing ideal lazy conflict management with a zero-cycle transaction validation and fully-parallel commits. We show that EazyHTM performs on average 7% faster than Scalable-TCC. In addition, EazyHTM has fast commits and aborts, can commit in parallel even if there is only one directory present, and does not suffer from cascading waits.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Performance

Keywords

EazyHTM, transactional memory

1. INTRODUCTION

Transactional Memory (TM) is a new technology which aims to make parallel programming easier. TM is based on the idea of speculatively running "transactions" of memory accesses in parallel, while guaranteeing exactly-once semantics as if the transactions were run in a serial order. If conflicts are detected during the execution, some of these transactions are aborted to maintain consistency.

To be effective, a TM implementation should provide high-performance and scalability. The two key design questions are: (i) How should speculative writes be handled: are they written to shared memory eagerly, while keeping an undo log so that they could be reversed, or are they buffered and published only at commit time? (ii) How should conflicts be detected and resolved: at the moment the problematic read or write is attempted, or deferred to commit time?

In this paper we are chiefly concerned with the latter question. To see how conflict management can affect the performance of a TM system, consider the example shown in Figure 1 (inspired by the work of Spear *et al.* [19]).

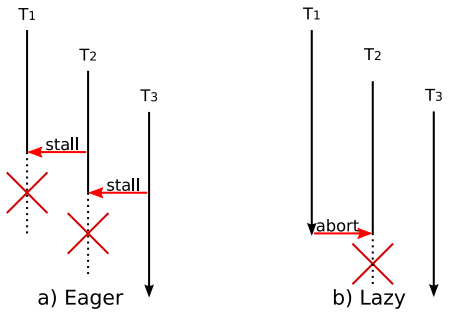

Figure 1: Why lazy conflict detection performs better under contention?

Eager conflict detection (Figure 1a) attempts to minimize the amount of wasted work performed in the system. Here, transaction T_1 conflicts with T_2, and is stalled. Then, T_2 later conflicts with T_3, and gets stalled too. Note that though T_1 does not conflict with T_3, it must stall until T_3 (and then T_2) either aborts or commits. Most eager HTM implementations suffer from these so-called *cascading waits* [4]. With lazy conflict detection (Figure 1b), all transactions execute until a transaction attempts to commit. In the example, when T_1 attempts to commit, it only aborts T_2. Once T_2 aborts, T_3 can also commit without conflicts.

In the above example, we make two observations. First, lazy conflict resolution allows two transactions to commit, while eager resolution allows only one. This difference is due to a fundamental facet of eager conflict resolution: it must

MICRO'09, December 12–16, 2009, New York, NY, USA.

address *potential* conflicts (caused by an offending access to a shared location), while lazy resolution deals with conflicts that are *unavoidable* in order to allow a transaction to commit. Second, attempts by eager systems to reduce "wasted" work are not always successful. In Figure 1a, for instance, the eager system stalls T_1. Since T_1 does not eventually abort, the work that was avoided had not been wasteful. Even if T_1 did abort, the amount to work saved would be minimal, due to the small size of transactions.

In summary, eager HTM systems can suffer from the following problems:

1. They must speculate which transaction is more likely to commit (and which should be aborted) when an offending access is attempted. At this time, the system has little information, but needs to speculate, which is inherently suboptimal. Solving this problem accurately requires sophisticated algorithms similar to [17].

2. Even if a successful prediction is made, a chain of waiting transactions still cause a cascading wait and the system needs to avoid deadlocks that may arise out of such (cascaded) stalls. Alternatively, if conflicts result in aborts, performance degrades due to unnecessary aborts.

Previous research has illustrated how *lazy* conflict detection can allow more parallelism [4,15]. Delaying the resolution of conflicts to commit time avoids making the difficult decision of which is the best transaction to abort. This can simplify the system. Furthermore, lazy conflict detection can result in higher performance than eager conflict detection when the amount of "wasted work" saved by eager conflict detection is offset by the number of unnecessary aborts that eager conflict detection incurs.

Traditionally, the tasks of *conflict detection* and *conflict resolution* are often conflated: they are either both performed eagerly, or both performed lazily at commit-time.

The main idea of this paper is to separate these two tasks; we perform conflict detection concurrently with a transaction's execution, but defer conflict resolution until either the transaction tries to commit, or until a conflicting transaction actually commits (at which point the tentative conflict becomes unavoidable). Unlike traditional eager conflict detection and resolution this means there is no need to anticipate which transaction is more likely to commit. Unlike traditional lazy conflict detection this means we can avoid commit-time validation (so overheads are lower). We call this an *"EAger-laZY"* approach, and our design EazyHTM.

With EazyHTM, we make several contributions:

- EazyHTM is the first instance of a transactional memory with eager conflict detection that defers conflict resolution until commit time that is implemented completely in hardware. Besides having the advantage of knowing the conflicts during the transaction execution, EazyHTM has low requirements on the underlying system. In particular, we do not require that directories are split for parallelism [7], or that a specific interconnection topology is used [8].

- Even with our pure-HTM protocol, EazyHTM allows non-conflicting transactions to commit in parallel. Moreover, commits appear to be instantaneous to other running transactions and aborts are instantaneous and almost single cycle instructions. Furthermore, these improvements are provided while still guaranteeing that pathological behavior [4] such as livelocks ("Friendly-Fire"), starvation for writing transactions ("Starving-Writer"), serialized commit or cascaded waits never occur.

- Our system takes advantage of core-to-core interconnection links to manage conflicts. This reduces the directory-core traffic, simplifies the design of the directory (which is largely unmodified), and allows greater parallelism. This core-to-core interconnection traffic is still small enough to have a practically null impact on performance, for most benchmarks (see Section 5).

In Section 2, we introduce the basic EazyHTM protocol. Optimizations and extensions are shown in Section 3. Microarchitectural modifications are explained in Section 4. Experimental results are discussed in Section 5. We discuss related work in Section 6 and in Section 7 we present conclusions and future work.

2. EAZYHTM: BASIC PROTOCOL

The EazyHTM protocol operates by cores sharing information within each other on every possible conflict, but not immediately aborting or stalling a transaction.

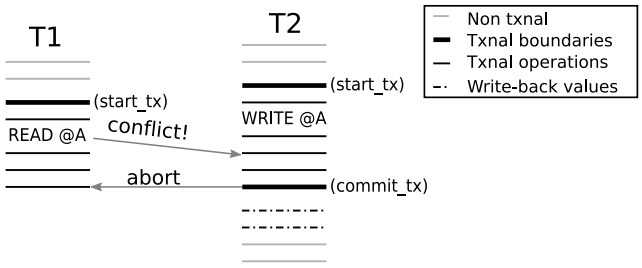

Figure 2: EazyHTM conflict detection and resolution: conflicts are detected eagerly, but transactions continue "racing" until one of them commits. The first to commit aborts the racing transaction.

In Figure 2, a transaction T_1 reads a cacheline that has been speculatively modified by transaction T_2. The transactions detect this situation, and note the conflict until they terminate their execution. Conflicts are always in one direction, i.e. the transaction that modifies a cacheline has a conflict with, and can abort, the one that reads the same line.

Since all conflicts are detected while a transaction is running, once a transaction (say T_2) is ready to commit, it knows exactly which transactions need to be aborted to maintain the system consistency. Therefore, an abort message is sent to all the conflicting transactions and once all conflicting transactions confirm their abort, speculatively written values are published. Both the abort request and the acknowledge are sent over the core-to-core interconnect to the corresponding core. The messages do not have to pass through a centralized router, instead they hop from one core to another until they get to the destination. Conflict resolution only requires the participation of processors involved in the conflict and does not involve the directory.

2.1 Conflict Detection

EazyHTM bases its conflict detection on the existing cache coherency functionality, currently used for non-transactional code to ensure that no races occur on shared accesses. Concretely, in a directory based implementation of cache coherence, this extension of the functionality only requires that the directory responds to one new message corresponding to a transactional access. Like any other directory protocol, this protocol is completely transparent to running code.

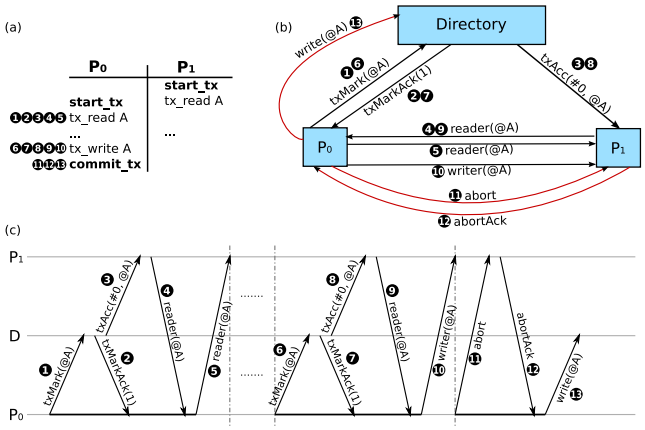

Figure 3: Messages for conflict detection and resolution – three memory accesses, one commit and one abort message: (a) Concurrent conflicting transactional executions on processors P_0 and P_1; (b) Messages exchanged between two conflicting transactions; messages are numbered in the order they are sent; (c) Time diagram of the message exchange, with time going from left to right. Thick horizontal line segments on P_0 line mark a single executed instruction.

When a transaction accesses a line, it sends the directory a special request $txMark(@addr)$, regardless of the access being a read or a write. The directory handles this request almost like a read request in an ordinary directory protocol. It marks the read for the line, and after allocating the line in shared mode, responds with an acknowledgement indicating how many other sharers the line has.

We illustrate message flow and conflict detection through an example of two transactions running on the processors P_0 and P_1(Figure 3). In this example, the transaction in P_0 starts, and performs a read which does not conflict with the read in P_1. It then does a write which conflicts with the transaction in P_1. Finally the transaction in P_0 commits while aborting the transaction running in P_1.

When P_0 executes `tx_read A` a $txMark(@A)$ message is sent to the directory ❶. As with typical MESI protocols, a processor only sends $txMark(@addr)$ on the first access to the line in the current mode (read or write). For subsequent transactional access to the same line, the processor uses values in its private cache. If the core has previously sent a *txMark* due to a transactional read and now requires to write a value, it resends the message to detect potential new races.

The directory first acknowledges P_0 with a $txMarkAck(1)$ message, where the parameter "1" indicates the current number of accessors for that cacheline ❷ and then sends a $txAccess(\#0, @A)$ message ❸ to all the other accessors, in this case P_1. This is possible since the directory keeps track of all speculative (transactional) or non-speculative accesses to all cachelines. As P_1 previously accessed the cacheline A, the directory knows the list and the count of all cacheline sharers. In the following text we are going to use term "accessor" instead of a "sharer", to represent both non-transactional and transactional cacheline sharers, which might have read and/or modified the line.

On receiving $txMarkAck(1)$, P_0 waits until it receives the specified number of messages from all other accessors (in this case, it waits until one message is received).

Meanwhile, $txAccess(\#0, A)$ initiates a point-to-point communication between old accessors, P_1, and the new one, P_0. Note that P_1 knows that the new accessor is P_0 because of the first parameter in the *txAccess* message. The list and explanation of all the messages that can be interchanged between processor cores is given in Section 2.5.

Continuing with the example of Figure 3, P_1 informs P_0 that it is a reader of the line A by sending the $reader(@A)$ message ❹. When P_0 receives a message, it sends a response message $reader(@A)$ to P_1 ❺. Now both transactions know the exact access mode of both transactions for the line, and both of them know whether there is a conflict between them or not. In this specific case, since both accesses are reads, there is no conflict between transactions.

In the example, a conflict occurs when P_0 executes the `tx_write A` instruction. P_0 sends a *txMark* message ❻ to the directory, which causes the directory to send exactly the same messages as in the non-conflicting situation, ❼ and ❽. At this moment, a point-to-point communication starts again, with P_1 sending a $reader(@A)$ message to P_0 ❾. P_0 responds with its access mode to the line, and sends a $writer(@A)$ message to P_1 ❿.

2.2 Tracking Possible Conflicts

The *racers-list* on processor P_i maintains a list of other processors that run transactions which conflict with P_i's current transaction. This over-approximates the set of transactions that need to be aborted when P_i's transaction commits (e.g. a conflicting transaction on another processor P_j may have aborted, and a different non-conflicting transaction started in its place).

To avoid false-aborts in this kind of case, each processor maintains a *killers-list* of processors that are allowed to abort its transaction.

Both the racers-list and the killers-list must be cleared at the start of each transaction.

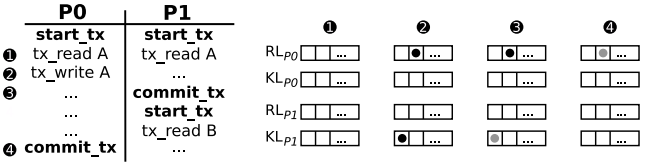

Figure 4: Racers-list (RL) and killers-list (KL). The racers-list records transactions that need to be aborted when this transaction commits. The killers-list records transactions that have the permission to abort this transaction.

Figure 4 presents an example with races. Initially, both

racers-list and killers-lists are empty on both processors. After event ❶, the lists are still empty as both accesses were reads. When P_0 executes the write instruction ❷, it receives a *reader*(@A) message from P_1. This adds P_1 to P_0's racers-list. Also, since P_1 gets a *writer*(@A) message from P_0, it adds P_0 to its killers-list ❷.

After P_1 commits the current transaction ❸ and starts a new transaction, the killers-list is cleared. This prevents P_0 from aborting the next transaction running on P_1 ❹, unless a new race is established.

2.3 Committing a Transaction

At commit time, the racers-list and killers-list provide the information for a transaction to know which other transactions it is conflicting with it; further commit-time validation is not required (unlike HTMs which use full lazy conflict management HTMs [7, 8]). With EazyHTM, when a transaction is ready-to-commit, it must ensure that all the transactions from its racers-list have terminated, either by being aborted or committed.

Commit with conflicts and aborts: When the transaction running on P_0 in Figure 3 reaches the commit instruction, it has to abort all the conflicting transactions in order to ensure isolation. When P_0 is ready to commit, it first sends an *abort* message to all processors from its racers-list (here, P_1) ⓫. P_1 aborts only if P_0 is in its killers-list. However, in both cases P_1 sends an acknowledge ⓬ to P_0's abort request.

Once P_0 has received *abortAck*s from all conflicting cores, it enters the committing state where it is guaranteed to commit successfully. During this period the transaction cannot be aborted and responds to all possible killers with an *abortNack*.

The processor writes all speculatively modified cachelines serially to the shared memory in the usual manner: acquires exclusive access from the directory, for each line in its write-set ⓭, which in turn invalidates copies held by all other accessors. After publishing all the cachelines, it exits the committing procedure and continues normal execution. Also, see Section 3 for optimizations for this process.

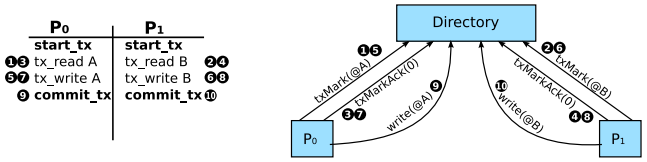

Figure 5: Committing without conflicts: transactions accessing different cachelines do not incur any extra communication between them.

Commits without conflicts: In case no conflicts are present, EazyHTM allows all non-conflicting transactions to commit in parallel. Figure 5 shows the execution of two non-conflicting transactions running on P_0 and P_1. Since the transactions datasets are disjoint, the directory does not send any *txAccess* message to processor cores during executions. Therefore, there are no core-to-core messages between P_0 and P_1. Both transactions have empty racers-lists at the moment of commit (not shown in figure).

Figure 5 also shows the low overhead EazyHTM imposes on non-conflicting accesses. In particular, a non-conflicting read/write results in the same number of messages as a normal, non-transactional read/write.

Racing commits: Though uncommon in practice, it is possible for multiple racing transactions with mutual races to reach the commit instruction at *exactly* the same time. In this case, transactions would receive an abort request from a transaction that they just sent an abort request to (and did not receive an acknowledgement from). One of the transactions must now abort to allow the other to proceed. EazyHTM breaks ties in this case by allowing the transaction running on a lower cpuid to win and commit. This transaction sends an *abortNack* to the transaction running on the core with the higher cpuid, which responds with an *abortAck* and aborts itself. This situation is extremely rare, so we use a simple criterion. Note that progress is still guaranteed. Random, round robin and a number of other tie-breaking policies may be easily added to eliminate the possibility of pathological cases leading to starvation.

2.4 Aborting a Transaction

With EazyHTM, transactions may only be aborted for one of the following reasons:

1. In response to an abort request sent by another transaction.
2. On exceptions or interrupts. In general, hardware transactions are small enough to complete between occurrences of exceptions or interrupts. Even TLB misses, which are unavoidable, become less important as the TLB warms up, and do not significantly affect our system.
3. When non-transactional code modifies a cacheline being accessed in a transaction: this allows us to support strong atomicity [3]. The feature is provided by detecting a cache coherency invalidation message from the directory and aborting if a part of the transaction gets invalidated by the directory.

Aborting a transaction in EazyHTM discards all the speculatively performed updates and restarts the transaction execution. Since we implement lazy version management, caches can quickly invalidate all speculative changes. The racers-list and killers-list (see section 2.2) are also cleared on abort.

Once all speculative changes are discarded and the lists cleared, the register file is restored to its previous state, just before the beginning of the transaction, and the control flow is reset to the first instruction.

2.5 Protocol State-Message Table

For completeness, we present a complete transactional state table of a processor core (Table 1). Each cell in the table describes the actions performed by a core upon receiving a message, depending on its current transactional state. The rows represent current state. The columns represent the incoming message from another processor core or from a directory. Dashes indicate impossible combinations. We define each CPU core state as follows:

- *Active*: A transaction is being executed on the core.
- *Ready to commit*: The transaction has executed all the code within the atomic block and is in the process

State/Message		reader @	writer @	rdwr @	nonTXnal @	tryLater	txAccess # @	abortAck	abortNack	abort
Active	No @	-	-	-	-	-	MSG nonTXnal #	-	-	MSG abortAck, if sender_id in KL: ABORT
	R @	MSG reader⋆	MSG reader⋆; SKL	MSG reader⋆; SKL	nop	REDO INSTR	MSG reader @	-	-	
	W @	MSG writer⋆; SRL	MSG writer⋆	MSG writer⋆; SRL	nop	REDO INSTR	MSG writer @	-	-	
	RW @	MSG rdwr⋆; SRL	MSG rdwr⋆; SKL	MSG rdwr⋆; SRL; SKL	nop	REDO INSTR	MSG writer @	-	-	
Ready to commit		-	-	-	-	-	MSG tryLater	CRL if RL == 0, then enter committing	Wait all pending abortAck/nack and ABORT	if my_id > sender_id then MSG abortAck, ABORT else CRL, MSG abortNack
Committing	No @ / R @	-	-	-	-	-	MSG nonTXnal #	-	-	if sender_id in KL: MSG abortNack else MSG abortAck
	W @	-	-	-	-	write, MSG nonTXnal #	-	-		
NonTx/ Aborting		-	-	-	-	-	MSG nonTXnal #	-	-	MSG abortAck

Table 1: State table for a core, showing the current state (left), and incoming message (top). Each cell shows the action to be performed for a cacheline address `@` and core ID `#`: `SRL` (set racers list), `SKL` (set killers list), `CRL` (clear racers list), `MSG` (send a message).

of aborting all racing transactions. Lasts between the beginning of the *commit* instruction and the reception of the last *abortAck* or *abortNack* (if any).

- *Committing*: The processor core has aborted all racing transactions and is now committing speculative changes. Once entered in this state, the transaction is invincible: it cannot be aborted.
- *No Tx*: The core is not executing transactional code.
- *Aborting*: The transaction has received an *abort* message and is processing it, i.e. flushing speculative changes.

In the following text we use symbol "@" for cacheline address, and symbol "#" for processor core ID. Some messages are related to one cache line and different actions may be taken depending on whether this line is present in the read-set, write-set or neither. Therefore, the states *Active* and *Committing* have sub-states:

- *R @*: The transaction has read @ speculatively, i.e. @ is in its read-set.
- *W @*: The transaction has modified @ speculatively, i.e. @ is in its write-set.
- *No @*: Neither *R @* nor *W @* bit is set
- *RW @*: Both *R @* and *W @* are set

The sub-states *R @* and *W @* are set in the private cache by the processor core before a transaction sends the request for the line to the directory. This makes sure that any incoming message regarding that line is handled properly.

All the messages, except for $txAccess(\#, @)$, are core-to-core communication. Following, all the new messages that our approach introduces are described. P_0 is the sender and P_1 is the receiver.

- $reader(@)$: P_0 indicates that @ is in its read-set.
- $writer(@)$: P_0 indicates that @ is in its write-set.
- $rdwr(@)$: P_0 indicates that @ is in its read-set and its write-set.
- $nonTXnal(@)$: P_0 indicates that it is accessing @ in a non-transactional way.
- $tryLater(@)$: P_1 is trying to access @ transactionally but P_0 cannot respond at the moment.
- $txAccess(\#, @)$: This message comes from the directory rather than from another core. It indicates the receiver that the core # is accessing @ transactionally.
- *abort*: Request to abort, sent from P_0 to P_1.
- *abortAck*: P_0, which just received an *abort* message from P_1, acknowledges to abort.
- *abortNack*: P_0, which just received an *abort* message from P_1, does not acknowledge to abort.

Certain state-message combinations are worth explaining since they may not be intuitive. For example, when a transaction is in *Committing* state and gets a $txAccess(\#, @)$ message, a $nonTXnal(@)$ message is sent as a reply (previous writing of the line if it is in the *W @* sub-state). This behaviour is so defined due to the "critical cacheline first" approach, explained in Section 3.

Note that, for clarity, we have somewhat simplified the part of the table related to the fields marked with a ⋆ (in the states *Active R/W/RW @*). The response messages are marked as being a "response" so that they do not get responded again by a receiving processor.

3. EAZYHTM: OPTIMIZATIONS

In this section we introduce a series of optimizations to the basic EazyHTM protocol. We describe how we use a critical-cacheline-first commit policy (Section 3.1), track which lines are being accessed only by readers (Section 3.2), and how we use write-back when committing updates (Section 3.3).

3.1 Critical-Cacheline-First

Regardless of how little validation is performed at commit-time, the duration of the commit-phase is bounded below by the time it takes to write the values speculatively modified by the transaction.

However, EazyHTM escapes this lower-bound by using a critical-cacheline-first transaction commit policy. After all transactions that were racing with the current transaction have acknowledged an abort, we take this moment of time (the moment of receiving last acknowledgement) as the unique point in time when the transaction commits.

Once validation is complete, the transaction starts writing all its speculatively modified lines to the shared memory, in some arbitrary order. If during this phase any other transaction wishes to access some not-yet-committed cacheline from the committing transaction, the committing transaction will get notified from the directory. The commit order is changed, and the critical cacheline gets committed first.

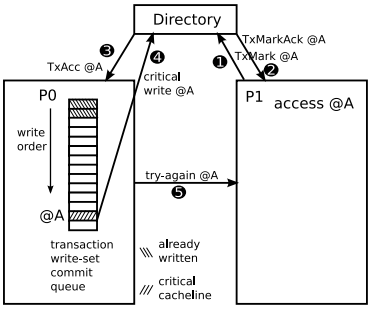

Figure 6: Critical Cacheline First illustration; while P_0 is committing values, P_1 requests a not-yet-written line A; this causes A to be written first, out of normal commit order

After this, a *tryLater*(@) response is sent to the requester. When the requester receives this message, it requests the cacheline again. This time it gets the new value from the shared memory and a *nonTXnal*(@) response from the committing transaction since the line will not be in the commit set anymore. We have effectively saved the stalling time of the requester which would be spent in waiting on the committing transaction to finish. An illustration of this situation is shown on Figure 6 Since T_2 receives a *nonTXnal*(@) message (not *tryLater* or *abort*), it appears that T_1 has finished committing, and so commits seem to be instantaneous. An evaluation of critical cacheline first is given in Section 5.

3.2 Transactionally Dirty Bit

If a transaction reads a line and all the other accessors are readers then messages exchanged between them will be informing one another about their reader-reader status. No modifications will be done neither to the racers list nor to the killers list. Therefore, these messages can safely be avoided.

In order to eliminate these messages, an extra bit per cache line is maintained in the directory. This "Transactionally Dirty" (TD) bit represents whether the cache line is in the write-set of at least one active transaction or not. To distinguish between transactional reads and writes, the $txMark(P, A)$ message to the directory has to be split in two messages: $txMarkRead(P, A)$ and $txMarkWrite(P, A)$. The directory handles these messages as follows.

- $txMarkRead(P, A)$: (1) if the TD bit is set, a message $txAccess(P, A)$ is sent to every accessor, as explained in Section 2. (2) If the TD is zero, no messages are sent to the other accessors, because they are all readers.
- $txMarkWrite(P, A)$: the directory sets the TD bit, and a $txAccess(P, A)$ is sent to every accessor (if any).

The TD bit is cleared on a non-speculative write to the cache line; i.e. when either (1) a transaction commits and thus writes all speculative values to the shared memory, or (2) non-transactional code writes to the line. Note that although this modifies the directory structure, the protocol is not changed in an extensive way. An evaluation of the transactionally dirty bit is given in Section 5.

3.3 Write-Back Commit

Following other lazy version management HTM proposals [5, 6, 11, 12], EazyHTM also implements the write-back commit optimization. Lazy version management HTMs have to publish (in some way) their speculatively modified lines when transaction commits. What can be done as an optimization is to publish only the addresses, and to leave the updated cacheline contents in private caches. This is called write-back commit.

EazyHTM implements write-back commit in the following way. During the transaction execution lines are augmented with speculative read/write status bits. When the transaction comes to a commit, it aborts all racing transactions, receives confirmation of their aborts and then asynchronously (without waiting for confirmation of every message before sending the next one) sends the addresses of all speculatively modified lines to the directory. The directory marks all these lines as exclusive to the processor core. At the same time, in private caches, the line is marked as non-speculatively modified.

A write-back of the cacheline contents is performed when either: (1) another core requests the line later in time, or (2) a line has to be modified speculatively again, by another transaction in the same processor core. In this case, the cacheline access mode is reduced from exclusive or modified to shared, and the execution continues.

4. MICROARCHITECTURAL CHANGES

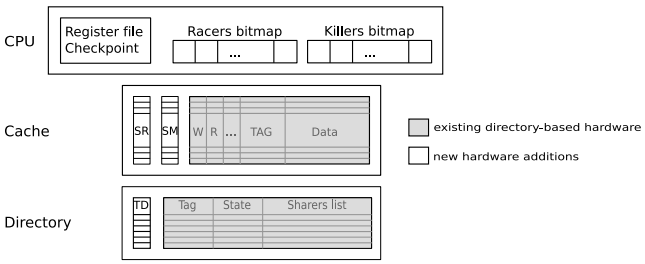

Figure 7: EazyHTM hardware modifications overview

This section introduces the hardware changes which are required for EazyHTM. Described hardware changes support both the basic EazyHTM protocol and the optimizations described in the Section 3. A graphical representation of those changes is shown in Figure 7 where:

Register file checkpoint: keeps a snapshot of the register file. The snapshot is taken at the beginning of the transaction. It is used to restart the transaction's execution in case it aborts.

Racers-list: stores a list of all transactions that have to terminate execution before this one can commit. It is implemented as a simple bit vector (bitmap), with one bit per core. Detailed explanation of Racers-list is given in Section 2.2.

Killers-list: stores a list of all transactions that are allowed to abort the transaction executing on this core. It is also implemented as a bitmap, with one bit per core. Detailed explanation of its functionality is given in Section 2.2.

Cache support: EazyHTM protocol requires tracking transactional accesses to lines from private cache. Thus we extend the private caches with two extra bits: a speculatively-read (SR) bit indicates that the associated cacheline has been read by the currently running transaction, and the speculatively-modified (SM) bit indicates that the cacheline has been modified by the current transaction. Multiple levels of private caches are possible, provided that they all track this information.

Directory Support: As commented in Section 3 we add a transactionally modified (TD) bit per directory entry. This bit marks if a cacheline has been speculatively modified by any transaction since its last non-speculative modification. Note that this modification is independent of the number of processors (i.e. if the system had more cores, the TD would still be one bit per line).

5. EVALUATION

In this section we evaluate the performance of EazyHTM using the STAMP benchmark suite [5]. We describe our simulation environment in Section 5.1. Then, in Section 5.2, we evaluate EazyHTM in comparison with a Scalable-TCC-like HTM [7], and in comparison with an "ideal" HTM.

5.1 Simulation environment

To evaluate the proposal, we have tried to make a fair comparison with other state-of-the-art lazy HTM proposals. We have decided to compare EazyHTM with another lazy (instead of eager) conflict management system, since there is a general agreement on their better performance when compared with eager conflict management systems [4,15]. Among current HTM proposals, we have chosen to compare with Scalable-TCC (STCC) [7] which works on Distributed Shared Memory (DSM) based memory systems and supports partly parallel transaction commits. We used an Alpha 21264 Full-System simulator, M5 [2]. This simulator models a bus-based multiprocessor. We changed its memory hierarchy to model a directory and core-to-core interconnection network (ICN). The configuration of the baseline EazyHTM and the STCC-like simulator used in our evaluation is shown on the Figure 8.

Processor(s)	1-32 cores, 2 GHz, sequential execution
L1 data cache	writeback, private per core, MSI, 32 KB, 4-way, 64B line, 2 cycles hit
L2 cache	writeback, private per core, MESI, 512 KB, 8-way, 64B line, 10 cycles hit
Main memory	MESI based directory, 100 cycles per access
ICN	2D Mesh, 10 cycles per hop

Figure 8: Baseline EazyHTM and STCC-like Simulator Configuration

We have tried to make the base simulator configuration as similar as possible to the one found in the Scalable-TCC proposal. Each processor has private L1 and L2 caches, and with these large private L2 caches, we almost did not have any transaction aborts due to cacheline evictions. This result matches the one presented in the Scalable-TCC paper [7]. If necessity arises, EazyHTM could be extended to handle transactions larger than the private cache size by implementing virtualization mechanisms like UTM [1], VTM [14] and HyTM [10].

In both implementations, the memory and the memory directory is placed after two levels of caches. One difference is that STCC-like HTM has DSM memory modules private per core. When processor needs to access an address that belongs to a memory module of another processor, it sends an inter-node request over the ICN. In EazyHTM there is no explicit requirement for DSM memory distribution. The memory is equally accessible from all processors and all memory addresses have the same access latency. We have also made a simulator configuration with the global (entire) memory directory, but this has been done in the evaluation just for configuration fairness. EazyHTM can work just the same with the shared-cache directory. For instance in this configuration it would be a directory in shared L3 cache.

All instructions have 1 cycle latency except loads, stores and similar instructions which access memory. The directory protocol we implemented for EazyHTM evaluation is MESI-based.

The topology of our core-to-core ICN is 2D mesh. This topology has technologically low cost, complexity and power consumption with modest performance [9]. More advanced interconnection topologies such as 2D torus or 3D torus/mesh will likely be faster and reduce the average latency and hop count between cores. The actual number of hops between any two cores on the die is determined by the ICN. An expected core-to-core ICN latency with today's technology would likely be 2-3 cycles per hop, plus 2-3 cycles per router. For having a consistent simulator configuration and for making a fair comparison with the Scalable-TCC HTM, we have used 10 cycles per hop for both EazyHTM and STCC-like implementations in all cross-comparisons.

5.2 Results

We compared EazyHTM with our Scalable-TCC implementation using STAMP benchmark suite and 9 different benchmark configurations: Labyrinth, Vacation-Low, Yada, Intruder, SSCA2, KMeans-Hi, KMeans-Low, Vacation-Hi, and Genome. "Hi" and "Low" workloads provide different conflict rates [5].

Since we are only interested in the time spent in the parallel section, all the results pertain to this section only. The time spent in transactions differs significantly from benchmark to benchmark. We show it in the first column in the Table 2, for all evaluated applications.

As can be seen in Figure 9, SSCA2 and Genome spend significant time in barrier-based thread synchronization (depicted as *sleeping*). Threads do spend some time sleeping in all of the benchmark configurations, but it is only significant in SSCA2 and Genome. As can be seen in this breakdown, this use of barriers limits the scalability of these programs.

On the same figure, beside sleeping, the execution time is split into *Useful*, *Stall*, *Commit*, and *Wasted*. We define "running time" as 1 cycle per instruction plus number of memory accesses per instruction multiplied by L1 hit latency. For committed transactions we break the execu-

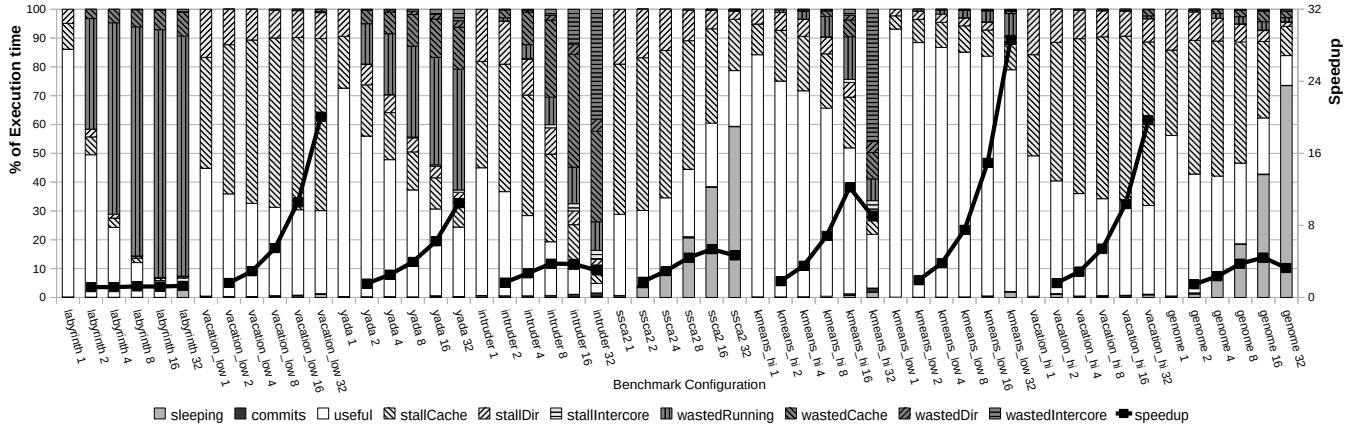

Figure 9: EazyHTM speedup (bold line) and execution time breakdown with 10 cycles per hop ICN

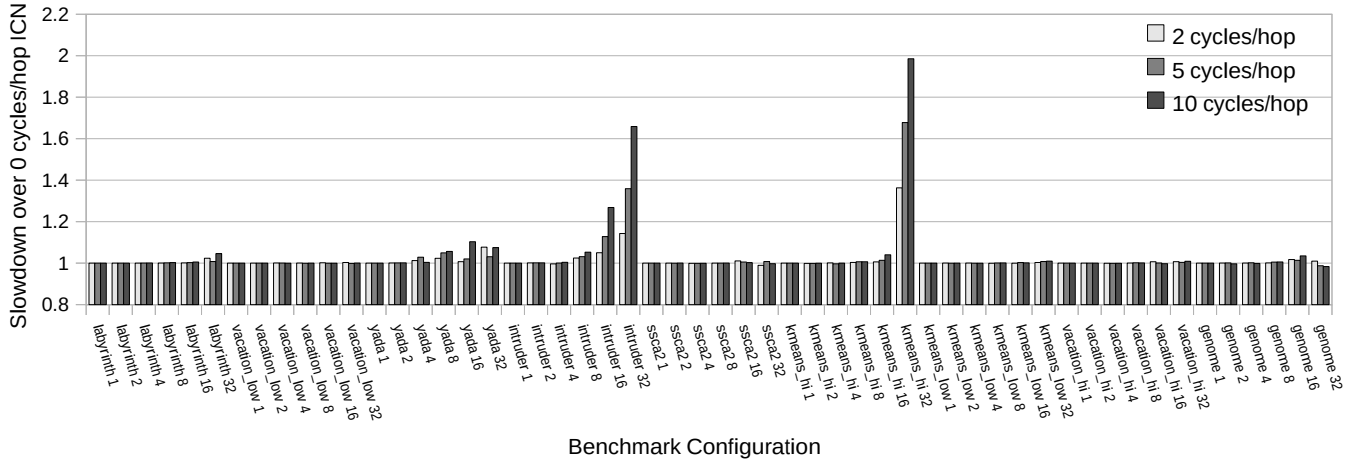

Figure 10: Sensitivity analysis to the core-to-core ICN latency (cycles per hop). Smaller is better.

tion time into: "Useful" time, that is the same as running time. "Stall" time is the time that all committed instructions had to wait, either on cache misses (stallCache), directory (stallDir) or intercore messages (stallIntercore). Stall time is marked in the breakdown with one color, but different causes of stalling have different hatchings. "Commit" time is the time spent in transaction commits. "Wasted" time is all the work that aborted transactions made, and is split into running time (wastedRunning), cache misses (wastedCache), directory accesses (wastedDir) and intercore messages (wastedIntercore). In both useful and wasted time, the access is classified as a (i) cache access, when the line is not present in the cache at the moment of request; (ii) directory access, when the access mode has to be changed – e.g. shared to exclusive.

We can see on the figure that Labyrinth has a practically flat speedup curve. After inspecting the source code and further examining the STAMP paper [5] we have found that for Labyrinth to scale, an "early release" [16,18] of cachelines has to be supported by HTM. Early release means that a cacheline is explicitly removed from the transaction during its execution, i.e. before commit. In Labyrinth, in the main loop of the application, every transaction reads an entire dataset (and thus adds it into the read set). On commit every transaction writes to some part of the dataset. In a simple HTM implementation, this part of execution is guaranteed to create a conflict on every commit. Since neither of our HTMs support early release, we have a fully conflicting execution.

On Figure 10 we show the sensitivity analysis of EazyHTM to different core-to-core ICN latencies. We start from an optimistic 2 cycles per hop and increase the latency to the pessimistic 10 cycles per hop.

Having a slower ICN seriously affects only the execution time of KMeans-hi with 32 threads, and Intruder with 16 and 32 threads. All other applications are mostly insensitive to the ICN link speed. One interesting case is Genome with 32 threads, where having slower ICN actually decreases the total execution time by 2%. The root of this is again in barriers. Having slower ICN makes threads spend less time sleeping in barriers: 101%, 97% and 96% of the time spent in barriers with 0 cycles per hop ICN, for 2, 5 and 10 cycles per hop, respectively.

To assess the overheads of EazyHTM, we also implemented an *Ideal MESI-based Lazy HTM* that performs transaction validation *instantaneously* (in zero clock cycles), without any

Application	Proc	%TX	%ABO	%CLF	%TD	%WBC
Labyrinth	1	99.9	0.0	0.0	0.0	4.4
	2	99.9	22.5	0.5	33.6	1.9
	4	99.7	46.5	0.5	67.5	1.7
	8	100.0	68.3	1.0	85.6	1.7
	16	100.0	82.6	0.4	94.2	0.3
	32	96.1	90.4	6.3	96.1	2.7
Vacation-Low	1	85.9	0.0	0.0	0.0	12.2
	2	86.0	0.0	0.0	27.5	8.3
	4	86.8	0.1	0.0	60.2	7.6
	8	86.0	0.3	0.0	80.3	7.0
	16	83.7	0.7	0.0	90.0	6.8
	32	80.4	1.5	0.0	94.9	6.6
Yada	1	100.0	0.0	0.0	0.0	8.9
	2	100.0	5.2	0.0	30.0	11.3
	4	100.0	9.2	0.1	57.7	5.7
	8	100.0	15.5	0.0	75.1	4.5
	16	99.6	23.7	0.3	84.6	3.7
	32	100.0	35.2	0.7	90.4	8.4
Intruder	1	39.1	0.0	0.0	0.0	17.3
	2	40.6	5.6	0.0	22.9	13.7
	4	42.5	20.1	0.2	53.6	9.7
	8	51.5	43.1	1.6	77.0	5.1
	16	69.6	70.3	12.0	83.0	0.4
	32	86.8	84.7	33.9	79.9	0.1
SSCA2	1	20.0	0.0	0.0	0.0	15.7
	2	18.7	0.0	0.0	24.6	10.8
	4	16.5	0.2	0.0	49.1	10.2
	8	12.9	0.4	0.0	69.2	8.4
	16	7.8	0.6	0.0	82.3	5.9
	32	3.2	1.2	0.0	90.3	1.6

Application	Proc	%TX	%ABO	%CLF	%TD	%WBC
KMeans-Hi	1	9.5	0.0	0.0	0.0	4.8
	2	9.5	1.0	0.0	3.2	4.2
	4	9.5	3.2	0.0	8.7	4.4
	8	9.5	8.9	0.1	22.4	4.0
	16	9.9	26.4	1.1	35.6	1.9
	32	13.0	72.9	17.6	36.8	1.5
KMeans-Low	1	3.8	0.0	0.0	0.0	2.1
	2	3.8	0.6	0.0	2.5	2.0
	4	3.8	2.3	0.0	6.6	2.1
	8	3.8	3.6	0.0	8.6	1.9
	16	3.8	5.3	0.0	13.1	1.9
	32	3.8	11.9	0.8	23.4	1.7
Vacation-Hi	1	86.0	0.0	0.0	0.0	10.7
	2	85.4	0.0	0.0	28.1	7.2
	4	84.8	0.5	0.0	60.8	6.8
	8	83.6	0.7	0.0	80.7	6.5
	16	84.3	0.9	0.0	90.5	6.7
	32	83.5	4.1	0.1	95.4	6.1
Genome	1	97.9	0.0	0.0	0.0	13.1
	2	92.7	0.6	0.0	53.8	8.0
	4	78.9	1.6	0.0	79.3	7.5
	8	57.6	3.4	0.0	90.1	6.7
	16	27.6	7.7	0.1	93.9	1.2
	32	9.2	13.5	0.5	95.8	3.2
AVERAGE	1	60.2	0.00	0.0	0.0	9.9
	2	59.6	3.97	0.1	25.1	7.5
	4	58.1	9.30	0.1	49.3	6.2
	8	56.1	16.0	0.3	65.5	5.1
	16	54.0	24.2	1.6	74.1	3.2
	32	52.9	35.0	6.6	78.1	3.6

Table 2: EazyHTM Execution Statistics.

Legend: **%TX** — Percentage of parallel section time spent inside transactions; **%ABO** — Percentage of aborts (abort rate), calculated as aborts/(aborts+commits); **%CLF** — Number of critical cacheline first invocations divided by the number of commits; **%TD** — Percentage of transactional directory messages saved with the TD bit in directory, calculated as msgNotSent/(msgNotSent+msgSent) **%WBC** — Execution time reduction using write-back commit.

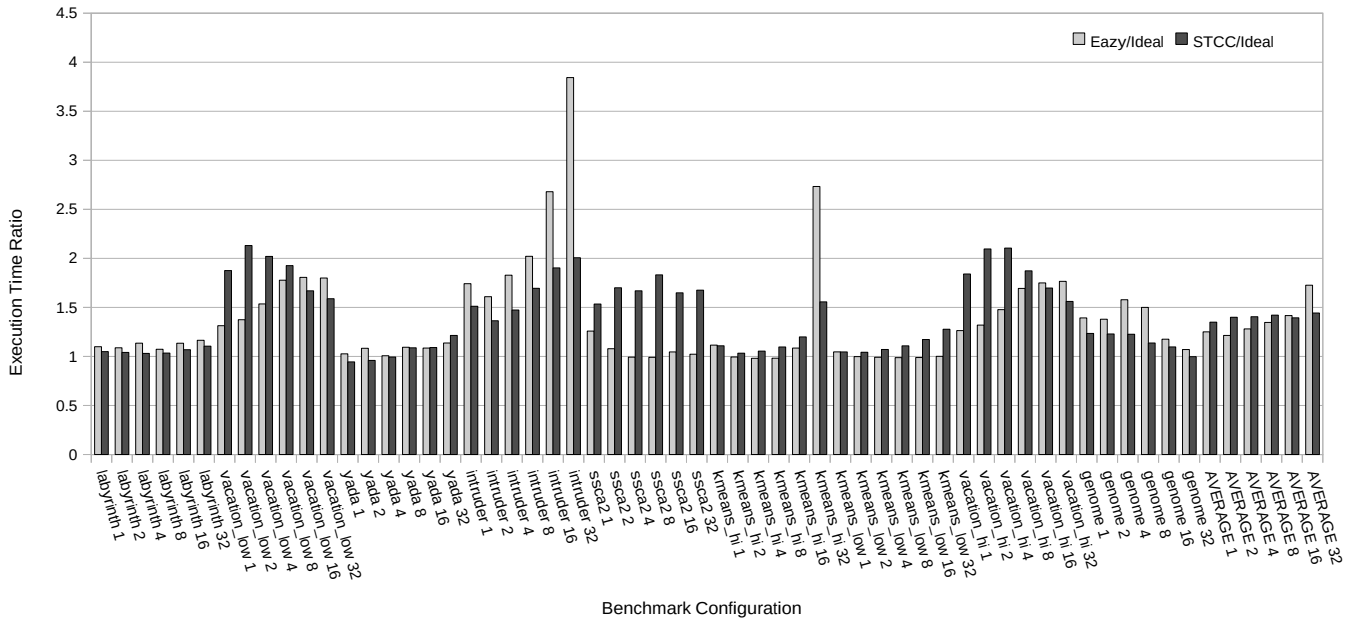

Figure 11: Execution time comparison, EazyHTM and Scalable-TCC-like HTM. Smaller is better.

extra directory or core-to-core messages. After performing the validation, transaction enters a write-back commit routine (see Section 3.3) and upgrades the access mode for all speculatively modified cachelines to "modified". At the same time it cleans all speculative bits from private caches. This entire commit process takes as much as is needed to send all get_exclusive messages to the directory. We have put a fixed cost of 3 cycles per one speculatively modified cacheline. While having this idealized lazy HTM in hardware is almost impossible, it serves as a good upper bound on

the best-case lazy HTM performance and directly evaluates *all* extra overheads of EazyHTM. The comparison of the execution times between EazyHTM, STCC-like HTM, and idealized lazy HTM is given in Figure 11.

EazyHTM shows a performance regression over idealized lazy HTM mainly with one application from the STAMP suite — Intruder. Intruder has a very high abort rate. With 32 cores more than 85% of all started transactions get aborted after performing some work (Table 2). This translates to the 84% of entire execution time being wasted, and out of this 44% represents the time spent in the core-to-core communication, 4% spent in sending and receiving directory messages, 39% in cache requests, and 13% in normal execution. It is also interesting that in one benchmark configuration, Yada with 1 processor, Scalable-TCC has 6% faster execution time from both EazyHTM and the Ideal MESI-based Lazy HTM. This is most likely the consequence of its different directory protocol, which shows some advantages in this specific memory access pattern. However, in some other configurations, e.g. in Vacation (both low and hi) and SSCA2, EazyHTM benefits better from their memory access sequence. Overall, EazyHTM shows a performance improvement of 7% over STCC-like HTM.

Table 2 shows an evaluation of the critical-cacheline-first commit optimization. We can see that even with the write-back commit optimization, some applications have many invocations of critical-cacheline-first. For example, in Intruder, the number of invocations is 1/3 of the number of commits.

In the same Table 2 we show the execution statistics for the transactionally dirty (TD) bit optimization in the directory. Here we have very encouraging results. Given that STAMP benchmarks are optimized to have a high number of cachelines with shared readers only, this optimization was able to drastically reduce the number of additional messages in the system. Of all the messages that would be sent with the original EazyHTM protocol running on 32 cores, 78% are filtered out on average. With some configurations, like Genome-32, Vacation-Hi-32, and Labyrinth-32, this number is over 95%.

6. RELATED WORK

Log-TM [13] describes a taxonomy of TM systems based on version management and conflict detection, placing Log-TM and Unbounded TM [1] into the eager-eager category. Large TM [1] and Virtual TM [14] are classified as eager-lazy and TCC [8] into lazy-lazy.

TCC [8] was the first hardware transactional memory with lazy conflict detection and lazy conflict resolution. However, it incurs from two bottlenecks: first, it utilizes a single common bus between processors; and second, all commits are serialized with a commit token which has to be acquired by a transaction at commit time.

Scalable TCC [7] enhances the original TCC proposal. It uses lazy conflict detection and lazy version management and supports partially concurrent transaction commits. It also introduces an alternative to common MESI/MOESI cache coherence protocols.

Scalable TCC assumes that execution is always transactional, and non-transactional code is converted to implicit transactions. This adds pressure to the importance of being able to perform commits in parallel. However, Scalable TCC is limited in its scalability by the number of directories, and with a small number of directories, commits may be often serialized. Currently, typical chip-multiprocessor implementations have few directories.

Unlike Scalable TCC, EazyHTM is designed to work as an extension to a traditional directory protocol, and to allow truly-parallel commits (rather than being limited by the number of directories present in the system). Lastly, unlike Scalable TCC, EazyHTM has explicit transactional and non-transactional modes that do not require implicit transactions.

Shriraman *et al.* introduce a *mixed* conflict resolution policy [15]. This handles conflicts eagerly or lazily depending on their type. Write-write conflicts are resolved eagerly to save wasted work and read-write conflicts are dealt with lazily, to exploit concurrency. The main difference between [15] and this work is that they both detect *and* resolve conflicts eagerly on write-write and lazily on read-write. On the other hand, we detect both kinds of conflicts eagerly and resolve all of them lazily.

7. CONCLUSIONS AND FUTURE WORK

In this paper we demonstrate how making small hardware modifications to already existing directory based, cache coherent chip multiprocessors allows us to implement a pure-hardware transactional memory system that detects conflicts eagerly and resolves them lazily.

We have shown that it is possible to have a good trade-off between hardware complexity, the performance, and the capabilities of the Hardware Transactional Memory systems. Several optimizations have been applied to the initial EazyHTM design, and we have obtained a significant reduction in the total number of conflict detection messages by ignoring those for read-only cachelines.

Performance evaluation was carried out using state-of-the-art TM benchmarks. EazyHTM gets an average of 7% performance improvement over current state-of-the-art HTM — Scalable-TCC. At the same time, the EazyHTM protocol provides a complete and exact snapshot of all the conflicts of a transaction during its lifetime. Having this snapshot presents a wealth of useful information which could be leveraged for further research into transaction prioritization, performance optimizations and power management.

8. REFERENCES

[1] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proc. 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, February 2005.

[2] Nathan Binkert, Ronald Dreslinski, Lisa Hsu, Kevin Lim, Ali Saidi, and Steven Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[3] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), November 2006.

[4] Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th*

Annual International Symposium on Computer Architecture, June 2007.

[5] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[6] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[7] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *HPCA*, pages 97–108, 2007.

[8] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, June 2004.

[9] D.N. Jayasimha, Bilal Zafar, and Yatin Hoskote. On-die Interconnection Networks: Why They are Different and How to Compare Them. In *Technical Report at http://blogs.intel.com/research/terascale/ODI_why-different.pdf, Microprocessor Technology Lab, Corporate Technology Group, Intel Corp.*, 2006.

[10] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, March 2006.

[11] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 53–65, June 2006.

[12] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Characterization of TCC on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2005.

[13] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *HPCA '06: Proc. 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, February 2006.

[14] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, June 2005.

[15] Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In *ICS '09: Proc. 23rd international conference on Supercomputing*, pages 136–146, June 2009. Also available as TR 939, Department of Computer Science, University of Rochester, September 2008.

[16] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *Proc. Workshop on Transactional Workloads*, June 2006.

[17] James Smith. A study of branch prediction strategies. In *International Symposium on Computer Architecture*, pages 202–215, 1998.

[18] Nehir Sonmez, Cristian Perfumo, Srdjan Stipic, Adrian Cristal, Osman Unsal, and Mateo Valero. unreadTVar: Extending haskell software transactional memory for performance. In *Proc. of Eighth Symposium on Trends in Functional Programming (TFP 2007)*, 2007.

[19] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09: Proc. 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150, February 2009.