2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Reducing False Transactional Conflicts With Speculative Sub-blocking State — An Empirical Study for ASF Transactional Memory System

Lifeng Nai, Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
{*lnai3, leehs*}*@gatech.edu*

***Abstract*—Conflict detection and resolution are among the most fundamental issues in transactional memory systems. Hardware transactional memory (HTM) systems such as AMD's Advanced Synchronization Facility (ASF) employ inherent cache coherence protocol messages to perform conflict detection among transactions. Such an implementation has the advantage of design simplicity, nonetheless, it also generates false transactional conflicts due to false sharing within cache lines, unnecessarily reducing the overall performance.**

In this work, we first investigated the behavior of false transactional conflicts under the AMD's ASF system. It is found that false conflicts showed rather stable pattern within each cache line that subsequently inspired our false transactional conflict reduction technique using our proposed speculative sub-blocking state. By adding an extra speculative state for each cache line's sub-block, we can maintain conflict detection at the granularity of sub-blocks while keeping the original cache coherence protocol intact. The overall design is simple and highly implementable for achieving a high-efficiency HTM system with minimum impact in hardware.

We evaluated our proposed technique using PTLsim-ASF and compared it with a baseline ASF HTM system and an ideal system with no false transactional conflict. Our results showed that the proposed lightweight technique can avoid false conflicts effectively and efficiently. With four sub-blocks in a cache line, our technique can eliminate 56.4% false transactional conflicts and 31.3% of all transactional conflicts on average, which approaches the performance of an ideal system.

***Keywords*-transactional memory; false conflict; parallel programming; microarchitecture;**

I. Introduction

In the era of multi-core computing, scalability is becoming the major concern of machine learning and data mining applications. In order to scale the performance of algorithms, parallel programming models as well as their required hardware support have received more and more attention. The goal is to provide a holistic solution to improve programmability, productivity, performance, and power. Toward this end, Transactional Memory (TM) [1], proposed as a replacement technique for traditional lock programming, provides much promise to meet the demand for future multi-core programming. TM allows programmers to write atomic code easily without concerning how the atomicity is achieved and maintained. Besides the ease of programmability, it also provides performance benefits from potentially concurrent execution of parallel transactions.

Numerous TM-based systems have been proposed and implemented. A TM system can be achieved in software transactional memory (STM) [2], hardware transactional memory (HTM) [3], [4], [5], or a hybrid system [6]. A slew of these research works particularly focused on implementation issues such as memory versioning, conflict detection, and isolation property. Moreover, microprocessor industry has also caught up and started to develop and provide TM support. For example, Sun/Oracle's Rock processor leveraged the common speculative hardware features shared by HTM and runahead execution to enable both execution models [7]. IBM announced TM support in their upcoming BlueGene/Q processor for the Sequoia supercomputer [8]. Intel also disclosed the ISA details of their version of TM support called TSX to appear in their next-generation microarchitecture code-named Haswell [9]. Among these endeavor, AMD, one of the earliest HTM proponents, introduced Advanced Synchronization Facility (ASF) [10] to support HTM on top of the x86 ISA. Instead of supporting complicated optimization methods, such as unbounded transactions [3] and active transactional scheduling [11], [12], AMD's proposal attempted a *best-effort* system that keeps hardware simplicity and minimum impact on whole system.

In an ASF-enabled HTM system, conflicts are detected by incoming cache coherence messages and the speculative state information is added onto each cache line. Due to the coarse granularity data management, false sharing between different cores will be present when different cores attempt to access non-overlapped data bytes within the same cache line. This is the same performance problem causing ping-pong effect and studied in prior shared-memory multiprocessor systems [13]. Worse yet, in an HTM system, the same issue will lead to false conflicts among transactions and cause unnecessary transaction aborts. In other words, useful works are disrupted and discarded, wasting power and degrading the overall performance of an HTM system. Our experimental results of STAMP [14] and RMS-TM [15] benchmark suites showed that as much as 46.7% of all transactional conflicts could be attributed to false conflicts.

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.113

Although conflict detection is a rather basic issue in an HTM system, ignoring this harmful behavior could substantially reduce the effectiveness of a TM system. This cost will be too expensive in exchange of the benefit of its programmability. Therefore, it deserves reconsideration for our current TM hardware design to minimize its destructive effect on performance.

False conflicts, in general, largely depend on applications' characteristics, such as sharing pattern, access pattern, and data allocation. The uncertainty of applications make them hard to mitigate. Typical solutions for false sharing issue in shared-memory system are less feasible when it comes to HTM systems. And previously proposed HTM specified false conflict reduction techniques [5], [16], [17] also have their shortcomings, making them infeasible for AMD's ASF-like HTM system. We will further elaborate them later.

To address the shortcomings of the prior work, our goal is to develop a technique that can efficiently reduce false conflicts with minimum impact on the original architecture design. In this paper, we propose a transactional false conflict reduction mechanism based on speculated sub-blocking state. Our technique maintains speculated states at the granularity of cache sub-block while keeping the original coherence protocol intact. The technique is lightweight, and highly implementable on top of an existing system supporting ASF-like hardware transactional memory. Using our technique, the false conflict rate can be substantially reduced while the design complexity and impact on general system are both kept in minimum. To evaluate our proposed technique, we ported STAMP [14] and RMS-TM [15] benchmark suites using AMD's ASF instruction extension, and performed performance evaluation using PTLsim-ASF simulator provided by AMD [18]. Our results show that with four sub-blocks in a cache line, our technique can reduce 56.4% false transactional conflicts and 31.3% of all transactional conflicts on average, which approaches the performance of ideal situation without any false conflict. The main contributions of this paper are:

- We ported STAMP [14] and RMS-TM [15] benchmark suites and analyzed the behavior and performance impact of false transactional conflicts in a commercial HTM system based on AMD's Advanced Synchronization Facility (ASF).
- We proposed a lightweight, highly implementable hardware solution that enables the maintenance of conflict detection at a finer granularity, *e.g.*, at the cache sub-block level.
- We evaluated the effectiveness of our hardware technique in resolving transactional false conflicts for an ASF-like TM system using our ASF-enabled TM benchmark suites on PTLsim-ASF simulator.

The rest of the paper is organized as follows. Section II discusses the background, related work and shortcomings of past works. As a motivation of this paper, false conflict behavior is analyzed in Section III. The implementation details are explained in Section IV. Section V evaluates our technique by comparing it to the baseline ASF system and a perfect system without false conflicts. Section VI summarizes the main conclusions of our work.

II. Background and related work

In HTM systems, transactional conflicts are always the most prominent problem affecting the overall execution performance. Some HTM systems, such as LogTM [5], proposed to detect conflicts with memory address signature and special cache coherence messages, while others, such as AMD's ASF [10] an HTM support to be commercially available for x86 architecture, was designed to infer transactional conflicts from an unmodified cache coherence protocol, which can lead to false transactional conflicts due to false sharing at the granularity of cache lines.

False sharing, a common performance issue in programming shared memory multiprocessor systems, was studied in prior works [13]. It can cause unnecessary bus traffic and even ping-pong effect, degrading the overall performance. Common solutions to addressing the false sharing problem include data structure re-grouping and maintaining coherence at a finer granularity using sub-blocks. Unfortunately, these prior proposals are less feasible in HTM systems because of the different features and requirements of HTM environment.

A software reconstruction technique requires specially tailored modification on code and data according to cache line size. It can be done either by programmers or by smart compilers and could be efficient if applied properly. If it is done by programmers, it would be quite impractical because it contradicts the most essential goals of using a TM system, *i.e.*, ease of programmability and programming productivity. TM systems were conceived to offer parallel code programmers a good high level abstraction in order to make parallel programming easy and efficient. It would surely be improper to require programmers to write low-level code to avoid false conflicts. On the other hand, if relying on smart compilers to accomplish the reconstruction work, those special tailored code and data structure will only take effect in specific hardware, which makes software not portable.

As for the hardware technique, it obviously can reduce false conflicts with its smaller cache coherence granularity. But it also requires special modification to the original cache coherence protocol. Such modification may bring unknown impact on the processor's general performance. An HTM system like ASF implemented in modern out-of-order core can be very complex for general purpose high-performance computation. The critical structures such as cache coherence protocol are better left intact rather than vastly modified by new ideas for specific architectural feature.

The false conflict issues in HTM systems were observed previously in [5], [16], [17]. Porter *et. al* [16] discussed the impact of different coherence granularities on false sharing in their speculative multi-threading (SpMT) environment. They also proposed a speculation-based method to eliminate false sharing by speculating the presence of true conflicts and trying to validate them later. Tabba *et. al* [17] proposed an HTM system called DPTM. They implemented the coherence decoupling technique proposed by Huh *et. al* [19] to reduce false conflicts.

Their methods share significant similarity in false sharing reduction issue. In their methods, whenever a cache line containing read data is invalidated, they always speculate that there is no true conflict and continue their execution without abortion. In the SpMT's solution, it marks the speculated cache lines as *unsafe* and check the validity by data comparison when they are accessed again, invalidated, and committed. In DPTM, it only validated at commit time without the marking mechanism.

SpMT does analysis and evaluation work in speculative multi-threading environment; the impact of false conflicts on pure HTM systems is still unknown. DPTM's technique brings benefit from false conflict reduction besides their general performance gain from Huh's original idea of value prediction [19]. Both of their techniques share similar drawbacks. First, they can only handle false conflicts caused by write-after-read cache lines. From the simulation results in Figure 2, we can see that read-after-write false conflicts also have quite significant portion. Ignoring this type of false conflicts will miss out great opportunities for performance optimization. Secondly, their techniques impose lazy conflict detection constraint on HTM systems, even if the original ones employ an eager conflict detection policy for certain considerations. It may break the original system's design philosophy and result in performance loss in some HTM systems.

III. Motivation

A. False conflict behavior

In this section, we analyze the characteristics of false conflicts in an ASF-enabled HTM system to motivate our work. We first ported STAMP [14] and RMS-TM [15] benchmark suites using AMD's ASF instruction extension [10]. We then simulated these TM benchmark suites using PTLsim-ASF [18] which was further modified to facilitate our characterization and analysis work.

First of all, we analyzed how serious false conflicts are in our target ASF-enabled HTM system and tried to understand the requirement of its solution space. Figure 1 shows the false conflict rate of STAMP and RMS-TM benchmark programs in AMD's ASF HTM system. [1] As shown in Table III, the benchmark suites we chose include several representative machine learning and data mining applications. From the result in Figure 1, we found that sometimes the false conflict rates cannot be easily disregarded. Most of the benchmark programs we experimented show a false conflict rate more than 40% while the rate can go as high as more than 90% in graph computing kernel ssca2 and mining application Apriori. The false conflict rates of other applications also show significant high number, such as the mining program utilitymine and learning application kmeans. On average, the false conflict rate is around 46%. Although conflict detection among transactions is considered a basic issue, prior studies focused much more on the efficient implementation of conflict detection mechanism but paid less attention in its performance implication. As shown in our experiments, to make transactions more effective and waste less processing power in an ASF-like HTM system, we should also consider, in our HTM design, the resolution of minimizing the false conflict rate among transactions.

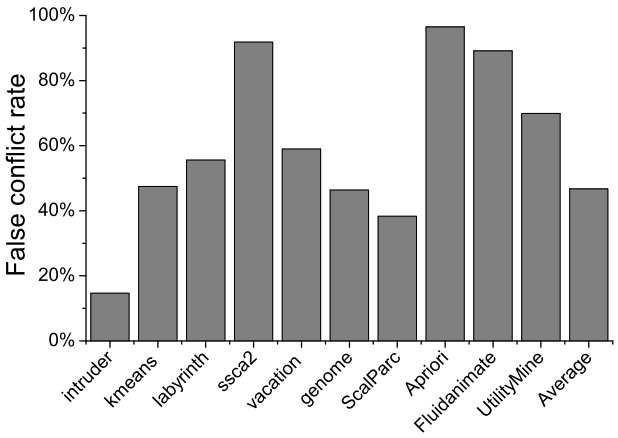

Figure 1: False conflict rate of STAMP and RMS-TM benchmark suites

As mentioned in the previous section, researchers have proposed methods to eliminate false conflicts caused by cache lines written after being read. The efficiency of their methods first rely on the fact that most false conflicts are from write-after-read (WAR) type. But our analysis shows a different observation. As shown in Figure 2, for certain benchmark programs, such as vacation and Apriori, write-after-read (WAR) false conflicts are indeed the dominant type, which makes it reasonable to ignore other types' false conflicts. Nevertheless, for other benchmark programs, *e.g.*, kmeans, labyrinth, and genome, false conflicts due to

[1] We choose not to present bayes benchmark because of its non-deterministic finishing conditions. We also exclude yada and hmm benchmarks because their transactions are extremely large and cannot fit into baseline ASF hardware.

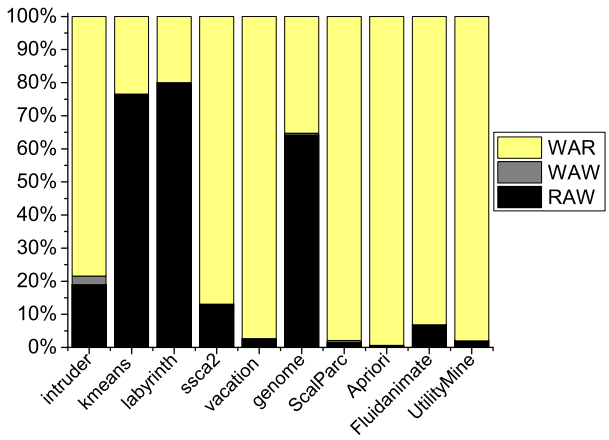

Figure 2: Breakdown of different false conflict types

read-after-write (RAW) (73% on average) are the primary conflict type. From this analysis, it is noteworthy that both WAR and RAW are important in resulting in false transactional conflicts and aborts. Hence, the solution for avoiding them should be general enough and can be applied to both types of false conflicts.

In reality, false transactional conflicts greatly depend on several factors including sharing pattern, access pattern, physical data allocation, and transaction ordering and timing for a running application. The dynamic nature of applications and its concurrent execution model makes it even more difficult to summarize general behavior; therefore, making it hard to mitigate false conflicts without runtime information. Our analysis results also support such intuition.

Figure 3 shows the time distribution of false transactional conflicts over the execution. We picked four representative benchmark programs, vacation, genome, kmeans and intruder from the STAMP benchmark suite and showed the cumulative number of false conflicts and launched transactions over time. The programs we picked are travel reservation, gene sequencing, kmeans clustering and network intrusion applications which well covered different application types. From the results, we can see that they all have similar close to linear distribution for the started transaction number over time. But for false conflicts, there are distinctive behaviors among different benchmark programs. Even within the same execution of one program, there are different phase behaviors demonstrated. For example, in the machine learning application kmeans, the number of false transactional conflicts grows quite linearly with the same trend of started transaction number. vacation also has a similar growth feature as its started transaction number. In contrast, in the gene sequencing application genome, the number of false conflicts grow more rapidly in two particular periods than the others, while the number of its started transactions is growing linearly during most of the time.

Figure 4 shows the physical address space distribution of false conflicts. We picked the same four benchmark programs as former paragraph and showed the false conflict number by physical cache line index. As shown, the results also show distinctive behaviors among benchmarks. In vacation and intruder, false conflicts have quite uniform distribution in most cache lines except few peak points. On the contrary, kmeans shows an interestingly different pattern. Here, false conflicts are mostly from a few specific cache lines. It is in accordance with the data layout of kmeans. In kmeans, most conflicts are caused by few globally shared data elements. Like time distribution of false conflicts, we can see that their address space distribution also hardly show any general spatial locality feature in L1 cache. There is no general trend in either time or space domain for false conflicts.

The results in Figure 3 and Figure 4 may have shed some light on potential possibilities to solve this problem from a programmer's viewpoint. However, from the perspective of hardware support, it is difficult to draw a common, predictable trait for false conflicts at the system level of an HTM. Given the dynamics and uncertainty due to programs' arbitrary access/execution pattern, it would be even more difficult to realize a simple, elegant solution to solve false conflict problem completely.

B. Access pattern of false conflicts inside a cache line

False conflicts are caused by the accesses of different, non-overlapped locations in the same cache line. Although their general behavior has some dynamics, the access pattern inside one cache line shows stable characteristic. Figure 5 shows the number of accesses of each location in one cache line. We picked the same four representative benchmarks for analysis. From the result shown here, we can see that accesses are distributed across a cache line at the granularity of 8 bytes in vacation, genome, intruder and 4 bytes in kmeans. This is corresponding to the four benchmarks' data structure size. In kmeans, 32 bits data granularity is used while others choose coarser data structure.

Although between benchmarks they still have differences on their exact access distribution inside a cache line, they all share a regularly scattered distribution pattern. This result indicates a great potential of reducing false conflicts with a lightweight sub-blocking technique.

In order to further demonstrate that, we also analyze the reduction rate of sub-blocking technique. Figure 8 shows the percentage of reduced false conflicts with different sub-blocks of one cache line. For the benchmarks discussed above, with 4 sub-blocks, the reduction rate can be even almost 100% in vacation and relatively good in others. While with 8 sub-blocks, all of the false conflicts are eliminated except for kmeans due to its finer data granularity. Even in kmeans, the false conflicts can be reduced significantly with only 4 sub-blocks as shown later in Figure 8. We can

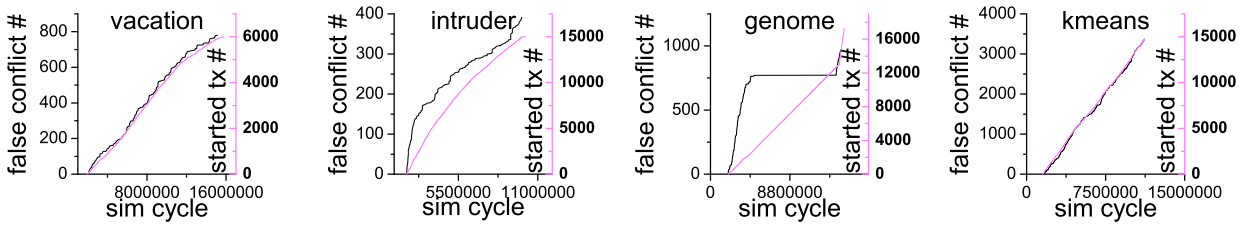

Figure 3: Cumulative number of false conflicts over execution

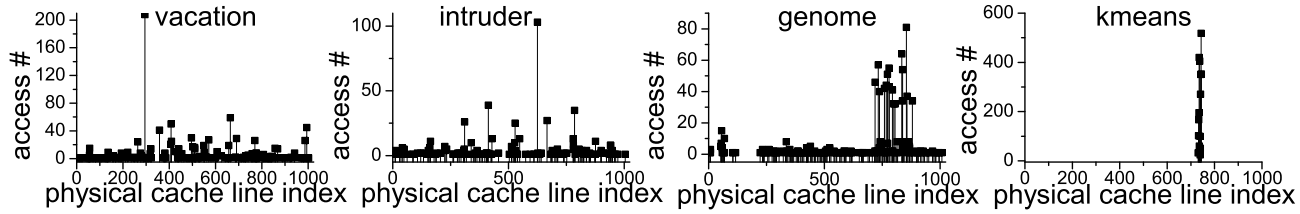

Figure 4: False conflict number by cache line index

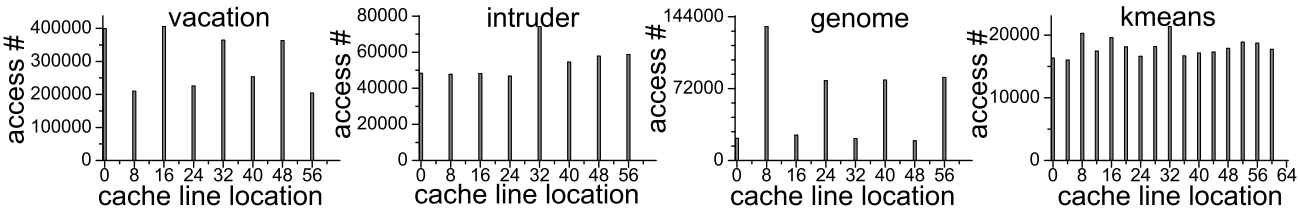

Figure 5: Number of accesses by location inside cache lines

see that a lightweight sub-blocking technique is efficient enough for most programs. Meanwhile its simplicity also well follows the same design philosophy of ASF, that is keeping the impact on existing system minimum.

This is the basic idea of our simple but very efficient and implementable technique for ASF-enabled TM systems. The implementation details will be discussed in the following section.

IV. Implementation

This section demonstrates how to apply sub-blocking to the hardware design to mitigate the negative effect of false conflicts.

A. Baseline system

First, for this study, we use the AMD's Advanced Synchronization Facility (ASF), an HTM extension for AMD64 processor, as our baseline system [10]. ASF uses the L1 data cache and load/store queue to buffer speculative data sets for lazy data versioning. To support that, each L1 data cache line was extended with two extra bits: an SW bit for speculative read and an SR bit for speculative write. ASF uses MOESI cache coherence protocol. Transactional conflicts are detected by checking the SW/SR bits against the incoming cache coherence protocol messages. An invalidation request message will conflict with both the SW and SR bits, while a non-invalidation request message will only conflict with the SW bit. If a conflict is detected, the earlier conflicting transaction will be aborted by discarding all the speculatively modified data and resetting the bits. If a transaction commits without conflicts, the buffered speculate data will be committed by gang-clearing the bits.

As one of the earliest HTM systems in modern multi-core processors from industry, ASF is a practical design. Given it was established on top of the x86 architecture, it represents the potential future trend.

B. Basic hardware mechanism

We propose to implement our lightweight technique as an extension to the baseline ASF system. Instead of checking conflicts at the granularity of a cache line, we divide the data portion of a cache line into sub-blocks. Two extra state bits are added for each sub-block: SPEC for speculative access and WR for access type as shown in Table I. Speculative accesses inside transactions will set these two bits of the corresponding sub-block according to their access types. As shown in Table I, the corresponding sub-block has never been speculatively accessed if both of these two bits are set to 0. Similarly, it has been speculatively written or read if the SPEC bit is 1. The dirty state is set through the returned coherence protocol message when current sub-block is speculatively written by other cores but has not

Table I: Sub-block state

SPEC	WR	State
0	0	Non-speculate
0	1	Dirty
1	0	Speculative Read (S-RD)
1	1	Speculative Write (S-WR)

caused any true conflict yet. It is used to detect potential conflicts and will be further discussed in the following section. Conflict detection will check the incoming messages against the corresponding sub-block's state bits. Thus, false conflicts of different sub-blocks can be completely avoided. To achieve this goal, we also need to check conflicts with both valid and invalidated cache lines for detecting conflicts of cache lines invalidated by false WAR conflicts. Besides, piggy-back bits will be added to the original coherence messages for handling dirty states.

C. Handling dirty state

Enabling sub-blocking for HTM requires careful handling of all possible states. The dirty state problem shown in Figure 6 can cause correctness issue and hence needs proper handling. As shown in Figure 6(a), a transaction T1 reads cache line B which is in T0's L1 cache. This cache line has been written by T0 earlier in a different sub-block. Since there is no true conflict, both transactions will continue. A subsequent T1 attempts to read A and produces a RAW conflict with T0. However, T0 will not get any coherence message because T1 hits its own cache. Both transactions will be allowed to commit which breaks the atomicity requirement of TM. Similarly in Figure 6(b), T1 will get an incorrect value from its subsequent read, if T0 aborts first.

In order to handle the problem shown above, we introduce the *dirty state* to each sub-block. If one sub-block is written by some other transaction but has not caused any true conflict, the speculate bits will be set to dirty state. It indicates that current sub-block is not reliable for further accesses. When the dirty sub-block is hit in subsequent memory accesses, it will be treated as a local L1 cache miss since the data may be incorrect. The current core will send out a non-invalidating coherence message to all other cores to request for the cache line. If the conflicting transaction is still ongoing, it will be aborted by this message. Otherwise, if the conflicting transaction has been aborted earlier, the cache line request will finally be fulfilled by the deeper levels of the memory hierarchy.

D. Detailed mechanism

1) Load access: When a transaction attempts to read a data item that is not in any core's L1 data cache, it proceeds, as in the ASF baseline, by issuing a cache miss request and setting the corresponding sub-block's bits (SPEC=1, WR=0

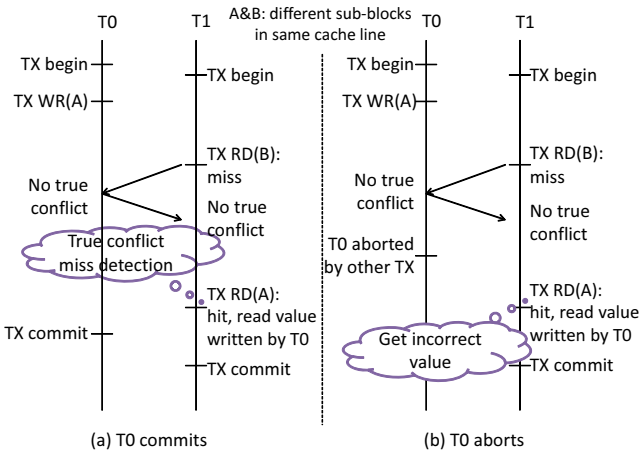

Figure 6: Problems with dirty state

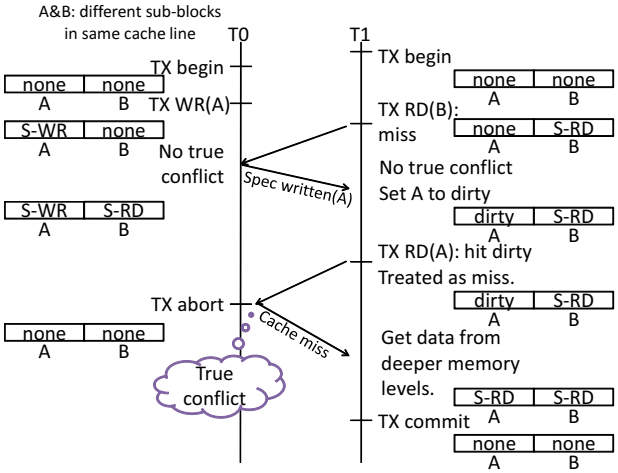

Figure 7: A detailed example of load access

shown in Table I) of the accessed data part after the line is loaded into the L1 cache.

On the other hand, if the data is in some other core's L1 cache, as shown in Figure 7, the requesting core will send out a non-invalidation message to all other cores as done by a typical cache coherence protocol. For the remote core that possesses the requested data, it will check the incoming message against the state bits of its corresponding sub-block. If there is no true conflict, it will return the data together with the special status bits indicating speculative written sub-blocks of this cache line. After receiving the return message, the requesting core will check against the piggy-back status information to see if any sub-block has been speculatively written by the remote transaction. If so, the corresponding sub-blocks will be marked as dirty (SPEC=0, WR=1 shown in Table I).

When a transactional load hits the local L1 cache, it will

check against the state bits of its sub-block to see if it is a dirty sub-block. If so, this load will be treated as a local L1 cache miss because it could miss true conflict or get incorrect data as discussed in the prior section. The local core would then send out a non-invalidating message to other cores for data requesting as done by a typical cache coherence protocol. If the remote transaction is still ongoing, it will be aborted by this coherence message. Otherwise, correct data will be fetched from either other cores' L1 cache or from the lower level memory. After this cache coherence message is returned, the requesting core clears the dirty state of this sub-block by setting the SPEC bit to 1 and the WR bit to 0.

2) Store access: When a transaction issues a speculative store, it sends out an invalidation message as done by a cache coherence protocol. The remote cores will then check the message against their corresponding sub-block's state bits for conflict detection. If there is a conflict, the transactions in the remote cores will be aborted based on the conflict resolution policy of the ASF-enabled system.

If the cache line has been speculative read by the remote core and there is no true conflict, the remote core can continue its transaction but the cache line with speculative information will be invalidated. In order to avoid miss conflict detection of cache lines invalidated by false WAR conflicts, all the speculative information will still stay inside the invalidated cache line. The later conflict detection will be decoupled with general cache coherence states. The conflict check will be done for both valid and invalidated cache lines.

If there is no true conflict but the cache line has been speculative written earlier by the remote core, the remote core also needs to abort its running transaction to avoid speculatively updated data getting lost in the invalidation process. As shown in Figure 2, false conflicts caused by write-after-write accesses consist nearly 0% in total. Thus, ignoring false conflicts due to write-after-write type will not lead to any considerable performance loss.

3) Commit and abort: During the period of commit or abort, the procedure is exactly the same as the baseline ASF system. The only difference is that a clear action needs to clear more bits than just two. The dirty state bits in other cores caused by the current transaction do not need special reset. When remote transactions commit, abort or hit the corresponding sub-blocks with dirty state, these dirty state bits will be cleared naturally. In order to keep the commit procedure as simple as the baseline, we leave the dirty state bits of remote cores untouched at the commit time.

E. Overhead of Proposed Implementation

The performance overhead of our method compared to the baseline ASF comes from two parts. The first part is the extra state bits check and clear. It can be minimized by proper hardware design. The second part is the slightly larger coherence messages. A few piggy-back bits will be added to the original coherence messages for transactional loads. Returning messages of a load request may need to send special status bits together with the data. This could consume extra time due to the larger data size. For a typical configuration of four sub-blocks, the extra number of status bits is four. Compared to the 64-byte cache line size, the extra data transmission time will be almost negligible. Moreover, it only happens when a load request hits other cores' written cache line. Such situation was originally considered as a conflict and could trigger abort in the baseline ASF system. Thus, the overhead of performance is considerably small and can be ignored comparing to the performance gain of our technique.

The hardware overhead mainly comes from the extra state bits for each sub-block inside each data cache line. For N sub-blocks, the extra bits can be 2N bits in each cache line. Compared to baseline ASF's design, the overhead is 2(N-1) bits per cache line. For a typical cache configuration, an ASF system has a 64KB L1 cache with a cache line size of 64 bytes. If sub-dividing the cache line into four sub-blocks, the hardware overhead compared to the baseline ASF will be 0.75KB, accounting for 1.17% of the original L1 cache size.

V. Evaluation

A. Simulation Methodology

To evaluate the effectiveness of our technique, we use PTLsim [18]. It supports full system simulation and features a detailed timing model of an out-of-order processor core. AMD provides extension to the baseline PTLsim to support their ASF mechanism [18]. The simulator was configured to match the generic configuration of AMD Opteron processors, with a three-wide clustered core, out-of-order issuing, and instruction latencies modeled after AMD Opteron microprocessor. The detailed configuration is shown in Table II. We use the original ASF design as the baseline system for comparison with our proposed speculative sub-blocking state technique. We also modelled a perfect system with no false conflict to demonstrate the effectiveness of our scheme. Although our method does not have any special restriction on the number of sub-blocks within a cache line, we chose four sub-blocks as the configuration parameter for all the performance evaluation. It is based on the results of our sensitivity study, which will be discussed in following section.

The STAMP [14] and RMS-TM [15] benchmark suites were used for our evaluation. We used the standard configuration for both of them. In our simulation, we excluded bayes due much to its non-deterministic finishing conditions and also yada and hmm for their extremely large transactions. The details of the remaining benchmarks are shown in Table III. They include machine learning algorithms like kmeans and scalparc, data mining applications like apriori and utilitymine, graph computing like ssca2 and other parallel applications. Instead of using the TM compiler

Table II: Simulation configuration

Feature	Description
Processors	8 AMD Opteron 2.2GHz Out-of-Order cores
L1 DCache	64KB, 64B cache line size, virtually indexed, 2-way set associative, 3 cycles load-to-use latency
Private L2 cache	512KB, physically indexed, 16-way set associative, 15 cycles load-to-use latency
Private L3 cache	2MB, physically indexed, 16-way set associative, 50 cycles load-to-use latency
Main memory	2048MB, 210 cycles load-to-use latency
D-TLB	48 L1 entries, fully associative, 512 L2 entries, 4-way set associative

Table III: Benchmark description

Benchmark	Description
intruder	network intrusion detection
kmeans	K-means clustering
labyrinth	maze routing
ssca2	graph kernels
vacation	client/server travel reservation system
genome	gene sequencing
scalparc	decision tree classification
apriori	association rule mining
fluidanimate	fluid simulation
utilitymine	association rule mining

provided in [18], we chose to rely on normal gcc compiler and put all TM-related ASF instructions in the library. This enabled us to avoid impact from potential unknown TM compiler optimizations and focus on our work. In order to avoid live locks, we also introduced a simple exponential backoff manager in the software library, which exponentially increases the backoff time according to transaction retry times.

B. Experimental Results

We present our simulation results in this section. First, we evaluate the effect of different configurations of our technique. In our mechanism, the number of sub-block to be used in a cache line is the most important parameter. With smaller sub-blocks, *e.g.*, finer-grained sub-blocks, we can reduce more false conflicts. Nonetheless, it also introduces more hardware overhead for more status bits to be added for keeping the speculative sub-blocking state. To choose a proper size of a sub-block, we need to make trade-off between the hardware overhead and the likelihood of false conflict reduction. To study the effect of the size of sub-blocks, we compared their respective false conflict reduction rate as shown in Figure 8.

As shown, for all benchmark programs, sub-block 16 (*i.e.*, 16 sub-blocks, each 4-byte large, in a 64-byte cache line) can completely eliminate all the false conflicts. Sub-block 8 also shows close to 100% reduction of false conflicts in most of the benchmark programs except for kmeans, which still consist of false sharing within an 8-byte sub-

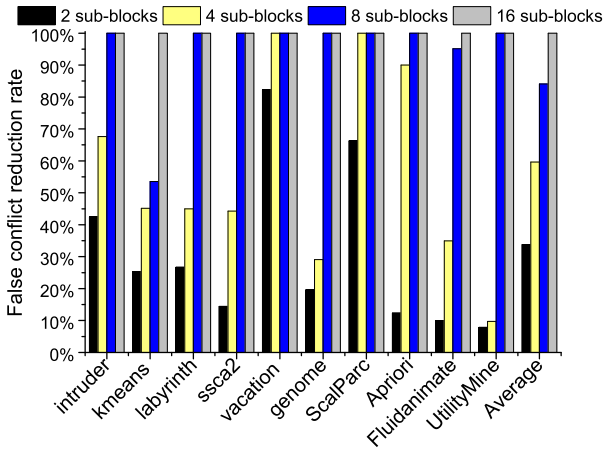

Figure 8: False conflict reduction rate of different configurations

block. When we further enlarge the size of the sub-block, four sub-blocks shows close to the perfect reduction rate in vacation, ScalParc and Apriori, and a relatively good false conflict reduction rate for others. Thus, the reasonable sub-block number should be chosen between 4 sub-blocks and 8 sub-blocks according to specific design considerations. At the end, we chose four sub-blocks (*i.e.*, 16-byte sub-blocks in L1 data cache) for our evaluation to strike the balance between hardware overhead and performance improvement potential. As we will show in Figure 9, although eight sub-blocks improve the false conflict reduction rate on several benchmark applications, four sub-blocks show close to perfect overall performance and is good enough for the overall conflict and execution time improvement.

Hence, we presented the results based on four sub-blocks in a cache line and compared its performance with two systems: the baseline ASF-enabled system and a perfect system with zero false conflict serving as the performance upper bound.

From the results in Figure 8, we can see that with four sub-blocks, our method does well on average. The reduction rate is very high, close to 100%, for vacation, ScalParc and Apriori. This indicates that the use of a 16-byte sub-blocking strategy is sufficient to differentiate true data sharing and false data sharing. However, some benchmark program, *e.g.*, UtilityMine also shows a very low reduction rate. We found that it is primarily due to the special characteristics of this benchmark program. In its transactions, several very fine-grained data structures were used. Thus, false sharing is still present among these much fine-grained data structures with our experimented sub-block with the granularity of 16-byte. In the sensitivity study result presented earlier, we can see that with smaller sub-blocks, the reduction rate of this benchmark can be dramatically improved.

Figure 9 shows the impact of our method on the number

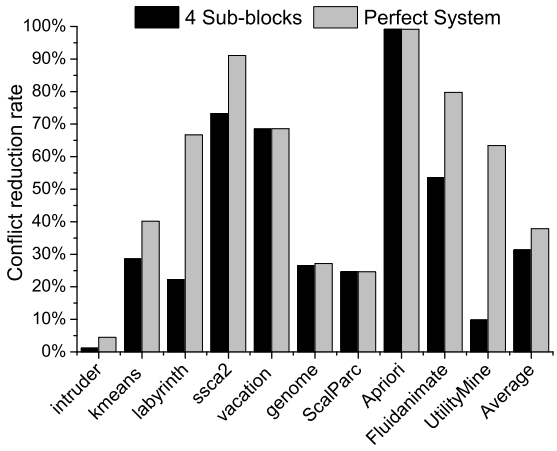

Figure 9: Percentage of overall conflict reduction

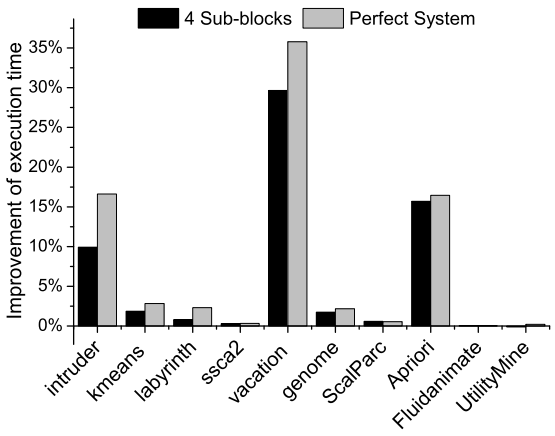

Figure 10: Improvement of overall execution time

of the overall conflicts including both true data conflicts and false conflicts. The perfect system here is the ASF system, which was configured to eliminate all the false conflicts to be used as an ideal scenario. It represents the upper bound of all false conflict optimization methods. Compared to the perfect system, our technique shows that on average we can achieve around 83% of perfect system's reduction rate with several benchmark programs approaching the perfect cases. On the other hand, we also observed that there are a few outliers such as intruder, UtilityMine, and labyrinth. Benchmark intruder has the lowest false conflict rate as shown in Figure 1, which makes the impact of our technique on overall conflicts relatively low. For UtilityMine, it is due to its low false conflict reduction rate. For labyrinth, it is because of the variance of simulation results. Most of labyrinth's aborts came from the user's aborts. The absolute number of its conflicts is sometimes even lower than 20, which makes the percentage result have a large variance. In general, our lightweight technique can eliminate a large portion of false conflicts caused by false sharing, leaving only true data access conflicts like a perfect system in several benchmark programs we evaluated.

Figure 10 shows the performance analysis of our technique. We evaluated the improvement on execution time over the baseline ASF system. We also showed the performance improvement of the perfect system as a performance upper bound.

The results show that our technique has positive impact on almost all benchmark programs except UtitlityMine, which has minor slowdown of 0.1%. This can be considered as a kind of simulation variance because of its very low false conflict reduction rate and extremely low contention rate. For benchmarks with long non-transactional execution time, the improvement is relatively small. In this result, benchmark intruder shows significant improvement of execution time while its reduction rate of overall conflicts is quite low. This is because it has very high average retry times, which makes the performance gain from eliminating conflicts much more pronounced. Similarly, benchmark vacation also received large improvement on the execution time for the same reason. Generally, for benchmark programs with good performance potential shown in a perfect system (*e.g.*, intruder, vacation and Apriori), our technique demonstrates significant improvement as high as 30%. For other benchmarks with long non-transactional execution time, the improvement will be less significant. But our technique also shows improvement approaching the theoretical upper bounds in most of the cases.

Overall, our simple, lightweight technique can effectively enhance an HTM system to recover the performance degradation caused by false transactional conflicts while the implementation cost is low and highly feasible.

VI. Conclusion

Transactional conflicts represent one of the most critical issues in achieving high performance for a hardware transactional memory (HTM) system. Transaction aborts due to conflicts not only lose useful work but also waste power. Given HTM support were announced by several key microprocessor vendors, to minimize the performance impact of transactional conflicts become imperative to attract more programmers to adopt this lock-free programming model.

To detect transactional conflicts in an HTM system, it is within the best interests of architects to leverage and reuse existing hardware to minimize the design changes. For example, in the AMD's ASF HTM system, conflict detection is done by using the inherent cache coherence protocol messages. Such mechanism simplifies the hardware changes for supporting HTM but also leads to false transactional conflict problems due to the fact that communication is performed at the granularity of cache lines.

In this paper, we analyzed the behavior of false transactional conflicts using AMD's ASF HTM system as a case study. We ported two benchmark suites using the ASF instruction extension and performed analytical study to understand their impact and explore viable solutions. In our results, false conflicts comprise a significant portion of the overall conflicts and show unpredictable patterns on time distribution and cache space distribution. However, within each cache line, their access patterns are quite stable. Base on our analytical results, we proposed a false conflict reduction technique by keeping the speculative state at the sub-block level. It maintains conflict detection at the granularity of sub-blocks while keeping the original cache coherence protocol intact. Our solution is low cost and easy to implement on top of an HTM system without much modification. Our evaluation results showed that the proposed technique can effectively reduce a major portion of the false transactional conflicts to approaching a perfect system. The performance improvement can reach up to 30% depending on the conflict behavior of applications.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[2] N. Shavit and D. Touitou, "Software Transactional Memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, February 2005.

[4] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[5] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *Proceedings of the 12th International Conference on High Performance Computer Architecture*, February 2006.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid Transactional Memory," in *Proceedings of the 12th International Conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XII, 2006, pp. 336–346.

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2009.

[8] R. A. Haring, "The IBM Blue Gene/Q Compute chip+SIMD floating-point unit," in *Proceedings of the 23rd Symposium on High-Performance Chips*, August 2011.

[9] A. Kleen, "Adding lock elision to linux," Linux Plumbers Conference, August 2012. http://halobates.de/adding-lock-elision-to-linux.pdf.

[10] J. Chung, L. Yen, S. Diestelhors, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman, "Asf: Amd64 extension for lock-free data structures and transactional memory," in *Proceedings of the 43rd Annual International Symposium on Microarchitecture*, December 2010.

[11] R. M. Yoo and H.-H. S. Lee, "Adaptive Transaction Scheduling for Transactional Memory Systems," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, ser. SPAA '08. New York, NY, USA: ACM, 2008.

[12] G. Blake, R. Dreslinski, and T. Mudge, "Proactive Transaction Scheduling for Contention Management," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, dec. 2009.

[13] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance," in *In Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, September 1993.

[14] C. M. Chi, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[15] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero, "Rms-tm: a comprehensive benchmark suite for transactional memory systems," in *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering*. ACM, March 2011.

[16] L. Porter, B. Choi, and D. M. Tullsen, "Mapping out a path from hardware transactional memory to speculative multithreading," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, September 2009.

[17] F. Tabba, A. W. Hay, and J. R. Goodman, "Transactional conflict decoupling and value prediction," in *Proceedings of the 11th International Conference on Supercomputing*, June 2011.

[18] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière, "Evaluation of amd's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010.

[19] J. Huh, J. Chang, D. Burger, and G. S. Sohi, "Coherence decoupling: making use of incoherence," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XI. New York, NY, USA: ACM, 2004.