QuickTM: A Hardware Solution to a High Performance Unbounded Transactional Memory

Sutirtha Sanyal*, Sourav Roy†
^{*,1}Barcelona Supercomputing Center, Barcelona, Spain; †Freescale Semiconductor India Pvt. Ltd.
*ssanyal77@gmail.com; †sourav.roy@freescale.com

Abstract

Transactional Memory (TM) is an emerging technology which simplifies the concurrency control in a parallel program. In this paper we propose QuickTM, a new hardware transactional memory (HTM) architecture. It incorporates three features to address known bottlenecks in the existing HTM architectures. First, we propose hardware-only dynamic detection of true-shared variables. Our result shows that true-shared variables account for only about 20% in the commitset of any transaction. Rest can be completely disregarded from the commit phase. This shortens every commit phase drastically resulting in a significant overall speed-up. Second, we keep both speculative and the last committed versions local to each processor. This benefits when a transaction is repeated in a loop. The processor request gets satisfied from the L1 data cache(L1D) itself. Furthermore, since both the versions are locally maintained, the commit action involves only broadcast of addresses. Third, we have proposed a mechanism to address overflow in transactions. In our proposal, each processor continues to run transactions even if one processor has overflown its L1D. Our technique eliminates the stall of a thread even if it conflicts with the overflown transaction. Overflown transaction commits in-place and periodically broadcasts its writeset addresses, termed "partial commit". This gradually reduces conflicts and allows other threads to progress towards commit. Moreover, the technique does not require any additional hardware at any memory hierarchy level beyond L1.

QuickTM outperforms the state-of-the-art scalable HTM architecture, Scalable-TCC, on average by 20% in the latest TM benchmark suite STAMP. It outperforms the original TCC proposal with serialized commit by 28% on average. Maximum speed-up achieved in these two cases are 43% and 67% respectively. Our proposal handles transaction overflow gracefully and outperforms the current overflow-aware HTM proposal, OneTM-concurrent by 12% on average.

1. Introduction

Multiple cores with shared memory is swiftly changing from being a novel to a standard feature. Process technologies are at physical limit that no more performance could be extracted out of a single core. Therefore, the current research are targeted to achieve better performance by using parallel processing techniques. Soon, many simple cores within a single die will be the common case. Recently Intel has revealed the prototype architecture consisting of eighty cores [1].

However, writing a correctly synchronized parallel program is a difficult task. Traditional way of guaranteeing correctness relies on mutual exclusion locks. Locks are both non-scalable and highly error-prone. Programmers need to use locks carefully and sparingly only on the essential portion of the code. Otherwise the code may not scale and even execution may be deadlocked.

Transactional Memory[2], [3], [4], [5], [6] is a technology where programmers just need to wrap a portion of the code in what is known as a transaction. It is then the responsibility of the TM framework to guarantee correct execution considering coincident shared memory accesses by multiple threads. TM can be implemented in software (STM), hardware (HTM), or in a hybrid way[6]. Even though software-based approaches[7], [8] support rich and flexible transactional semantics, they have relatively high performance overhead compared with a hardware solution. Thus in our study[1] we focus on hardware-based transactional memory[2], [3], [9]. There are several choices in implementing conflict detection and version management policy in transactional memory. One choice is to defer both the conflict detection among multiple threads and versioning till the commit time. This is known as lazy-lazy(Lazy conflict detection-Lazy version management) or LL type of TM system. This gives highly concurrent execution. Another choice relies on committing data immediately (or versioning eagerly) to favor the common case behavior of transactions. However, for the latter case, the conflict also has to be detected eagerly. Otherwise, the isolation property of the TM system will be violated. This type of system is known as eager-eager (or EE) type of TM system.

In this paper we propose a new hardware transactional memory(HTM) architecture named QuickTM. Our policy

[1] *The significant part of this work was carried out while first author was in Barcelona Supercomputing Center till May 31, 2010.*

is lazy-lazy or LL. In our architecture we have proposed several features to boost the performance level of a conventional HTM system. We have proposed in our architecture a hardware mechanism to detect true-shared memory accesses made within a transaction. Our detection mechanism is dynamic and identifies the shared nature of variables online. Our results show that on average only 20% variables modified within a transaction are true-shared and need to be considered during the commit phase. Rest 80% thread-local variables can be omitted. This drastically reduces the commit overhead for a HTM system. Typically "commit" is an expensive operation where committed data are pushed into a non-speculative level of memory[2] or addresses need to be broadcasted[3]. This phase is the bottleneck in the highly concurrent HTM system where conflict detection and versioning are deferred till the end of a transaction. Therefore, disregarding all thread-local variables from the commit has a directly proportional impact on the length of each commit phase. This great reduction in each commit length in turn gives an overall program speed-up of 20% in the state-of-the-art TM benchmark suite STAMP[10] over the state-of-the-art HTM, Scalable-TCC[3].

Supporting overflown transactions in a true hardware manner is essential for acceptability of HTM in commodity processors. Current state-of-the-art, One-TM[11], stops all other conflicting processors when transaction in one processor exceeds the L1 data cache(L1D) capacity. It executes the overflown transaction in an irrevocable mode guaranteeing commit. Another flavor of One-TM, One-TM concurrent, only stops processors which have conflicts with the overflown transaction. Compared to this, in our proposal, we do not stop execution in any processor. Moreover, our overflow handling scheme is light-weight requiring minor modifications only in the L1D.

This paper makes following contributions:

- We propose a novel HTM architecture named QuickTM. In our proposal several performance enhancements are incorporated to speed-up the current state-of-the-art in HTM, Scalable-TCC[3].

  We implement a dynamic detection scheme of true-shared variables. Our scheme involves only modifications in the cache coherence scheme. During the commit phase, we consider only true-shared variables modified in the transaction. Our simulation results based on the latest TM benchmark STAMP[10] show that on average only 20% (in some cases 1%) of the variables in the commitset are truly shared among threads (Figure 8). Rest 80% variables are thread-local and cannot generate any conflict with other threads.
- In our architecture we store both committed version and the speculative version in the same set of the cache, but in different ways. This keeps both versions local to a processor. If a transaction is repeated in a loop, this speed-up the access to same variable as the previously committed version will still be found in the L1D. Next, by keeping both versions within L1 boundary, on commit, only addresses need to be sent out. Thus it achieves address-only commit where data bus width does not affect the commit time. Moreover, we consider addresses of only true-shared variables. Therefore, the commit action reduces to broadcast of very few true-shared addresses which makes it a very "fast" operation. The above two features put together outperforms the Scalable-TCC by 20% on average (with a maximum speed-up of 43%) in the STAMP benchmark.
- In QuickTM we propose a novel mechanism to handle transaction overflow. Our scheme does not halt any processor and does not require additional bits at any memory hierarchy level beyond L1. Once a transaction overflows, it gets a guarantee to commit. From that point, it updates in-place. Also, post overflow point, it periodically broadcasts addresses of updated true-shared variables for other threads to detect conflict. This is termed "partial commit". Other non-overflown threads can continue the execution. However, they may get aborted due to conflict when a partial commit occurs. On the other hand, if there is no further modifications to the same line by the overflown transaction, then no conflict is generated and the other thread continues without abort. Thus the stall at the conflict point is completely eliminated. In OneTM-concurrent[11], transactions with conflict get stalled and restart execution from beginning after the overflown transaction commits. Contrary to this, in QuickTM transactions tend to commit immediately after overflown transaction ends. We report an average speed-up of 12% over OneTM-concurrent, in the STAMP benchmark where overflow situation is created either by using large input data set or by reducing the cache size. In summary, QuickTM with its integrated architecture, is an ideal candidate for adoption into the modern commodity processors.

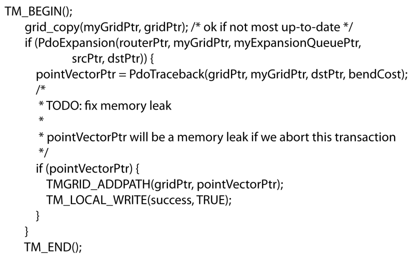
2. Background

2.1. The TCC Architecture

Our base line system is modeled after the TCC[2], [3]. The TCC (Transactional Coherence and Consistency) is a technique to support the Transactional Memory in hardware. It implements the Lazy-Lazy form of the HTM. In this, the conflict is detected at the end of the transaction when it is about to commit. Till that time, all its changes are marked as speculative. Original TCC implementation augments L1 data cache line with two additional transactional bits, "Read" and "Write". A transactional read sets the R bit and write sets the W bit. At the time of commit, only one processor is permitted to flush its commitset into the bus. This causes non-scalability in this bus-based model as it disallows parallel commit.

The Scalable TCC protocol[3] relies on the distributed shared memory structure. In the system, multiple directories are present which maps different segments of the physical memory. Processors load data from different directories and then update them speculatively in their own private L1D. A centralized token vendor generates a token id when the processor reaches the commit stage. This token id (TID) acts as a timestamp for the transaction commit. The existence of multiple directories allows parallel commit from several processors. However, when two processors attempt to commit in the same directory, they get serialized based on their timestamp value. After a successful validation phase where no invalidation message is received, a processor can start committing. All speculative stores are committed in the directories and sharer of those cache-lines are sent invalidations.

2.2. Thread-local and True-shared Variables in Transaction



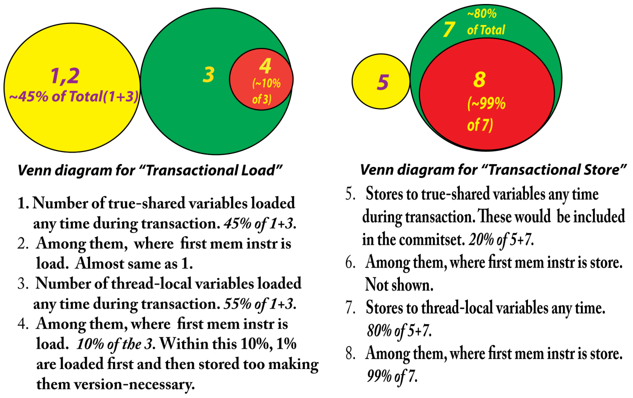
```
TM_BEGIN();
      grid_copy(myGridPtr, gridPtr); /* ok if not most up-to-date */
      if (PdoExpansion(routerPtr, myGridPtr, myExpansionQueuePtr,
                  srcPtr, dstPtr)) {
         pointVectorPtr = PdoTraceback(gridPtr, myGridPtr, dstPtr, bendCost);
         /*
          * TODO: fix memory leak
          *
          * pointVectorPtr will be a memory leak if we abort this transaction
          */
         if (pointVectorPtr) {
            TMGRID_ADDPATH(gridPtr, pointVectorPtr);
            TM_LOCAL_WRITE(success, TRUE);
         }
      }
      TM_END();
```

Figure 1. An example from labyrinth (STAMP) showing thread-local and true-shared variables in transaction

In this section we discuss the distinction between true-shared and thread-local variables within a transaction. As shown in the Figure 1, within the transaction, variables such as "pointVectorPtr", "success" are defined in the thread-local stack. Moreover, note the variable "myGridPtr". It is a pointer to a chunk in process heap. However, contents this pointer referring to are not shared among threads. This portion of heap is thread-local. Therefore both these two categories of variables can be filtered out from the commit packet because they do not generate conflict with other threads. For eager system, their addresses need not be hashed into the bloom-filter[12]. Rest few variables are truly-shared among threads and TM system must employ its versioning and conflict detection mechanism while accessing them.

2.3. Load-Store Pattern to Transactional Variables

In Figure 2 we show typical dissection of true-shared and thread-local variables using two venn diagrams. One analyzes the typical transactional load pattern while the other examines transactional stores.



Venn diagram for "Transactional Load"

1. Number of true-shared variables loaded any time during transaction. *45% of 1+3.*
2. Among them, where first mem instr is load. Almost same as 1.
3. Number of thread-local variables loaded any time during transaction. *55% of 1+3.*
4. Among them, where first mem instr is load. *10% of the 3.* Within this 10%, 1% are loaded first and then stored too making them version-necessary.

Venn diagram for "Transactional Store"

5. Stores to true-shared variables any time during transaction. These would be included in the commitset. *20% of 5+7.*
6. Among them, where first mem instr is store. Not shown.
7. Stores to thread-local variables any time. *80% of 5+7.*
8. Among them, where first mem instr is store. *99% of 7.*

Figure 2. Venn diagram representation of transactional load/stores to true-shared and thread-local variables.

It is to be emphasized that, the number of thread-local variables are not 80% of the total number of transactional variables. We have shown that, 80% of all stores occurring to transactional variables will be directed to thread-local variables. Therefore, the commitset would consist of mostly thread-local variables. Within thread-local variables, in [13] it is shown that, 99% of them would be modified first before being used. Rest $\sim$1% denotes the few version necessary thread-local variables where they were first loaded and then modified. However, in this work we do not include this feature. This is because, our detection scheme identifies true-shared variables based on requests from other processors during the transaction. Therefore until a transaction starts committing, any variable can be identified as shared.

3. Related Work

In [13], [12], authors proposed a solution which detects the local or shared nature of an address belonging to process heap by inspecting a certain bit in its permission bit field or by a stack bound comparison for thread-local portion of the stack. However, the techniques presented in [13] requires modification in the OS and run-time whereas in [12] a modification in the OS, compiler, run-time and programmer supplied hints are required. Compared to this, in QuickTM, we have carried out the detection scheme entirely in hardware eliminating any software dependency.

The basic architecture of keeping both versions of data in processor local cache was first proposed in [5]. However, their technique detects conflicts eagerly which reduces amount of concurrent execution. The other two proposals to maintain atomicity are a) broadcast the data at the commit time to a non-speculative level of memory such as L2[2] or b) keep the data in the L1 but if the processor tries to modify it again in a transaction, then first the committed value needs to be flushed to L2[3], [14]. Compared with latter two proposals, the former one has obvious advantages. Repeated transactions can fast access the same datum from L1D and also after abort all previously committed versions still remain in L1D. However, a slight high associativity[9] in the cache is required to hold both the versions.

UTM[15] supports multiple overflown transactions by spilling data to memory. However, it requires additional storage in memory for keeping transaction logs. OneTM-concurrent[11] proposes to solve the overflow issue by allowing only one overflown transaction to exist at any given time. However, it stalls conflicting threads. In our proposal, we do not stall any thread on conflict. All threads move forward even if they conflict with the overflown transaction, if necessary with an abort.

Sanyal et al. had first shown in [16] how a novel back-off scheme can improve both performance and energy consumption of a processor supporting HTM. The back-off scheme improves performance by eliminating commit-time futile spin where processors spin to get commit permission but are eventually aborted. Later EazyHTM[17] achieved about the same performance, avoiding futile commit spin by accounting for conflicts early during the transaction. However, instead of giving any energy savings, it adds on to processor energy consumption with several hardware additions.

Harris et al. proposed an optimization[18] which avoids repeated undo-logging (or versioning of previous value) in eagerly versioned STM. Though it requires logging at least once. Moreover, it is not useful for highly concurrent STMs which keep redo-log instead of undo-log. This is because redo-log must be updated on every write and consulted on every read. However, the technique proposed in [13] is different and more impacting because it altogether eliminates the versioning necessity by 99% for all thread-local variables.

4. The QuickTM Additions to Architecture

4.1. Cache Architecture

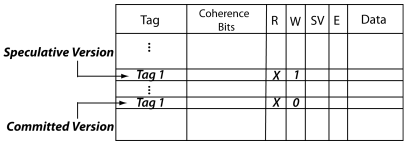


Figure 3. Modifications to the data cache

The Figure 3 shows that, in the QuickTM cache architecture, only 4 control bits are added to the L1 data cache lines.

- R bit denotes if a cache line belongs to the read-set of the transaction.
- W bit denotes if a cache line belongs to the write-set of the transaction.
- SV bit denotes whether the line is detected as shared among multiple transactions.
- E bit, if set means that the line is speculative and corresponding committed line is evicted.

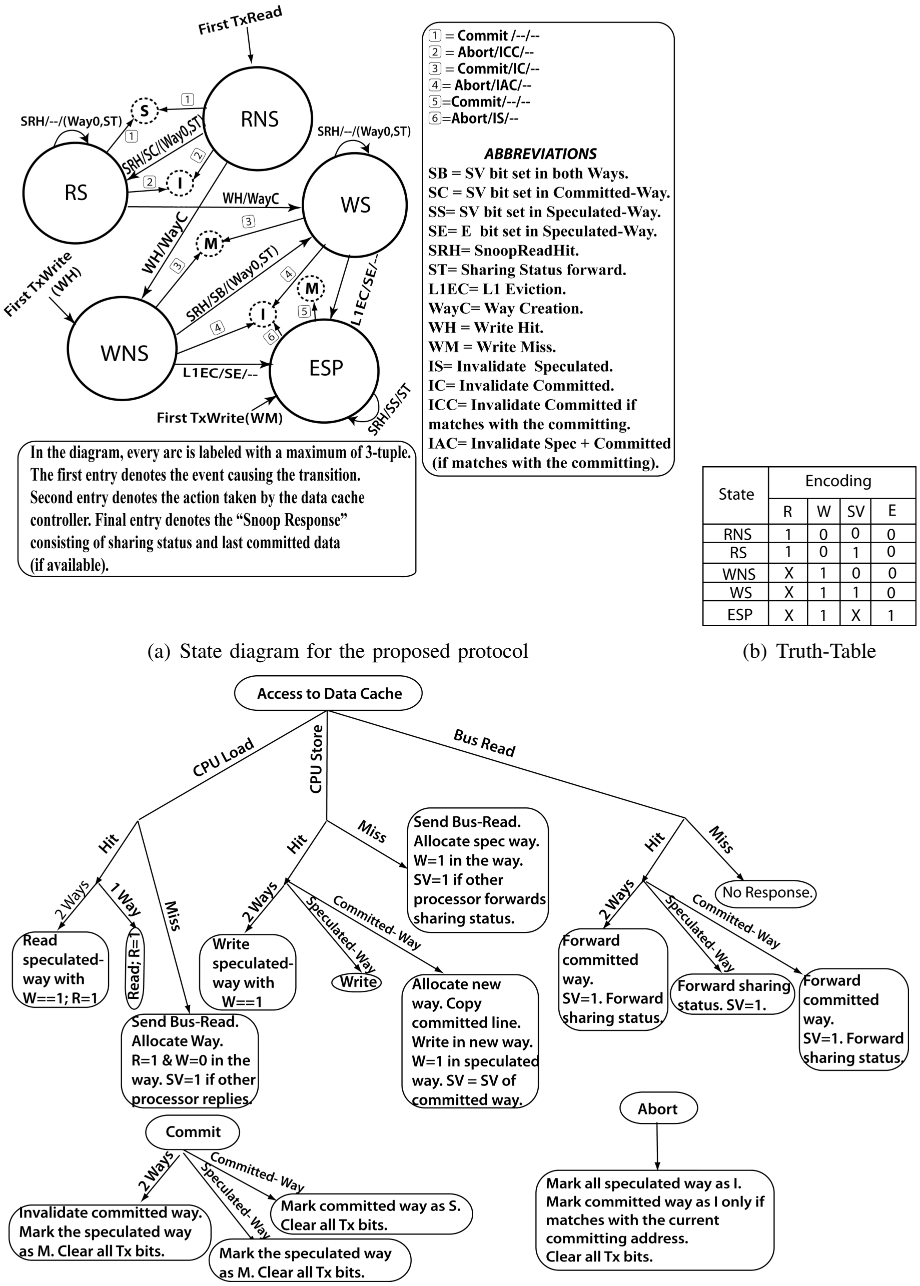
In QuickTM we store two cache lines with same address tag but with different W bits, in two different ways on the same set. The data versioning bit W is provided to the cache lines so that both committed (in Way 0) and speculated (in Way 1) versions can occupy the L1D cache. Note that these cache directory bits ideally should be maintained at a word(or byte)-level of granularity because; often a true-shared and a thread-local variable will occupy the same cache line. Therefore, if the SV bit is maintained at the cache-line granularity then all cache lines could end up being marked as shared even though, in actuality, only few words in the entire line are truly shared. The area overhead of keeping these bits at a finer level of granularity is negligible[13]. The QuickTM protocol described in the next section is independent of the granularity level at which R, W and SV bits are maintained. Therefore, for the sake of brevity, we would assume a cache line level of granularity while describing the protocol.

4.2. QuickTM Protocol Operations

In this section we will describe QuickTM operations and cache-coherence modifications required to implement them. Figure 4(a) shows our proposed modifications on top of standard MESI protocol. The fundamental technique is however orthogonal to the underlying coherence mechanism. Note that, the additional states do not require any extra bits. Necessary transactional bits such as R,W, SV and E are used to encode them as shown in Figure 4(b). Our cache model is write-back and write-allocate. This is the most common mode of cache operation for medium to large sized caches found in high performance processors[19], [20], [21].

We are proposing following states along with the standard MESI states for every cache line:

- RNS (*Read Non-Shared*) = A line stays in this state whenever the W bit corresponding to it is 0 and the R bit is 1.
- RS (*Read Shared*) = Same as RNS but with SV bit set to 1.
- WNS (*Write Non-Shared*) = This state means that, we have both speculative line(Way 1) and the committed line(Way 0) in L1D.
- WS (*Write Shared*)= Same as WNS but with SV bit set to 1.
- ESP (*Evicted with Speculative Present*) = This means the cache line in the L1D is speculative and the corresponding committed copy is absent.



1 = Commit /--/--
2 = Abort/ICC/--
3 = Commit/IC/--
4= Abort/IAC/--
5=Commit/--/--
6=Abort/IS/--

ABBREVIATIONS

SB = SV bit set in both Ways.
SC = SV bit set in Committed-Way.
SS= SV bit set in Speculated-Way.
SE= E bit set in Speculated-Way.
SRH= SnoopReadHit.
ST= Sharing Status forward.
L1EC= L1 Eviction.
WayC= Way Creation.
WH = Write Hit.
WM = Write Miss.
IS= Invalidate Speculated.
IC= Invalidate Committed.
ICC= Invalidate Committed if matches with the committing.
IAC= Invalidate Spec + Committed (if matches with the committing).

In the diagram, every arc is labeled with a maximum of 3-tuple. The first entry denotes the event causing the transition. Second entry denotes the action taken by the data cache controller. Final entry denotes the "Snoop Response" consisting of sharing status and last committed data (if available).

(a) State diagram for the proposed protocol

State	Encoding			
	R	W	SV	E
RNS	1	0	0	0
RS	1	0	1	0
WNS	X	1	0	0
WS	X	1	1	0
ESP	X	1	X	1

(b) Truth-Table

Access to Data Cache
CPU Load
CPU Store
Bus Read
Hit
Miss
2 Ways
1 Way
Read speculated-way with W==1; R=1
Read; R=1
Send Bus-Read. Allocate Way. R=1 & W=0 in the way. SV=1 if other processor replies.
Hit
Miss
Send Bus-Read. Allocate spec way. W=1 in the way. SV=1 if other processor forwards sharing status.
2 Ways
Speculated- Way
Committed- Way
Write speculated-way with W==1
Write
Allocate new way. Copy committed line. Write in new way. W=1 in speculated way. SV = SV of committed way.
Hit
Miss
No Response.
2 Ways
Speculated- Way
Committed- Way
Forward committed way. SV=1. Forward sharing status.
Forward sharing status. SV=1.
Forward committed way. SV=1. Forward sharing status.
Commit
2 Ways
Speculated- Way
Committed- Way
Invalidate committed way. Mark the speculated way as M. Clear all Tx bits.
Mark the speculated way as M. Clear all Tx bits.
Mark committed way as S. Clear all Tx bits.
Abort
Mark all speculated way as I. Mark committed way as I only if matches with the current committing address. Clear all Tx bits.

(c) Decision-Tree

Figure 4. QuickTM protocol description

4.3. Load-Store Accesses

We will first discuss the cache operations for load and store accesses generated by the processor core. When a transaction starts, the default value of the W bit is equal to 0. They are all unused since both R=0 and W=0. All loads until the first store will read from the committed line, setting the R bit in the process. In this state (RNS), it is guaranteed that the data present in the line is still same as the committed copy. Later, some other processor may ask for the data. In that case, it will result in a *Snoop-Read-Hit (SRH)* in the owner. As shown in the Figure 4(a), the owner moves the state of the line to "RS" from "RNS". As a response, it forwards the data it has in its L1D. Also, it needs to forward the *Sharing Status* to the requesting processor after setting the SV bit in its own cache line. This is because, at the commit time, it needs to know that some other processor also asked for the same line (by checking the SV bit).

On the first store to a committed line that results in cache hit, a new way is allocated for the speculated version. This could be achieved without introducing wait states or stalls for the core. For example, during the first cycle when the store address is sent to the cache, the committed line is read. A new way is allocated from the free lines in the set. If a free line is not available, the cache replacement scheme allocates a new way by writing back a non transactional dirty line or a committed line where the corresponding speculative version is present. In a later cycle when the store data are sent to the cache, it is superimposed on the line read previously and written to the newly allocated way. This new allocated line is marked as the speculated version with its W bit set. Its SV and R bits are copied from the committed line. Since both speculative and committed copies reside in the same set, there is also a possibility that during the creation of the new way, the old way containing the committed data gets evicted. In this situation the line state moves to "ESP". This state means that the speculative version is still within L1 boundary but the committed version of the line is not. The "ESP" state is reached on the very first transactional store itself, if that causes a Write-Miss ("WM"). Note that for a write miss, we do not copy the committed line and then allocate a different way for the speculated version. We employ "no-fetch on write" policy[21] on transactional write misses. The only copy L1D contains is considered to be the speculated version with W bit set to 1. A free way is allocated if available and if not a committed way is chosen for replacement.

```
H = | TagHit [0:n-1] //Bitwise OR, n=Number of Ways
I = ^ TagHit [0:n-1] //Bitwise XOR
Double Hit (DH) = H & ~I //True if two Tag hits
Control Vector = (DH==1)?  (TagHit [0:n-1] & W[0:n-1]) : TagHit [0:n-1]
//For cpu read response.
Control Vector =  (TagHit [0:n-1] & ~W[0:n-1])  //For snoop read response.
```

Figure 5. Equations for modified tag-comparison

Note that, L2 should have the least priority as the provider of data in case of read misses. This is because, more often than not, the latest committed version of a line would be found in private L1D of some other processor. Figure 4(c) shows the decision tree for the modified cache controller. The tag comparison path could be augmented minimally to forward proper version of data in same cycle. The boolean equations corresponding to that are shown in Figure 5.

4.4. Transaction Commit and Abort

The final set of operations are the transaction commit and abort. During a transaction commit, a processor broadcasts cache line addresses that have been modified and are also detected as shared. These are speculated lines whose SV and W bits are set. Like TCC, we can employ a Store Address FIFO (SAF) to store the line indices for the commit addresses. Whenever W and SV bits of a line are set, we insert the address in the SAF. At the end of commit, all speculated lines are upgraded to become committed and corresponding previously committed copies are invalidated.

In case of an abort all speculative copies are invalidated. Additionally, an invalidation of the previously committed copy is required if the line address matches with the current ongoing committing address. This means that the line has been detected as shared and in the next iteration, new updated value must be read from other processor's L1D.

4.5. QuickTM Architecture for Distributed Shared Memory

In this section we will discuss the QuickTM architecture for a directory based distributed shared memory(DSM) system. Apart from the obvious scalability advantage of a distributed directory based system over a common bus based system, the former provides multiple non-conflicting transactions to commit in different directories in parallel. For the large scale NUMA systems supporting QuickTM, the processors maintain the same L1D cache architecture as in the bus based system.

The directory structure is quite similar to that employed in the scalable TCC protocol[3]. It tracks the list of processors that have read the cache line - this is called the sharer list. It also tracks the processor that committed to the line the last time-this is called the owner. In case of directory architecture, a bus read request in QuickTM is replaced by a data request to directory. For cache misses, the corresponding directory is enquired for the committed version as shown in the Figure 6. The owner entry in the corresponding directory line gives us the processor whose L1D possesses the committed version. In this figure, we have assumed that the owner has written-back the cache line so a L1D miss would occur in the owner too. In step 3 the owner notifies the directory that a L1D miss has occurred. Now, a L1D miss in the owner may be due to complete absence of that line or presence of only the speculated version. In the latter case, the owner tags the message with a sharing status. Directory further contacts the local main memory in step 4. After getting the response from the memory, data are forwarded to the requester along with the sharing status info. At the end, P0 has been marked in the sharer list but P1 continues to be the owner as it committed the line last.

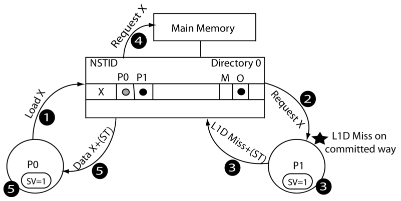
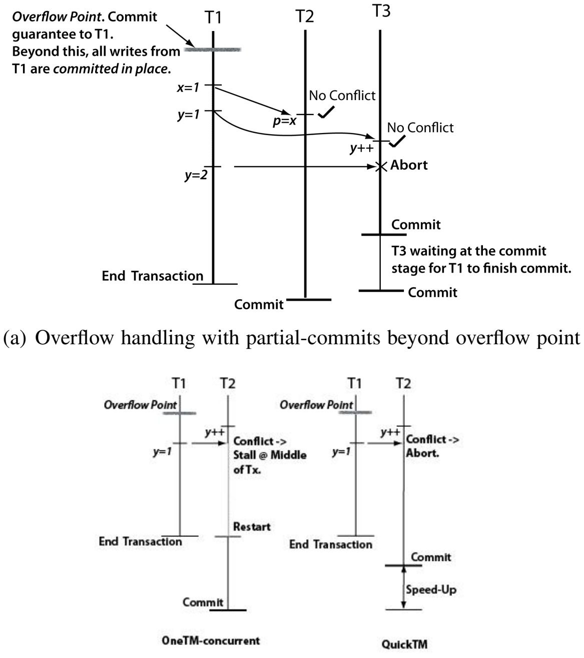


Figure 6. Execution steps in case of cache-miss

The commit and abort handling are similar to the protocol described in the previous section.

5. QuickTM Overflow Management

QuickTM handles overflow with minimal overhead in hardware. Further there is no need of intervention by the OS for lazy cleanup of transactional bits from physical memories[11]. As shown in the Figure 7(a), we consider three threads executing transactions T1, T2, and T3. Now the transaction T1 overflows. As soon as T1 overflows, it is provided a “commit-guarantee”. This means that T1 is guaranteed to be the next transaction to commit. This is achieved in hardware by an overflow signal broadcast from T1 to all processors in the system. Each snooping processor stores this overflow signal and the corresponding CPU id in a dedicated register. They cannot commit unless this overflow bit is cleared by T1. When T1 finally ends the transaction, it broadcasts an “end transaction” message which clears the overflow bit in all CPUs. Till this time other transactions like T3 which has finished execution has to wait for T1 to commit. If other transactions are at the point of overflow and the overflow bit is already set, they are stalled. This guarantees only one overflown transaction exists in the system. After T1 overflows, it does not require versioning anymore since from that point it becomes irrevocable.



(a) Overflow handling with partial-commits beyond overflow point

(b) Speed-up obtained by avoiding stall

Figure 7. Overflow management protocol in QuickTM

At the overflow point, T1 broadcasts the store addresses where SV bit is set. This is safe since T1 now has exclusive commit permission. After that, it periodically broadcasts store addresses, referred as “partial commits”. QuickTM broadcasts after n cache lines are modified, where n may be 4, 8 or higher. Several stores may end up in the same cache line due to spatial locality and hence this would reduce the commit bandwidth further. As shown in the Figure 7(a), access to the variable x by T2 after T1 has modified it, does not create a conflict. Finally in another case, the variable y is modified by T1, then T3 and then again by T1. Here, T3 is aborted. Thus the overflowed transaction T1 aborts other transactions that conflicts with the committed cache lines. Figure 7(b) shows how avoiding stall at the middle of a transaction gives a significant speed-up. On the left, OneTM-concurrent stalls the transaction T2 in the middle. T2 cannot restart until T1 finishes commit. On the right, in QuickTM, we allow T2 to abort and restart. Here, if T1 does not modify y again, then later T2 can read the committed version directly and commit early.

6. Simulation and Results

Simulations are done in a substantially modified version of the M5 full-system simulator[22] adding support for QuickTM, simulating Alpha 21264 architecture. Table I lists the important parameters used in the simulation. For benchmark, we have used all applications from the current state-of-the-art TM benchmark suite, STAMP[10]. Applications were run with the input set specified for simulation run.

Table 1. Parameters used in simulation

Feature	Description
CPU	2-16 single issue in-order Alpha 1GHz
L1D	64KB 64 byte line 8-way 1 cycle access
L2	2MB 64 byte line 16-way 10 cycles access
Interconnect	64B wide data bus, 2-D Mesh , 10 cycles per-hop
Main Memory	1GB, 100 cycles latency
Directory	Full-bit vector sharer list, 10 cycles latency

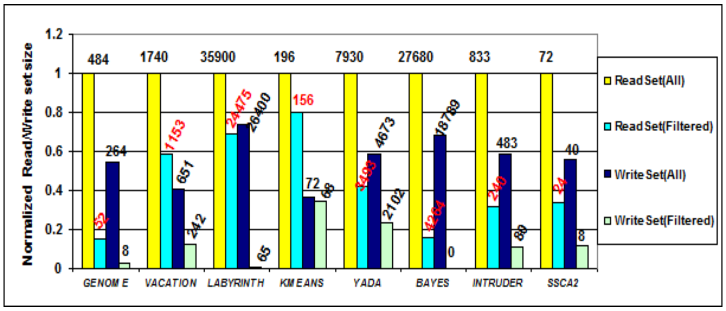
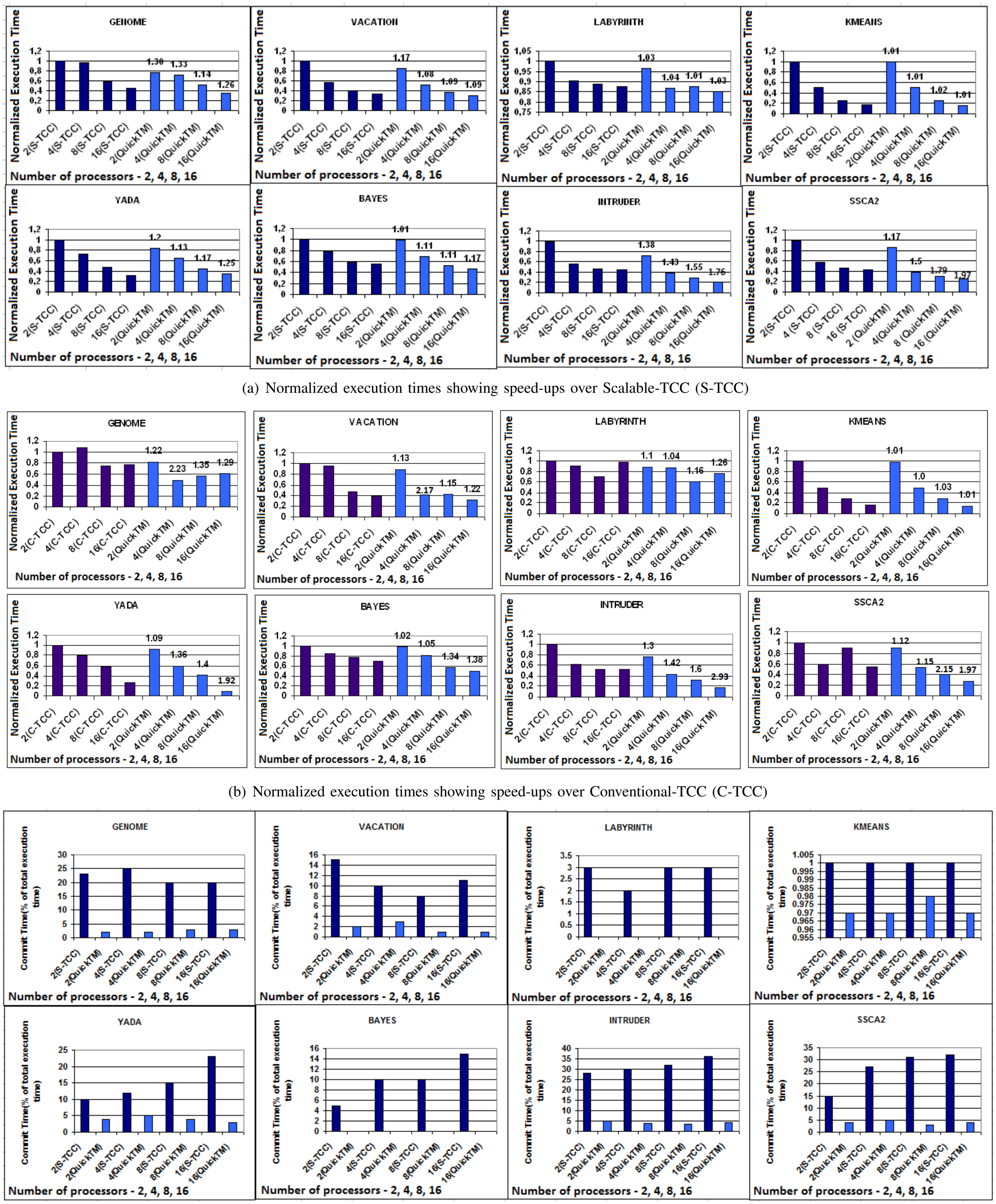


Figure 8. Reduced Readset/Writeset size(bytes) with only true-shared variables



(a) Normalized execution times showing speed-ups over Scalable-TCC (S-TCC)

(b) Normalized execution times showing speed-ups over Conventional-TCC (C-TCC)

(c) Percentage execution time in commit (Scalable-TCC and QuickTM)

Figure 9. Results with QuickTM

In the Figure 8 we show the reduced read/writeset size obtained with QuickTM. First readset and writeset sizes of atomic blocks are averaged over complete execution. Then the atomic block with maximum read and writeset size (in bytes) is reported. We report the size without using true-shared variable detection mechanism of QuickTM and next to it we show the reduction obtained with QuickTM. An average reduction of 80% is observed across all applications. *Note that, for the application bayes, the size of the atomic block with the largest writeset (TMfindBestInsertTask in learner.c) reduces to 0 in QuickTM.*

We compare QuickTM performance with two flavors of lazy-lazy HTM, i.e., current state-of-the-art Scalable-TCC[3] and the original bus-based version[2]. We support parallel commit using directories in QuickTM too (section 4.5) when comparing with Scalable-TCC. Note that, in this work we do not compare with eager-eager HTM such as Log-TM[4] since our policy is fundamentally lazy-lazy. Log-TM may end up having both versions of data in L1D with different tags even though the protocol does not explicitly enforce that. These types of systems however can show deadlock and several performance pathologies[23]. The primary source of performance degradation is that any thread must be stalled (or aborted) when it attempts to read a datum already committed by a thread before finishing the transaction. Otherwise the isolation property gets violated.

Figure 9(a) shows the speed-up obtained with respect to the state-of-the-art scalable HTM Scalable-TCC (referred as S-TCC in the figure). We use the write-back commit feature of S-TCC where data reside in L1D till the next transactional modification. Speed-up numbers are normalized with 2 processors configuration and denoted on the top of the bar showing values corresponding to QuickTM. Here a value of n means a speed-up of nx times with respect to the Scalable-TCC for the same number of processor configuration. We consider 2, 4, 8 and 16 processors configurations. QuickTM gives an average speed-up of 20% over Scalable-TCC. Figure 9(b) shows the speed-up obtained with respect to the bus based conventional TCC (referred as C-TCC in the figure) without any parallel commit. Here, QuickTM gives an average speed-up of 28%. Note that, no application overflows with prescribed simulation input set for the given L1D size and associativity. Figure 9(c) shows the percentage of total execution time spent in commit for each application. Here, in case of parallel commits, we take the union of the commit times.

6.1. QuickTM Performance with Overflow

The Figure 10 compares the overflow handling performance of QuickTM with that of OneTM-concurrent[11]. We have created overflow for applications bayes and labyrinth for the 64KB L1D. With a smaller 16KB L1D we have compared two more applications, yada and vacation. All other architectural features between the two are same as QuickTM - only the overflow handling mechanism is different. Here QuickTM does not stall any thread upon conflict and after overflow partially commits after every 8 cache lines are modified. In OneTM-concurrent, stalled transactions restart execution after overflown transaction commits; whereas in QuickTM most stalled transactions commit immediately after the overflown transaction ends. This contributes in a speed-up with respect to the OneTM-concurrent. QuickTM outperforms OneTM-concurrent by 12% on average. The number of transaction overflows in bayes is small compared to labyrinth, hence QuickTM gives much higher performance for labyrinth.

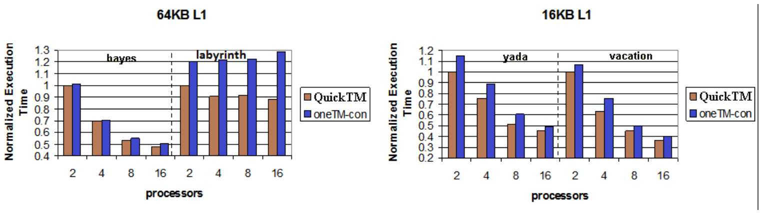


Figure 10. Performance comparison of QuickTM and OneTM-concurrent in presence of overflown transactions

7. Conclusion

In this paper we proposed QuickTM, a novel hardware transactional memory (HTM) architecture. We have combined several performance enhancement features to make it a robust and high performance HTM system. Firstly, we implemented a hardware only detection scheme of true-shared variables as the transaction executes. On-line separation of true-shared and thread-local variables has a very significant impact on the performance of HTM systems. This is because, on average, more than 80% (in some cases 99%) of data being modified in a transaction are thread-local and thus expendable from the commit overhead. Discarding plentiful thread-local accesses shortens each commit time drastically. This results in an overall speed-up. Secondly, in QuickTM , we have integrated this with another performance enhancing feature. Here, we fit both versions of the data (speculative and previously committed) in the same set but in different ways of L1D. This keeps all versions of data local to the processor. To achieve this we have proposed a novel cache coherence protocol. Finally, we have addressed the issue of transaction overflow. We have proposed to keep only one overflown transaction. However, all other threads keep executing their transactions. The overflown transaction partially commits its writeset at regular intervals which ensures that if other threads are not invalidated, then they can later read the committed data. QuickTM outperforms the current state-of-the-art Scalable-TCC[3] by

20% on average (with a maximum of 43%) in the state-of-the-art TM benchmark suite STAMP[10]. It outperforms the original bus-based TCC with serialized commit by 28% on average (with a maximum of 67%). With transaction overflow, QuickTM outperforms the OneTM-concurrent by 12% on average.

References

[1] T. Mattson, R. Van der Wijngaart, and M. Frumkin, "Programming the Intel 80-core network-on-a-chip terascale processor," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, pp. 1–11.

[2] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*, vol. 1063, no. 6897/04, pp. 20–00, 2004.

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

[4] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "LogTM: Log-based transactional memory," *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[5] M. Herlihy and J. Moss, *Transactional memory: architectural support for lock-free data structures*. ACM New York, NY, USA, 1993.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," *Proceedings of the 2006 ASPLOS Conference*, vol. 41, no. 11, pp. 336–346, 2006.

[7] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," *Lecture Notes in Computer Science*, vol. 4167, pp. 194–208, 2006.

[8] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott, "Lowering the overhead of nonblocking software transactional memory," in *Proceedings of the Workshop of Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. Citeseer, 2006.

[9] A. McDonald, J. Chung, B. Carlstrom, C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural semantics for practical transactional memory," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 53–65, 2006.

[10] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[11] C. Blundell, J. Devietti, E. Lewis, and M. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," *Proceedings of the 34th annual international conference on Computer architecture*, pp. 24–34, 2007.

[12] L. Yen, S. C. Draper, and M. Hill, "Notary: Hardware Techniques to Enhance Signatures," *Proceedings of the 41st annual International Symposium on Microarchitecture (MICRO)*, 2008.

[13] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero, "Dynamically Filtering Thread-Local Variables in Lazy-Lazy Hardware Transactional Memory," in *Proc. of the 11th IEEE International Conference on High Performance Computing and Communications (HPCC-09)*, 2009, pp. 171–179.

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

[15] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie, "Unbounded transactional memory," *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 316–327, 2005.

[16] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero, "Clock Gate on Abort: Towards Energy-Efficient Hardware Transactional Memory," in *Fifth IEEE Workshop on High-Performance, Power-Aware Computing (HP-PAC), in IPDPS*, May 2009.

[17] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "EazyHTM, Eager-Lazy Hardware Transactional Memory," in *Proc. of the 42th Intl. Symp. on Microarchitecture, NY, USA*, 2009.

[18] T. Harris, S. Tomic, A. Cristal, and O. Unsal, "Dynamic filtering: Multi-purpose architecture support for language runtime systems," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. ACM, 2010, pp. 39–52.

[19] R. Kessler, "The alpha 21264 microprocessor," *IEEE micro*, vol. 19, no. 2, pp. 24–36, 1999.

[20] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, and J. Sancho, "Experiences in scaling scientific applications on current-generation quad-core processors," in *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*, 2008, pp. 1–8.

[21] N. Jouppi, "Cache write policies and performance," in *Proceedings of the 20th annual international symposium on Computer architecture*. ACM New York, NY, USA, 1993, pp. 191–201.

[22] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE MICRO*, pp. 52–60, 2006.

[23] J. Bobba, K. Moore, H. Volos, L. Yen, M. Hill, M. Swift, and D. Wood, "Performance pathologies in hardware transactional memory," *Proceedings of the 34th annual international conference on Computer architecture*, pp. 81–91, 2007.