Open access • Proceedings Article • DOI:10.1109/SBAC-PAD.2011.18

Classification and Elimination of Conflicts in Hardware Transactional Memory Systems — Source link

Mridha-Mohammad Waliullah, Per Stenström

Institutions: French Institute for Research in Computer Science and Automation, Chalmers University of Technology

Published on: 26 Oct 2011 - Symposium on Computer Architecture and High Performance Computing

Topics: Transactional memory, False sharing and Cache

Classification and Elimination of Conflicts in Hardware Transactional Memory Systems

Mridha Mohammad Waliullah, Per Stenstrom

Classification and Elimination of Conflicts in Hardware Transactional Memory Systems

MM Waliullah
INRIA IRISA
Rennes, France
waliullah.mridha@gmail.com

Per Stenstrom
Chalmers University of Technology
Goteborg, Sweden
pers@chalmers.se

***Abstract*—This paper analyzes the sources of performance losses in hardware transactional memory and investigates techniques to reduce the losses. It dissects the root causes of data conflicts in hardware transactional memory systems (HTM) into four classes of conflicts: *true sharing*, *false sharing*, *silent store*, and *write-write conflicts*. These conflicts can cause performance and energy losses due to aborts and extra communication. To quantify losses, the paper first proposes the 5C cache-miss classification model that extends the well-established 4C model with a new class of cache misses known as *contamination misses*. The paper also contributes with two techniques for removal of data conflicts: One for removal of false sharing conflicts and another for removal of silent store conflicts. In addition, it revisits and adapts a technique that is able to reduce losses due to both true and false conflicts. All of the proposed techniques can be accommodated in a lazy versioning and lazy conflict resolution HTM built on top of a MESI cache-coherence infrastructure with quite modest extensions. Their ability to reduce performance is quantitatively established, individually as well as in combination. Performance is improved substantially.**

Keywords-Transactional Memory; Contamination Misses; Intermediate Checkpointing; Manycore;

I. INTRODUCTION

The shift to multicores has caused an acute need of new approaches to reduce the efforts in designing parallel programs. Transactional memory (TM) [12] is a promising approach to extract parallelism with potentially less effort. It does so by providing primitives to mark code blocks as atomic in a composable manner. Unlike traditional locking schemes such atomic blocks can be executed in parallel by offloading programmers from dependency analysis between atomic blocks.

TM has been proposed to be implemented in hardware on top of existing hardware (HTM), in software on top of existing hardware (STM), or in software using hardware acceleration [11]. Deployment of STM systems in practice remains questionable due to performance overheads. On the other hand, while HTM can be built on top of standard MESI cache-coherence protocols with a reasonable cost [18,20,23] they are prone to performance and energy losses caused by data conflicts triggered by accesses to the same cache block.

This paper dissects the root causes of data conflicts in HTM systems and their impact on performance. In the process of dissecting the root causes of data conflicts, we find that one class of conflicts – *true sharing conflicts* – cannot be avoided as it is caused by inherent communication among threads. The second class of conflicts – *false sharing conflicts* – is artifactual and shows up because conflicts are detected at the granularity of cache blocks rather than words. Two other classes of conflicts that we identify, and that can be avoided, are *silent store conflicts* [13] and *write-write conflicts*.

Conflicts are detrimental to both performance as well as energy consumption as they result in aborts that lead to wasted execution and additional cache misses. A second objective of the paper is to seek for methods for analyzing performance losses due to data conflicts. In this process, we note that transactional execution results in a new type of cache misses that stem from the fact that a speculatively modified, or contaminated, block has to be invalidated if the transaction is aborted. To this end, we propose the 5C cache-miss classification model that extends the well-established 4C model [7] with a new class of cache misses known as *contamination misses*.

Equipped with the root causes of data conflicts, the third objective of the paper is to propose techniques that can remove conflicts and their impact on performance. For false sharing conflicts we propose a scheme, inspired from Chen and Dubois [5], that uses two block sizes – one for conflict detection and one for transfers – to reduce the number of false sharing conflicts and to bring down the number of cold misses. As for silent store conflicts, we propose a scheme for silent store detection and elimination for transactional memory protocols by adapting previously proposed schemes aimed at cache coherence protocols [13]. While true conflicts cannot be removed, as they are inherent in parallel programs, their impact on performance can be reduced. To this end, we revisit a scheme earlier proposed by Waliullah and Stenstrom [24] for a TCC-like environment [10] that dynamically inserts a checkpoint before a conflict and rolls back to that checkpoint instead of to the beginning to reduce the amount of wasted work and its associated contamination misses. Our modified scheme leverages the eager conflict-detection capability of MESI protocols to achieve a high precision in insertion of checkpoints.

We show how the techniques can be integrated in a MESI-based lazy versioning and lazy conflict resolution HTM protocol with modest extensions. We consider the individual as well as the combined performance gains of the techniques. We find that these techniques individually as well as in combination are very effective in reducing the impact of data conflicts on performance. In summary, the paper makes the following contributions:

- It defines a framework for reasoning about the root causes of data conflicts in transactional memory systems and presents taxonomy for data conflicts.
- It proposes a new cache-miss classification scheme – the 5C model – in which a new type of cache misses – contamination misses – comprises the 5th C.
- It proposes how to integrate three techniques in a MESI cache protocol to remove or lessen the impact of data conflicts on performance and establishes the gains in isolation and in combination.

Section II establishes the architectural model and the framework for reasoning about the root causes of data conflicts and their impact

on performance. Section III presents the 5C miss-classification model. Section IV presents our proposed techniques and how they can be incorporated in the baseline system. The experimental methodology is described in Section V followed by our experimental findings in Section VI. We end the paper by putting our work in context to related work in Section VII before concluding in Section VIII.

II. Architectural Framework and Its Characterization

A. Baseline Architectural Framework

We consider a chip multiprocessor that has a number of processor cores with private L1 caches connected via a split-transaction bus to a shared L2 cache. Cache coherence among private caches is maintained with a snoop-based MESI cache coherence protocol. This system is a building block in a scalable tiled CMP architecture. This paper focuses on HTM for each such building block.

The HTM protocol supports lazy version management [2,4,10,21] and lazy conflict resolution [4,10,23] and is built on top of the MESI protocol. We choose lazy protocols as it uncovers more parallelism and the decreases likelihood of pathologies [25]. To maintain lazy versioning each cache line is extended with two bits: an SR (speculative read) and an SW (Speculative write) bit [10,18]. Conflicts are detected eagerly by the MESI coherence messages but are resolved lazily. Support needed to keep track of prior conflicts involves a bit map (KMAP) per node (core + L1) with as many bits as the number of nodes. If a snoop request reaches a remote node where the line is modified a *write conflict* signal is sent to the requester. On receiving the write conflict signal, the requester records the remote node as a possible 'killer' of the transaction by setting the corresponding bit in KMAP before performing the read or write. When a node commits, all transactions that have marked it in their KMAP will abort. A commit operation is carried out by sending a COMMIT message on the bus – no write set is broadcast.

Apart from the SR and SW bit, an RCONF (Read CONFlict) bit is associated with each cache line. RCONF indicates that the cache line is speculatively read and a conflict exists with a remote writer. Upon receiving a write conflict signal, in addition to recording the conflict in KMAP the RCONF bit is also set for the cache line. On abort, in addition to sending an ABORT message on the bus, all the cache lines with the SW bit set have to be invalidated. The cache lines that have the RCONF bit set have to be invalidated on both abort and commit. The abort message enables other transactions to reset the aborting node from their KMAP. All transactional metadata, e.g., SR, SW, KMAP are reset on both abort and commit.

B. Classification of Data Conflicts

While TM can expose concurrency, data conflicts (conflicts for short) force transactions to abort and serialize which lead to performance and energy losses. We explore the root causes of conflicts next. Conflicts are detected when a transaction speculatively read from a location that is speculatively modified by another non-committed transaction. Conflicts detection can be done lazily when a transaction commits or eagerly when it occurs as it is done in our baseline. Upon detection, a conflict can be resolved immediately (eager resolution) or deferred until a transaction commits (lazy resolution). For the eager resolution a conflicting transaction can be stalled to avoid a squash but in lazy resolution execution of conflicting transactions have to be squashed. In both cases performance is hampered.

Since conflicts are detected on the granularity of cache blocks, they come in two flavors – *essential (or true)* and *non-essential conflicts* – in analogy with cache misses/invalidations in a cache coherence protocol [9] as shown in Fig. 1.

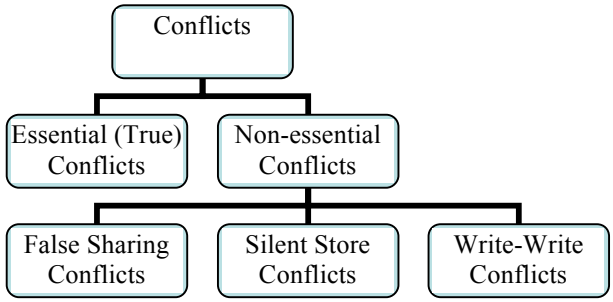


Figure 1. Classification of conflicts

A conflict is an *essential (or true) conflict* if any of the conflicting accesses to the same block refer to the same word in the memory and a new value is communicated. A true conflict cannot be avoided as it is triggered by communication inherent to the parallel program. However, the effect of true sharing conflicts can be reduced which will be considered in Section IV(B).

A conflict is a *non-essential conflict* if no real communication is made between conflicting transactions. Non-essential conflicts can be further classified into three different categories: *false sharing conflicts*, *silent store conflicts*, and *write-write conflicts*. A conflict is referred to as a false sharing conflict if the conflicting access pair refers to different words in the same cache line. False sharing conflicts can be eliminated by reducing the conflict detection granularity. Our experiments in later sections show that a significant amount of false sharing conflicts is introduced for commonly used cache line sizes.

A *silent store conflict* is a non-essential conflict where the write causing the conflict does not change the original value [13]; hence, no communication is made. Silent store conflicts can be avoided by simply ignoring certain protocol actions. A conflict is considered a *write-write conflict* if the conflict is caused by two transactions writing to the same location and no read is performed by any transaction prior to the write. While most existing HTM protocols take action [25] on such conflicts they could be ignored.

Section IV presents techniques to remove or lessen the impact of the conflicts and Sections VI will quantitatively establish how common the different conflicts are and to what extent their impact can be lessened by the proposed techniques.

III. A New Miss Classification Model

Many lazy versioning HTM designs [10,14] use private caches as temporary storage for speculatively modified data. On commit the data is made part of the consistent state while an abort causes speculatively modified (contaminated) lines to be invalidated. Re-executing the aborted transaction causes losses in performance as well as energy because of two reasons. First, the aborted transaction has to abandon the execution already done. Second, all cache lines that have been speculatively modified by the aborted transaction have to be invalidated. When these lines are accessed again, either when the aborted transaction is re-executed or later, they will cause extra cache misses that result in losses in performance and energy. This is a new type of cache miss resulting from contamination of cache blocks in the process of speculative modifications in a transaction. We call them *contamination misses* and they form the 5th C in our proposed 5C cache-miss classification model that extends the commonly used 4C model [7] (compulsory, capacity, conflict and

coherence misses) with an extra miss category. Contamination misses could be an interesting measure to evaluate HTM protocols.

In the miss classification method defined by Dubois et al. [9], a miss for a block is classified based on the reason it was evicted from the cache. In the context of a lazy versioning transactional memory system, a block is evicted from the cache because the block is replaced (capacity or conflict miss), the block is invalidated (coherence miss) or the block is contaminated and the transaction is aborted (contamination miss). Of course, if it is evicted because of a replacement and the replacement could have been avoided still it could have been evicted because of invalidations. In the following definition of a contamination miss, replacement misses have precedence over coherence misses, which have precedence over contamination misses if all are possible.

A miss is defined as a contamination-miss if the following conditions are fulfilled:

a. The block is evicted (invalidated) because it is contaminated by a transaction that is aborted.

b. There is no coherence invalidation request pending for the block when **a.** is performed.

While it may seem to suffice to only establish that the block was evicted because it was contaminated, it may actually happen that a coherence invalidation request is pending for the block. This might happen in a lazy conflict resolution HTM protocol where coherence invalidation for a speculatively read block is processed lazily. In Section VI, we will quantify the relative fractions of misses using the 5C model.

IV. Performance Improvement Techniques

A. *Multiple Cache-line Granularities (MCG)*

It is well known that trading off the cache line size is important to reap maximum performance from a cache memory hierarchy. In uniprocessor systems, larger cache lines exploit spatial locality to reduce misses whereas they also increase the probability of wasting space by bringing more data into the cache than needed. In multiprocessor systems, false sharing is introduced and the number of false sharing misses typically grows with the cache line size [9]. In a TM system, false sharing introduces the problem of false sharing conflicts. The performance impact of false sharing conflicts can be considerably higher than those of false sharing misses because a false sharing conflict may lead to re-execution of the entire transaction. Hence, trading off the cache line size in a TM system is more important than in a conventional cache coherent system.

To reduce the number of false sharing conflicts one must maintain conflict detection at a finer granularity which would call for smaller line sizes. However, smaller cache lines increase the number of cold misses. A way out of this dilemma, inspired by Chen and Dubois [5], is to support two line sizes: a larger line size, which is a multiple of the smaller line size, for transfer of non-shared blocks and a smaller line size for coherence invalidation. We call the technique as *multiple cache-line granularity* (or MCG for short). In the rest of the discussion, we refer to the larger blocks as *transfer blocks* and the smaller blocks as *invalidation blocks*. An invalidation block that is subject to an access request or a coherence message is called the *critical block*. Cache line metadata is maintained in invalidation block level.

In the proposed technique, when a memory access misses in L1, the L1 controller requests for the transfer block (i.e., multiple invalidation blocks with an indication of the critical block) if none of the invalidation blocks that are part of the transfer block exists in the cache. Otherwise, it requests only the critical block. Any other L1 cache that has the critical block forwards it along with other valid invalidation blocks that are part of the transfer block. If no L1 cache responds, L2 serves the request. An extra signal, *SingleLine*, is used which is set if any of the L1 caches has an updated copy of any of the invalidation blocks in the transfer block. If SingleLine is not set, the L2 cache transfers all the invalidation blocks that are part of the transfer block; otherwise L2 transfers only the critical block.

One alternative to MCG is sub-blocking and maintaining metadata for detecting conflicts at the sub-block level. MCG requires more space for tagging every small block whereas in sub-blocking a single tag entry is used for the entire block (analogous to the large block in MCG). However, all other transactional metadata have to be associated with each sub-block. The technical difficulty for sub-blocking in our baseline is that we allow a cache line to be available in the L1 caches for transactional accesses when the line is in the modified state in another L1. This is possible because of KMAP and RCONF bits. In a sub-blocking mode when a cache line is in the modified state it can modify any word without sending any coherence message. That makes it impossible to update RCONF bits in other L1s at the sub-block level. The second benefit of MCG over sub-blocking is the provision for adapting the size of the block in case of data sharing. In MCG, if any part of the transfer block is modified in any L1 only the critical block is served and two different nodes can work on two different invalidation blocks without any communication. In sub-blocking, the entire block has to be transferred even if another node is using a different sub-block.

B. *Intermediate Checkpointing (IC)*

Intermediate checkpointing (or IC for short) is a technique originally proposed by Waliullah and Stenstrom [24] that aims at reducing the amount of work that has to be discarded when a transaction aborts. The execution of a transaction is divided into two segments with respect to a conflict – *safe execution* and *unsafe execution* as shown in Fig. 2. Safe execution starts from the beginning of an execution until the transaction performs a conflicting access and the rest of the execution is referred to as unsafe execution. Ideally, an aborted transaction needs to squash only the unsafe part of the execution.

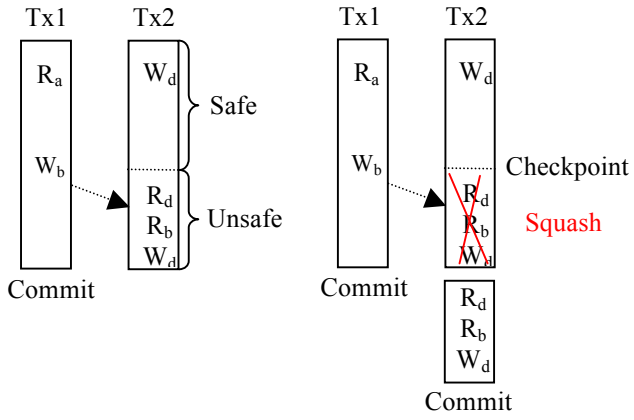


Figure 2. Safe and unsafe execution

As shown in Fig. 2, all of the execution of Tx2 need not be squashed; only the unsafe portion as shown in the right part of the figure must be squashed.

In the original intermediate checkpoint proposal [24] checkpoints are inserted to protect the safe execution from squashes. Each execution segment separated by checkpoints is called a *subtransaction*. An aborted transaction restarts from the beginning of

the earliest subtransaction that is unsafe. To be able to restart from the checkpoint, an undo log (iLog) is used to store old values in case a subsequent subtransaction modifies the same location. For example, location *d* in Fig. 2 is modified in both subtransactions separated by the checkpoint and if the transaction is restarted from the checkpoint the second write to location *d* has to be undone.

To support IC in the baseline requires that each cache block is associated with a pair of SR and SW bits for each subtransaction, a set of registers for each subtransaction, an undo log (iLog), and a mechanism for deciding when to take a checkpoint. Fortunately, it was shown in [24] that supporting as few as two subtransactions reap most of the benefits. In addition, an undo log is used assuming that locations are modified in subsequent subtransactions. As we will experimentally show in Section VI, this is not always the case and we will also evaluate an implementation of IC without an undo log. In a design without an undo log, modifications must be safely tracked so that the transaction rolls back to a safe point in case of an abort. A valid flag is associated with each checkpoint. If a subtransaction modifies a location that is modified in a previous subtransaction all previously taken checkpoints are invalidated. In that case, any conflict in these subtransactions leads to re-of the entire transaction.

In [24] a history-based prediction scheme is used for determination of a conflicting access and a checkpoint is inserted before performing such an access. We employ two techniques for inserting checkpoints. A conflicting location can be accessed in two scenarios: 1) no other transactions have yet speculatively modified the location 2) other transactions have already speculatively modified the location. In scenario 1, coherence messages will not raise any conflict and we use history-based prediction as in [24] to flag the access as a potential conflicting access. In scenario 2, the coherence message will raise a conflict and a checkpoint is inserted.

C. Suppressing Silent Store (SSS)

Every write to a shared location is potentially a source of aborts in transactional memory systems. Silent store [13] is a well-known phenomenon where the value carried with the modification is the same as the old value. Earlier studies [13] note that a silent store can be performed without invoking any cache protocol action but their impact on transactional memory systems is not studied.

The proposed *suppressing silent stores* technique (SSS for short) works as follows. First, if a write hits and the block is in shared state and the new value is found to be the same as the old value then the store operation is ignored. Neither is any coherence message sent nor is the SW bit set. Second, if the write misses in the cache a write request is sent (assuming a write-allocate cache protocol). Silent store detection can happen first when the block is returned. If the store is silent it can be ignored and need not set the SW bit. Hence, this will avoid conflicts with future readers.

D. Putting it All Together

Combining all the three techniques in the same HTM environment can be done straightforwardly. The first technique reduces false sharing conflicts by using smaller line sizes to detect conflicts but use larger transfer sizes to reduce the number of replacement misses. The second technique removes silent store conflicts by suppressing protocol actions for silent stores. The third technique reduces the wasted work due to essential as well as non-essential conflicts by not squashing safe execution.

The techniques are expected to improve HTM performance in isolation and in combination. However, the scope for boosting performance by these techniques is not orthogonal. For example, while MCG can eliminate false sharing conflicts, IC can reduce the wasted work due to such conflicts. Therefore, the combined effect of these techniques is not expected to be fully additive. In Section VI, we experimentally study their performance in isolation and in combination.

V. Experimental Methodology

To evaluate our techniques we extend the baseline system in Section II(A) with structures and protocol actions described in Section IV. The implementation is based on Simics [15], a full system functional simulator. The memory hierarchy simulation module for TM simulation tracks all the memory transactions at the clock-cycle granularity. The system is configured as a CMP that contains sixteen in-order processor cores interconnected via a split-transaction bus. A snoop-based MESI protocol is employed for maintaining coherence among the L1 caches. There are two independent buses – one for snoop and another for data. Snoop responses are synchronized with the request whereas data transfers are asynchronous. The bus width for data transfer is 32 bytes.

Each core has a 64-KByte private L1 cache, which is also being used for version management. As far as the cache-line size, we consider two default sizes: 32 and 64 bytes. We refer to the baseline with 32 and 64-byte cache lines as *Baseline32* and *Baseline64*, respectively. A 2-KByte size bloom filter is used for tracking evicted speculatively read (SR) lines whereas an 8-entry victim buffer stores evicted lines that are speculatively modified (SW). The assumed processor and bus clock frequency is 2 GHz, which means that the peak bandwidth of the split transaction bus is 64 GBytes/second. Table I summarizes the architectural parameters of the experimental system.

TABLE I. Architectural Parameters

Parameters	Values
Processors	16 in-order cores each running at 2 GHz
L1 Parameters	64KB, 4-way, 32/64 byte line size, LRU replacement, 2 cycles access latency
L2 Parameters	2MB, 16-way, 32/64 byte line size, Random replacement, 40 cycles access latency
Bus Bandwidth	64 GBytes/second
Memory Latency	200 cycles
OS & Arch.	Solaris 10 & Sparc V9
Compiler	Gcc 4.1.2, -O2

For the MCG mechanism, we use 32 bytes as invalidation line size and 64 bytes as transfer line size. In case of IC with an iLog, it uses 128 entries buffer. We present results for IC both with and without the iLog buffer. Based on the observations in [24], the IC implementation inserts a single checkpoint.

We use the STAMP [17] benchmarks that comprise eight applications written with transactional semantics. Simics' magic instruction is used to annotate begin and end of transactions. The input parameters used follows the recommendations given in [17]. Due to the inconsistent behavior reported in previous studies [20] we exclude the application Bayes from our experiments. Another application, Labyrinth, copies a shared maze in local data structure at the beginning of each transaction. This leads to a potential conflict even if two transactions work on two independent segments in the maze. As indicated in the source code, early release of the read set is the trick to avoid it. However, our HTM design does not support early release. To avoid serialization, we have modified the original source code so that the shared maze is accessed on demand. The detailed application parameters are given in Table II.

To reduce the impact of simulation variability and specific scheduling effects we use the methodology described in [1] by

Alameldeen and Wood. For each configuration, we run five simulations where each run uses memory latency within the 5% range of the actual parameter (200 cycles). We then take the average of the results.

TABLE II. APPLICATION PARAMETERS

Applications	Parameters
Genome	-g256 –s16 –n16384
Intruder	-a10 –l4 -n 2048 –s1
Kmeans	-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt
Labyrinth	-i random-x16-y16-z3-n32.txt
SSCA2	-s13 –i1.0 –u1.0 –l3 –p3
Vacation	-n2 –q90 –u98 –r8192 –t4096
Yada	-a20 –i633.2

VI. EXPERIMENTAL RESULTS

A. Baseline Performance Characteristics

We first analyze the performance of the baseline system for the two cache-line sizes. In the diagrams of Fig. 3(a) and 3(b), the left and right bars for each application represent results for a 32-byte cache-line size (Baseline32) and a 64-byte cache-line size (Baseline64), respectively. Fig. 3(a) shows the execution time of the STAMP applications for Baseline32 (left) and Baseline64 (right) while the later is normalized to Baseline32. Execution time is further decomposed into three categories. *Squash* represents wasted cycles due to squash, *Commit* represents cycles spent on successfully committed transaction and *NonTX* represents cycles spent on non-transactional execution. The number below each bar is the standard deviation of the execution time across the five runs with different memory latencies. As we can see, the standard deviation is in general very low. We see that three applications (Intruder, Yada, and Labyrinth) suffer from a huge number of squashes which have a detrimental effect on execution time. Two different trends are visible. Firstly, applications that suffer from squashes in Baseline32 deteriorate further in Baseline64. Secondly, the applications that do not suffer from squashes benefit from 64-byte cache lines.

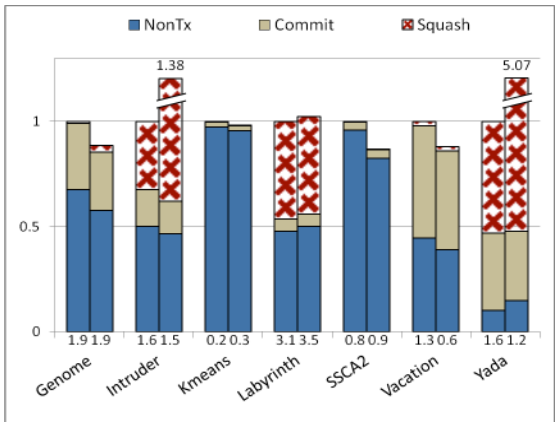
Fig. 3(b) depicts breakdown of the wasted work (called Squash in Fig. 3(a)) in the three applications that suffer significantly from squashes. In the diagram, *True Sharing* represents percentage of wasted cycles due to true sharing conflicts, *False Sharing* represents percentage of wasted cycles due to false sharing conflicts, and Silent Store and write-write represent that of silent store and write-write conflicts, respectively.

We can see that going from 32-byte to 64-byte cache lines the ratio of false sharing conflicts increases significantly which explains the first trend in Fig. 3(a). As we will confirm later, the second trend is due to the reduced number of 3C misses for 64-byte cache lines compared to the 32-byte cache lines. The diagram shows a very little impact of silent store conflicts and zero impact of write-write conflicts. The results clearly show that conflicts are a serious contributor to performance losses in the baseline HTM system and reinforce the need for the techniques to reduce it.

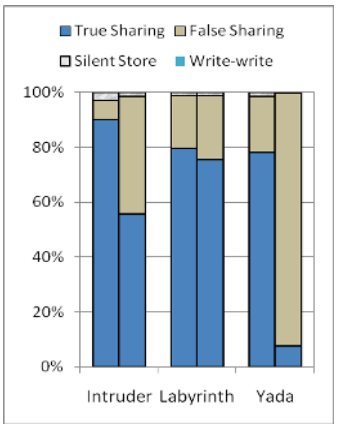
B. Cache Miss Classification and the Frequency of Contamination Misses

To provide a deeper insight into the performance differences of Baseline32 and Baseline64, we examine the relative frequency of different categories of cache misses using the 5C cache model introduced in Section III. Fig. 4 shows cache miss breakdown in both baselines. The left and the right bars in each cluster represent results for Baseline32 and Baseline64, respectively. The misses are classified into three major categories – the bottom section lumps together cold, capacity and conflict misses (3C), the middle and the top section represent coherence and contamination misses, respectively. As expected, contamination misses only appear in the applications that suffer from squashes (Intruder and Yada, in particular). Another important confirmation is that the 3C (cold, conflict and capacity miss) component is reduced as we go from a 32-byte system to a 64-byte system. This observation is leveraged in the MCG technique.

Fig. 5 shows the performance losses due to contamination misses. Again, the left and the right bars in each cluster represent results for Baseline32 and Baseline64, respectively. The figure depicts the percentage of the execution time that is spent on serving contamination misses. As we can see, performance losses due to contamination misses in Intruder and Yada are quite substantial even in the tightly coupled bus-based system with low (on-chip) miss latencies that we assume. Contamination misses can be more costly in multi-chip systems that experience higher latencies. Even though the contamination miss rate in Baseline64 (Fig. 4) is significantly higher than in Baseline32 for Yada the performance penalty bars look similar. This is because the penalties are normalized to the execution time of the respective baselines. The goal here is to show the significance of contamination misses.



(a) Execution time breakdown



(b) Conflict breakdown

Figure 3. (a) Normalized execution time breakdown of the applications. For each configuration we run five simulations as described in Section V and then take the average. Relative standard deviation (in percentage) is given at the bottom of the respective bar. (b) Conflict breakdown of the three applications that suffers significantly from squashes. In both diagrams, left and right bars of each application represent Baseline32 and Baseline64, respectively.

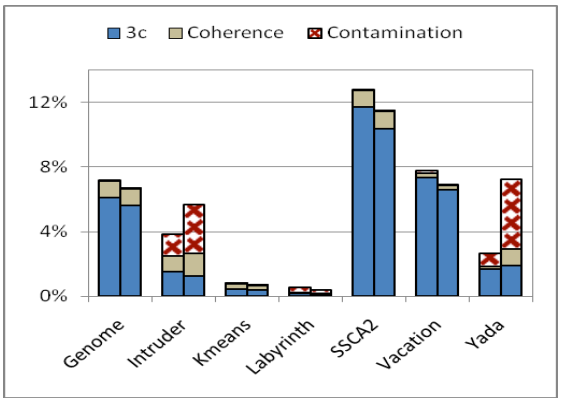


Figure 4. Miss rates in the 5C model

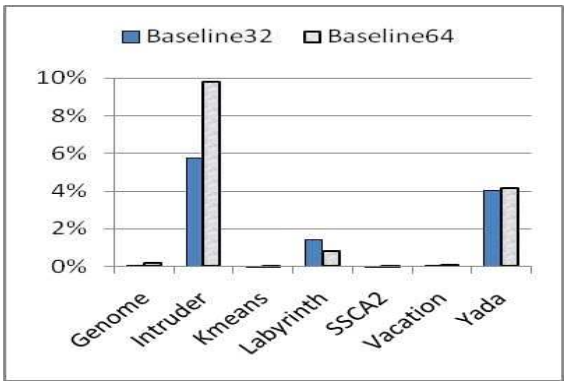


Figure 5. Performance penalties of contamination miss

C. *Performance Analysis of MCG*

To get the benefit of both large cache lines (fewer 3C misses) and small cache lines (fewer false sharing conflicts) we adopt the mechanism where data transfer is done in 64-byte chunks and invalidations use 32-byte lines. In Fig. 6(a), the left bars represent the execution time of Baseline32 and the right bars represent the execution time of Baseline32 enhanced with MCG. The right most single-bar shows the geometric mean of the execution time of the MCG technique where the percentage of improvement over the baseline appears at the top. Fig. 6(b) represents similar numbers for Baseline64.

In Fig. 6(a) we see that the execution time is reduced by between 4% and 15% across the applications (on average 8%). In the enhanced MCG system, using 32-byte invalidation line size the conflict behavior is the same as in Baseline32 but 64-byte transfer line sizes exploit spatial locality which provides a performance boost. In Fig. 6(b), MCG in this case results in an average performance improvement of 24%. As expected, the more dramatic improvement stems from the fact that 64-byte cache lines in this baseline result in lots of false sharing conflicts of which quite many are eliminated in the MCG enhancement by using 32-byte invalidation granularity. We also see lower execution time for SSCA2 which does not exhibit any false conflicts. We observe lower conflict misses for this application in the enhanced system. Our conjecture is that it is an effect of a smaller granularity of cache line management that utilizes cache space appropriately.

D. *Performance Analysis of IC*

We analyze the impact of intermediate checkpointing (IC) on the performance losses caused by conflicts for the three applications that suffer from significant number of squashes. Even though the technique is effective for all applications conflicts in other applications are not significant to have an impact on overall execution time. In Fig. 7(a) the left and right bars for each application represent the execution time on Baseline32 without and with IC-with-iLog, respectively. The rightmost single bar shows the geometric mean of the execution time for Baseline enhanced by IC-with-iLog. The average reduction in execution time is depicted on the top of the bar. The data assuming Baseline64 is shown in Fig. 7(b).

In the figure, we see that for all the three applications, IC reduces the execution time in both baselines. On average, we see 8% and 13% reduction of the execution time in Baseline32 and Baseline64 respectively. We have also experimented with IC without iLog. We see that the execution time for Intruder remains the same but for Labyrinth and Yada no improvement over the baseline is observed. The reason is that these two applications have large transactions and modify certain cache lines before and after the IC. To get benefit from IC in such situations requires an iLog.

E. *Performance Analysis of SSS*

Fig. 8 represents the normalized execution time in systems that implement SSS. Fig. 8(a) represents results for Baseline32 and Fig. 8(b) represents the results for Baseline64. We see that in general there is no significant performance impact by implementing SSS which is not so surprising considering the very low amount of silent store conflicts observed in Fig. 3. One interesting aspect of SSS is that it can degrade performance if a transaction has to abort after suppressing silent store. In that case, SSS will just delay the abort instead of rescuing the transaction. We conclude that for the set of applications studied, essential and false-sharing conflicts are the most important root causes.

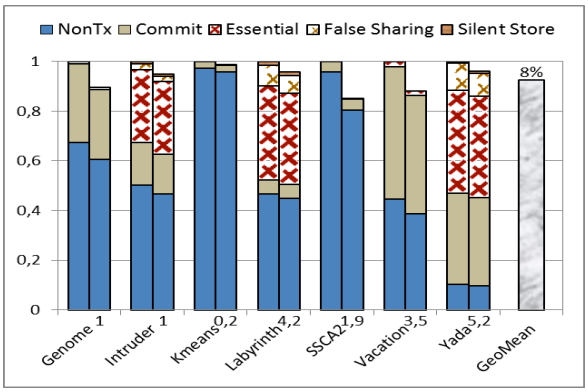
F. *Combined Effect of the Techniques*

Finally, we combine the techniques and study their impact on performance on Baseline32 and Baseline64. Fig. 9(a) represents the execution time of each of the techniques in isolation and in combination normalized to that of Baseline32 and Fig. 9(b) represents the same data for Baseline64. For each application, the four bars correspond to (from left to right) the execution time of SSS, IC, MCG and the Baseline with all the techniques, respectively. For each configuration we run five simulations and then take the average. As we see in the previous results standard deviation of the runs is within 5% of the average.

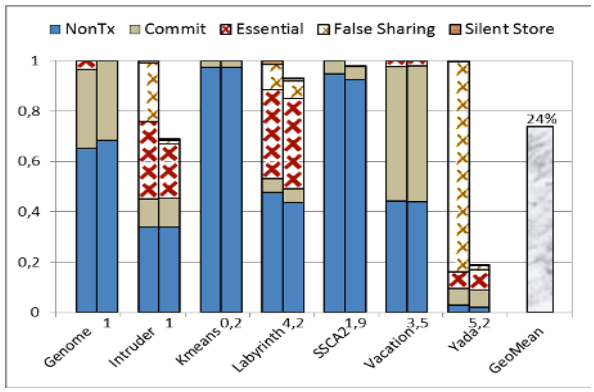
We see that combining the techniques we get on average 10% reduced execution time for Baseline32 and 28% reduced execution time for Baseline64. We get more performance in Baseline64 because of the enormous amount of false sharing conflict in that baseline.

VII. RELATED WORK

Several studies have been published in the past to reduce false sharing misses in invalidation-based cache coherence protocols. Chen and Dubois [5] partition the address block into several invalidation blocks to make invalidation granularity lower than the transfer granularity. Dahlgren et al. [8] propose sequential hardware prefetching to exploit spatial locality. In their proposal, k consecutive blocks are prefetched on a cache miss. These studies try to exploit spatial locality and remove false sharing misses in conventional cache-coherent infrastructures. This study revisits these issues in the context of transactional memory systems.

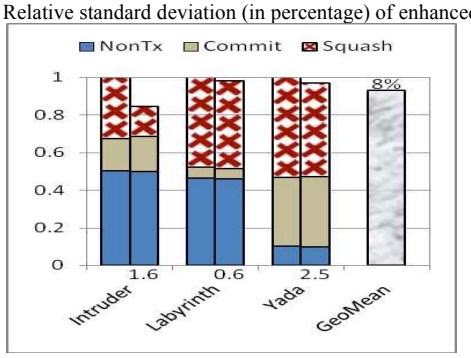


(a) MCG in Baseline32

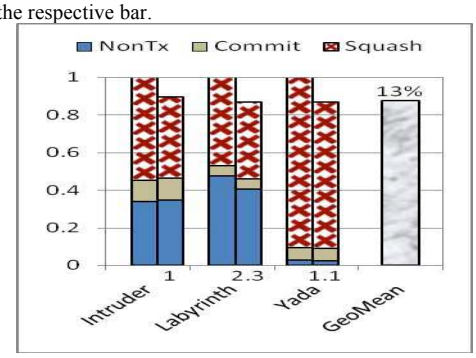


(b) MCG in Baseline64

Figure 6. (a) Execution time of MCG normalized to the Baseline32. (b) Execution time of MCG normalized to the Baseline64. For each application, the left bar represents execution time of the baseline and the right bar represents the enhanced system. For each configuration we run five simulations and then take the average. Relative standard deviation (in percentage) of enhanced system is given at the bottom of the respective bar.

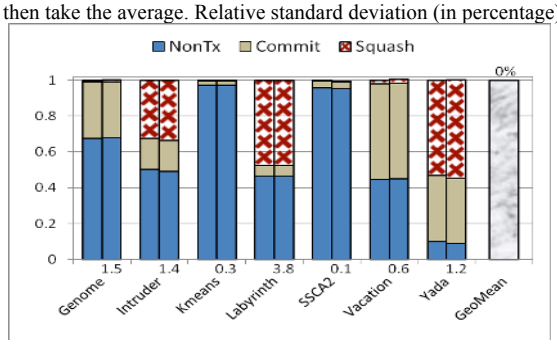


(a)

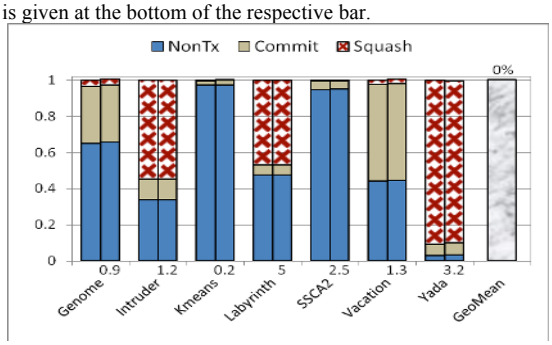


(b)

Figure 7. (a) Execution time of IC normalized to the Baseline32. (b) Execution time of IC normalized to the Baseline64. For each application, the left bar represents execution time of the baseline and the right bar represents the execution time of the enhanced system. For each configuration we run five simulations and then take the average. Relative standard deviation (in percentage) of the enhanced system is given at the bottom of the respective bar.

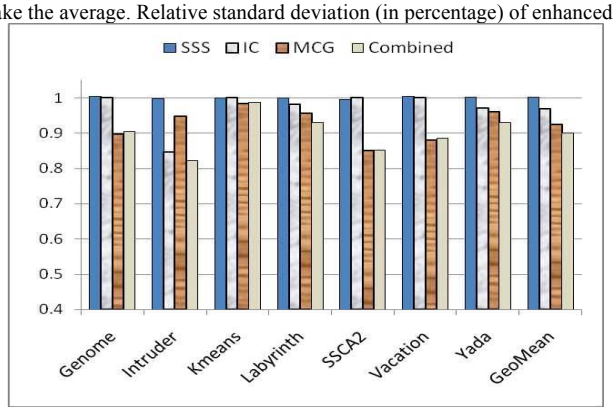


(a)

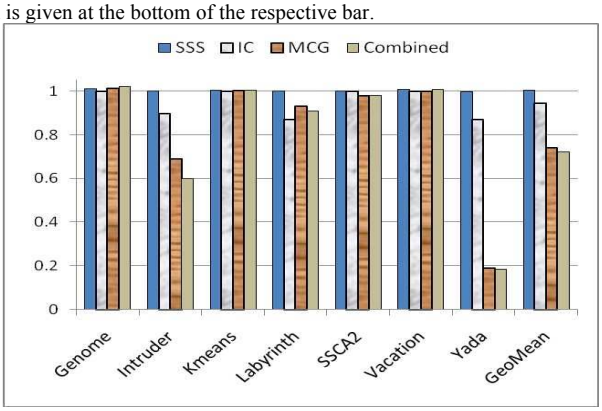


(b)

Figure 8: (a) Execution time of SSS implemented and normalized to the Baseline32. (b) Execution time of SSS implemented and normalized to the Baseline64. In each cluster, left bar represents execution time of the baseline and right bar represents the enhanced system. For each configuration we run five simulations and then take the average. Relative standard deviation (in percentage) of enhanced system is given at the bottom of the respective bar.



(a) Enhanced Baseline32



(b) Enhanced Baseline64

Figure 9. Execution time of enhanced systems in isolation and in combination.

Intermediate checkpointing to reduce wasted work has been proposed by Waliullah and Stenstrom [24]. In that work, intermediate checkpointing is analyzed in a TCC-like HTM design space. In this work, we have analyzed it in the context of MESI-based HTM designs. The new opportunity is to use eager conflict detection to make more accurate insertions of checkpoints. We also studied the impact of the undo log on the efficiency. Colohan et al. [6] proposed another similar work in the context of thread level speculation. In that work, the authors propose sub-threading by inserting checkpoints after a fixed number of instructions and do not take conflicting accesses into account. One can also compare nested transaction [16,19] with intermediate checkpointing. While nested transaction is a software concept intermediate checkpointing is a dynamic hardware technique that optimizes execution of transactions.

Silent store in the context of transactional memory is captured in transactional value prediction (TVP) scheme proposed by F. Tabba et al. [22]. In the TVP scheme, a transaction is allowed to proceed even if a read hits a line that is stale in the cache. A store is performed without sending any exclusive write request. Correctness is ensured by validating all memory operations before commit. The validation is done by comparing the consumed data with the latest version. In the process, the effect of false sharing and silent store is nullified. While TVP is built on top of a revised TM protocol that ignores cache coherence messages for conflicts our scheme is built on top of a standard MESI cache coherence protocol.

Bobba et al. [3] introduce a framework for reasoning about performance tradeoffs between HTM systems with respect to version and conflict resolution management. They identify seven performance pathologies that help in selecting an optimal strategy for version and conflict management. Once that strategy is established, the resulting HTM system can still suffer from conflicts that result in performance losses. The framework presented in this paper helps understanding the root causes of the remaining conflicts so that proper optimizations can be applied.

VIII. Concluding Remarks

This paper studies the root causes of data conflicts in hardware transactional-memory systems (HTM). Four classes of conflicts are identified: true sharing, false sharing, silent store, and write-write conflicts. In order to quantitatively establish the losses in performance, we extend the 4C model for cache miss classification with a new category called contamination misses. We consider several techniques to address the root causes of conflicts in HTM systems. In particular, we contribute with a technique to reduce the number of false sharing and silent store conflicts and revisit intermediate checkpointing to reduce the impact of conflicts regardless of root cause.

Overall we find that true and false sharing conflicts can have a significant impact on performance on HTM systems whereas conflicts due to silent stores and write-write conflicts are not common. While most of the performance losses stem from re-execution of transactions due to aborts, extraneous communication in servicing contamination misses is another important source. The proposed techniques can be integrated with modest efforts. By especially supporting finer-grain cache line sizes for conflict detection and intermediate checkpointing we show that on average performance can be improved by 10% on a baseline with 32-byte cache lines and 28% on a baseline with 64-byte cache lines.

Acknowledgements

This research is partially sponsored by the SARC and the VELOX project funded by the EU. Most of the work is done when the first author was at Chalmers as a PhD student. The authors are members of HiPEAC – a Network of Excellence funded by the EU. The first author is an ERCIM postdoctoral fellow at INRIA.

References

[1] A. R. Alameldeen and D. A. Wood, "Variability in Architectural Simulations of Multi-threaded Workloads", *Proceedings of the 9th Annual International Symposium on High-Performance Computer Architecture*, Anaheim, CA, February 8-12, 2003.

[2] C. S. Ananian, K. Asanovi'c, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory", *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, San Francisco, CA, pp. 316-327, February 2005.

[3] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance Pathologies in Hardware Transactional Memory", *Proceedings of the 34th International Symposium on Computer Architecture*, June. 2007.

[4] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors", *Proceedings of the 33rd International Symposium on Computer Architecture*, June 2006.

[5] Y.S. Chen and M. Dubios, "Cache protocols with partial block invalidations", *Proceedings of 7th International Parallel Processing Symposium*, CA, USA, April, 1993.

[6] C. B. Colohan, A. Aliamaki, J. G. Steffan, and T. C. Mowry, "Tolerating Dependences Between Large Speculative Threads Via Sub-Threads", *Proceedings of the 33rd International Symposium on Computer Architecture*, pp. 216-226, Boston, MA, June, 2006.

[7] D.E. Cullar, A. Gupta and J.P. Singh, "Parallel Computer Architecture: A Hardware/Software Approach", Morgan Kaufmann Publishers Inc., CA, USA, 1998.

[8] F. Dahlgren, M. Dubois, and P. Stenstrom, "Sequential hardware prefetching in shared-memory multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, Issue 7, July, 1995.

[9] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy and P. Stenstrom, "The detection and elimination of useless misses in multiprocessors", *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, California, US, 1993.

[10] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, J. Davis, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency", *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 102-113, München, Germany, June 19-23, 2004.

[11] T. Harris, J. Larus and R. Rajwar, "Transactional Memory", *Synthesis Lectures on Computer Architecture*, Vol. 5, No. 1, June 2010.

[12] M. Herlihy and J. E. B. Moss, "Transactional Memory: architectural support for lock-free data structures", *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 289-300, May 1993.

[13] K. M. Lepak, G. B. Bell and M. H. Lipasti, "Silent Stores and Store Value Locality", *IEEE Transactions on Computers*, vol. 50, Issue 11, November 2001.

[14] M. Lupon, G. Magklis, A. Gonzalez, "FASTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery", *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, 12-16 Sept. 2009

[15] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform", *IEEE Computer*, vol 3, issue 5, page 50–58, February 2002.

[16] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis and K. Olukotun, "Architectural Semantics for Practical Transactional Memory", *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, Boston, MA, June 17-21, 2006.

[17] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," *Proceedings of the International Symposium on Workload Characterization*, September 2008.

[18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill and D. A. Wood, "LogTM: Log-based Transactional Memory", *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)*, pages 258–269, Austin, TX February 11-15, 2006.

[19] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Seift, and D. A. Wood, "Supporting nested transactional memory in LogTM", *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSXII)*, pp. 359-370, 2006.

[20] A. Negi, MM Waliullah and P. Stenstrom, "LV*: A low complexity Lazy Versioning HTM Infrastructure" ", *Proceedings of 10th IEEE IC-SAMOS,* July 2010.

[21] R. Rajwar, M. Herlihy, and L. Lai, "Virtualizing transactional memory", *Proceedings of the 32nd International Symposium on Computer Architecture*, pp. 494-505, June 2005.

[22] F. Tabba, A. W. Hay, and J. R. Goodman, "Transactional Value Prediction", *Proceedings of the ACM SIGPLAN Workshop on Transactional Computing*, February 2009.

[23] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Haris, and M. Valero, "EazyHTM: Eager-Lazy Hardware Transactional Memory", *Proceedings of the 42nd International Symposium on Microarchitecture*, New York, December 2009.

[24] M. M. Waliullah and P. Stenstrom, "Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems", *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Miami, Florida USA, April 2008.

[25] A. Shriraman and S. Dwarkadas, "Analyzing Conflicts in Hardware-Supported Memory Transactions", International Journal of Parallel Programming, vol. 9, number 1, pp. 33-61, 2010.