Using A Runtime to Overcome The Pathologies in Hardware Transactional Memory Systems

Zhichao Yan, Dan Feng, Yujuan Tan
Wuhan National Laboratory for Optoelectronics
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, China
Email: zhichao_yan@smail.hust.edu.cn, dfeng@mail.hust.edu.cn, tanyujuan@gmail.com

***Abstract*—As one of the most potential solution to improve thread level parallelism and reduce most ordinary programmers' burden on parallel programming, transactional memory (TM) systems have attracted a great deal of attention from both industry and academic since the notion was proposed in 1993. Since then, various designs and implementations are proposed to improve the performance while reducing the overheads. However, recent investigations of the high-contention and coarse-grained workloads reveal various pathologies that will offset the performance benefits. In this paper, we analysis the advantages and disadvantages of existing conflict management and version management schemes, make a case study in the interplay of them to learn its impact on performance. In particular, we apply a runtime environment to recognize application's dynamic behaviors and resolve transactional conflicts to make up the gap between the upper layer application's diversity and the underline hardware's capability. Throughout the comprehensive evaluation, we find that our proposal can obtain a significant performance improvement across the applications selected from the STAMP benchmark suite on DynTM, which is regarded as one of the latest progress in HTM systems.**

I. Introduction

Nowadays, the processor design and manufactory technology has transformed from frequency scale to core scale since 2005 when the traditional method hit the power-thermal wall. Although chip multiprocessor (CMP) is the reality norm, most software can't exploit the performance benefits and how to keep the software performance up with the current and future CMP processor becomes one of the most urgent problems in the whole computer community. Existing methods, such as various locking mechanisms, will waste the programmers a lot of time to cope with multiple threads competing for the shared data in a shared memory environment. To alleviate the burden on programmers while improving the thread level parallelism of the multi-threaded program, researchers proposed to incorporate the transaction notion from database into parallel programming, which was called transactional memory (TM) system [10]. Due to the performance and programmability advantages (i.e., optimistic execution, natural composition, fault tolerance, no priority inversion, etc.) over the traditional lock mechanism under the multi-threaded environment, transactional memory has received a lot of attention from both the academia and industry, thus becoming a potential solution of the long-standing problem in parallel programming on current multi-core and future many-core platforms [9]. Moreover, along with a lot of comprehensive studies on TM systems, several leading technology industry companies, such as Sun Microsystems [5], Azul [6], IBM [8], AMD [1] and Intel [11], have already designed CMP processors with hardware transactional memory (HTM) support.

TM systems borrowed the transaction notion from the database community to coordinate concurrent access to the shared data in the multi-threaded applications on the shared-memory environment, which organized the program's possible racing blocks into transactions that would atomically execute in isolation, meaning that the transactional modifications will perform totally on commit or not at all on abort and the surrounding threads are not aware of any transactional modifications until the transaction commits its work. Any memory access violating the isolation requirement is regarded as a transactional conflict that must be detected and resolved properly. In order to maintain atomicity and isolation, besides taking a checkpoint of the processor's state at the very beginning of each transaction, TMs must maintain both the old and new data values upon each transactional write operation during the execution until the end of the transaction for abort and commit respectively. Since the first TM [10] was proposed in 1993, TMs have developed in a variety of directions, including: software TM (STM), hardware TM (HTM), and a combination of the two (Hybrid TM). In order to implement a highly efficient TM, designers must take care to deal with conflict management and version management, where the former determines when to detect and how to solve the transactional conflicts and the latter handles on how to store and when to merge the speculative modifications during the transaction execution [9]. In particular, this paper makes a study on the interplay between conflict management and version management, finds the necessity of considering the dynamic behaviors of concurrently running transactions during their runtime and proposes a runtime support

This paper is based on "A Case Study of The Interplay between Conflict Management and Version Management in Hardware Transactional Memory Systems," by Z. Yan, D. Feng, and Y. Tan, which appeared in the Proceedings of the 4th International Symposium on Parallel Architecture, Algorithms and Programming (PAAP), Tianjin, China, Dec 2011. © 2011 IEEE.

doi:10.4304/jcp.8.1.18-26

in hardware transactional memory systems to overcome the possible pathologies under the high-contention and coarse-grained workloads, moreover, we further to reduce the long-length transaction's overheads by incorporating the thread level speculation optimization on the runtime augmented HTM scheme.

Early studies chose the low-contention and fine-grained handcrafted applications to evaluate TM's performance, which hid the side effects from the conflict management and version management on performance. As more investigations on the high-contention and coarse-grained applications are made by the transactional memory community, more performance pathologies are exposed that may offset TM's performance benefits. When we made a case study on existing TMs on the high-contention and coarse-grained applications, we found several pathologies stem from the interplay between the conflict management and version management. For example, the time overheads spent on version management schemes not only lengthen the transaction's execution period but also result in more transactional conflicts that leads to a vicious circle, which will degrade the performance dramatically. Meanwhile, how to manage the transactional speculative versions during the conflict occurrence and resolution will seriously impact the time spent on the version management that may also impact the surrounding transactions trying to access the shared data in this transaction. So the interplay between conflict management and version management will be a key to guide the optimization on future transactional memory systems. Existing research already found that there was not a known conflict resolution policy that can perform universally well under various environments [19]. We argue that transactional memory systems need a runtime environment to make up the gap between the various transactional applications and the underlying support in hardware. In this paper, we try to dissect the DynTM [13] scheme, one of the latest progress in hardware transactional memory systems, and find some interesting appearance that can guide the future HTM design [22]. Especially, we add a software runtime to recognize the dynamic conflicting behaviors of the running transactions and guide the conflict resolution that can significantly boost the performance. After evaluation, we find that our new method obtains an average speedup of 11.7% under the workloads selected from the STAMP benchmark suite over the DynTM proposal. To further reduce the transactional overheads on the high-contention, coarse-grained and long-length transactions, we have incorporated the thread level speculation optimization in the runtime augmented DynTM scheme and found this optimization can achieve an average speedup of 42.9% over the original scheme throughout the 3 high-contention, coarse-grained and long-length transactional workloads such as bayes, labyrinth and yada.

In what follows, we will first introduce necessary background knowledge on conflict management and version management in Section II, then to present our case study on the interplay between conflict management and version management and propose an improving DynTM with software runtime support to guide the transaction schedule and conflict resolution in Section III, at last we make a conclusion in Section IV.

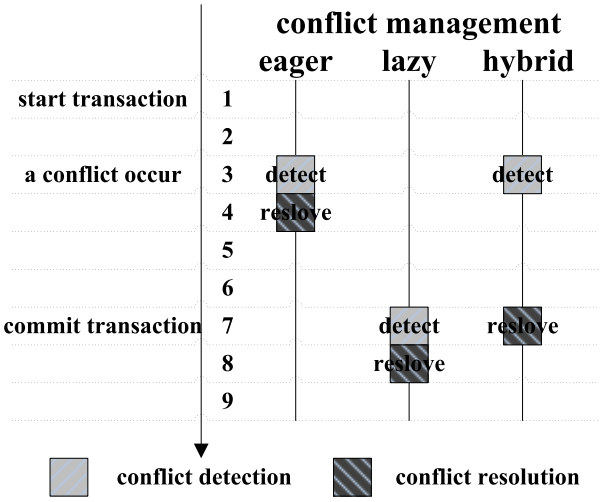

Fig. 1: The Categories of Conflict Management

II. Background

In this section, we will firstly introduce necessary background knowledge about the conflict management and version management in HTM systems, then present two observed pathologies stemmed from the interplay between conflict management and version management to motivate our research.

Conflict management decides when to detect transactional conflicts and how to solve the conflicts. Usually, transactional conflicts can be either detected upon each memory access eagerly [17], [24] or checked at the commit stage lazily [7], [18]. Designers always extend the cache coherence protocol to detect transactional conflict, which doesn't incur much overhead on detecting the conflicting access operations. Earlier designs coupled conflict resolution with the time when the transactional conflict is detected, recent designs propose to decouple conflict resolution from conflict detection that can detect conflicts eagerly but may solve the conflicts in either eager or lazy mode, where the former solves the conflict when it is detected and the latter defers the conflict resolution on commit [21], [20].

In order to depict the conflict management clearly, we draw the conflict management schemes in Figure 1, which include eager, lazy and hybrid modes to manage transactional conflict. The eager scheme detects and resolves transactional conflicts eagerly, which can avoid wasting computation on the conflicting transactions but may be a pessimistic method that blocks the thread level parallelism. The lazy scheme, in the opposite, defers both conflict detection and resolution on commit to exploit more thread level parallelism but will waste more computation time on a transaction that may be aborted due to transactional conflicts. To overcome the disadvantages and combine the advantages of these schemes, researchers propose the hybrid scheme [21], which will execute well

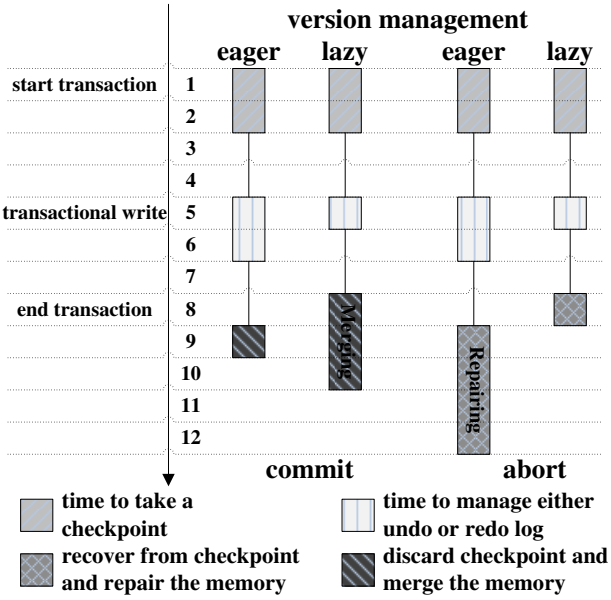

Fig. 2: The Overheads of Version Management Schemes

in the fast path and suffer from existing pathologies in the slow path.

On the other side, existing version management schemes are generally divided into the eager and lazy categories, where the former directly updates the new values "in place" after storing the old values in an undo log that will trap into the software library to restore the old values prior to the transaction on abort [24], and the latter buffers the new values in a redo log that will merge them with the safe memory on commit and discard them on abort [7]. Usually, there are two different types of time overheads incurred by the unexpected transactions during their execution, namely repairing time and merging time, corresponding to the eager and lazy schemes for abort and commit respectively, that spent on transferring the consistency data among different memory hierarchies. As a result, the eager schemes make the expected commit operation fast at the cost of trapping into the software to restore the old values on abort, and the lazy schemes are more suitable to handle the high-contention applications, which will result in a lot of abort operations, but they will dramatically degrade the performance if the buffer overflows during the transaction execution, thus incurring the thrashing pathology [4]. Besides the eager and lazy version management schemes, Lupon et.al proposed the FasTM [12] scheme that adopted the lazy scheme when the speculative data fitted in the first level data cache and degenerated to the eager scheme when transactional data overflowed from the first level data cache via exploiting the inconsistency among the different memory hierarchies. So this hybrid scheme may also suffer from the repairing and merging overheads as what the eager and lazy schemes do. A. Armejach et al. [2] propose to use a reconfigurable L1 cache to reduce the data movement overheads, which requires more silicon area for the reconfigurable cache that is dedicated for transactional access operations exclusively.

In Figure 2, we approximately describe the overheads incurred by the existing version management schemes, where the eager scheme performs well on commit while the lazy scheme works well on abort. However, the extra time overheads on repairing (i.e., time slot of 9, 10, 11 and 12 in the eager scheme on abort) or merging (i.e., time slot of 8, 9 and 10 in the lazy scheme on commit) the memory not only lengthen transaction's execution period by themselves but also lead to more conflicts with the surrounding transactions attempting to access the shared data that will waste the computations by aborting the conflicting transactions or stalling the execution till the conflict is solved. Moreover, these overheads may lead to a vicious cycle, thus blocking the thread level parallelism, and this situation will especially worsen in the high-contention and coarse-grained applications that represent the characteristics of future multi-threaded programs written by most ordinary programmers.

Lupon and et.al [13] found the complementarity characteristics between the eager and lazy transaction schemes, so they proposed to combine the advantages of these two approaches and avoid their disadvantages via a dynamic transactional mode selector that could select the proper scheme during the execution that can simultaneously support both eager conflict detection and eager version management and lazy conflict detection and lazy version management. In this paper, we revisit this DynTM scheme, try to make a case study to learn the interplays between conflict management and version management, find some interesting phenomena that can guide the future HTM design and finally propose an enhanced DynTM with a runtime support to overcome the possible pathologies.

In Figure 3, we list two typical pathologies, repair pathology and merge pathology, stem from the interplay between the conflict management and version management. For example, as shown in the left part of Figure 3, transaction 1 was aborted by the conflict between transaction 1 and transaction 2, and during the repair period in the transaction 1, transaction 3 was aborted due to the conflict with transaction 1. So any surrounding transactions trying to access the repairing data would be aborted. This case is common in the high-contention and coarse-grained transactional applications and the repair pathology may lead to a vicious circle that degrades the whole performance significantly. On the other hand, the overheads spent on merging the speculative data on commit in the lazy mode also induced the merge pathology. As shown in the right part of Figure 3, transaction 3 was firstly aborted due to the conflict between transaction 2 and transaction 3, and then transaction 2 was also aborted due to the conflict between transaction 1 and transaction 2, this case may happen in the coarse-grained transactions with a lot of modifications that lengthened the merge period on commit. Both repair and merge pathologies happen in high-contention and coarse-grained applications that hide in the interplay between conflict management and version management. When these aborted transactions restart later, the same problem may happen again. In this

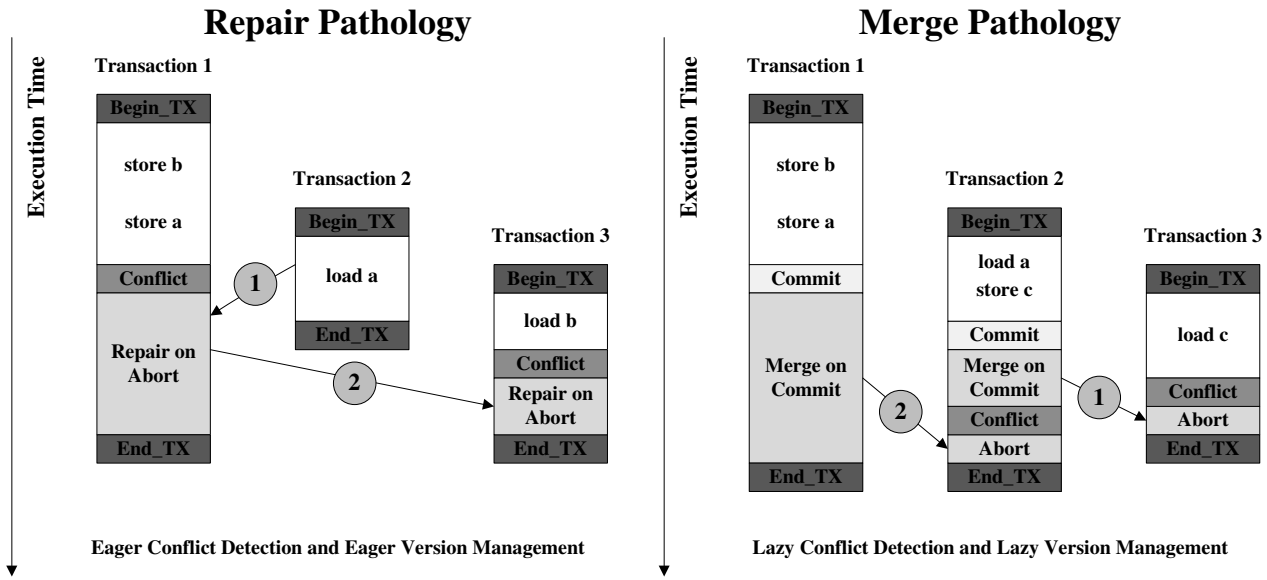

Fig. 3: The Pathologies on Repairing and Merging

paper, we find the simple hardware history-based selector in DynTM cannot handle the complex behaviors of the transactional applications and propose a software runtime [3] to recognize the dynamic conflicting behaviors of the running transactions and guide the conflict resolution to overcome the repair and merge pathologies.

III. A Case Study and Improvement on DynTM

In this section, we will first make a case study on representative transactional memory systems, especially on the DynTM scheme, which is regarded as one of the latest progress in hardware transactional memory systems, then we integrate our runtime environment in the DynTM scheme and evaluate the performance gain over the original scheme, and finally we propose a optimization on the runtime to further reduce the transactional overheads especially under the high-contention, coarse-grained and long-length transactional workloads.

A. A Case Study on Representative HTMs

In order to quantify the effects of various hardware transactional memory systems, we have simulated them on GEMS 2.1 [15], a popular simulator based on the Simics [14] platform. The experimental environment contains a 16-core CMP with necessary hardware components to support HTM functions. Table I lists the basic parameters of the simulated CMP system and Table II summaries the workloads selected from the STAMP benchmark suite [16] that is designed to evaluate the coarse-grained and high-contention workloads without a lot of handicraft optimizations to represent the program written by the ordinary programmers in future. As shown in Table II, we choose kmeans and ssca2 to represent the low-contention and fine-grained application, vacation to represent the low-contention and coarse-grained application, intruder to represent the high-contention and fine-grained application, and bayes, genome, labyrinth and yada to represent the high-contention and coarse-grained application. These

TABLE I: Configuration of The Simulated CMP System

Processor Core	1.2 GHz in-order, single issue
L1 Cache	32 KB 4-way, 64-byte line, write-back, 1-cycle latency
L2 Cache	8 MB 8-way, write-back, 15-cycle latency
Main Memory	4 GB, 4 banks, 150-cycle latency
L2 Directory	Bit vector of sharers, 6-cycle latency
Interconnect	Mesh, 2-cycle wire latency, 1-cycle route latency
Signature	2 Kbit Bloom filters

TABLE II: Workloads Selected from The STAMP Benchmark Suite

	Input Parameters	Contention	Grain
bayes	32 vars, 1024 records	High	Coarse
genome	16k segs, 256 gene, 16 length	High	Coarse
intruder	10 attacks, 4 length, 2k flow	High	Fine
kmeans	40/40 clusters, 2048 rows	Low	Fine
labyrinth	32x32x3 maze, 64 routes	High	Coarse
ssca2	8k nodes, 3 edges, 3 length	Low	Fine
vacation	4 queries, 4k transactions, 16k relations	Low	Coarse
yada	20 angle, 633.2 input mesh	High	Coarse

applications represent a wide spectrum characteristic of the transactional workloads with more attention focus on the high-contention and coarse-grained application that is the trend of transactional workloads in future.

Before present the experimental results, let's briefly review the LogTM-SE, TCC and DynTM schemes. LogTM-SE updates the new value in place instantly after recording the old value in its thread's private undo log while it needs to trap into the software library to restore the aborting transaction's old values in a first-in-last-out order along with the undo log. TCC holds the uncommitted new values in its capacity-limited hardware buffer while it merges them with the safety memory via the extended write back protocol on commit if its buffer can hold the uncommitted modifications, otherwise it will allocate a new space in memory to hold the overflowed uncommitted modifications that will incur a lot of time to transfer these

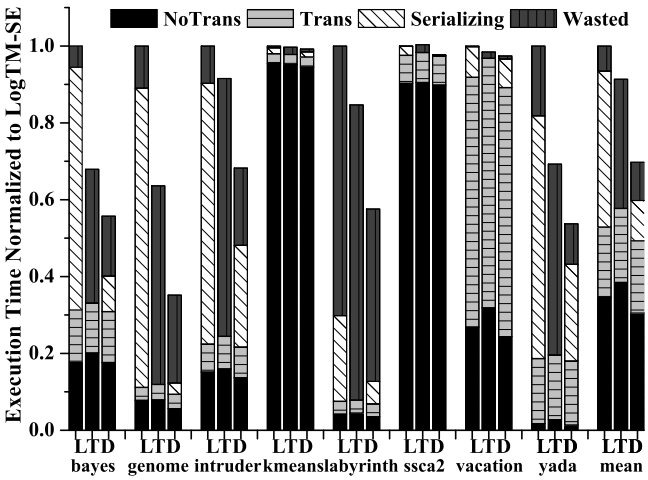

Fig. 4: The Execution Times of 8 Applications Selected from The STAMP Benchmark Suite on 3 Representative HTMs, Where L, T and D Represent The LogTM-SE, TCC and DynTM Schemes Respectively and All Their Execution Times Are Normalized to That on The LogTM-SE Scheme

data. DynTM uses a transactional mode selector (i.e., a hardware-based history predictor) to predict the possible result at the beginning of each transaction to choose the eager or lazy scheme, where the eager scheme also detects the transactional conflicts eagerly and the lazy scheme defers its conflict detection on commit stage. Due to the L1 data cache's capacity limit, DynTM will degenerate to the eager scheme if the transactional data overflows the data cache.

Figure 4 presents the breakdown execution time of 8 applications selected from the STAMP benchmark suite on the current representative HTMs. In order to get comprehensive understanding about the execution overheads, we divided the execution time into these 4 categories, the $NoTrans$ and $Trans$ components represent the time spent on non-transactional and transactional computations respectively, the $Serializing$ component represents the time spent on serializing the conflicting transactions when the transactional conflicts are detected and the $Wasted$ component represents the time spent on the aborted transactions. From Figure 4, the hybrid scheme (i.e., DynTM) achieves the best performance among these 3 schemes. At the same time, we find that the eager version management scheme (i.e., LogTM-SE) costs a lot of time on serializing the conflicting transactions because it must acquire the exclusive permission before the in-place update that requires the eager version management scheme couples with the eager conflict detection. The lazy version management scheme (i.e., TCC) defers resolving the transactional conflicts on commit that may exploit more thread-level parallelism but it will waste more time spent on the aborted transactions. In general, DynTM can reduce the repair pathology in the eager mode and the merge pathology in the lazy mode when the prediction is correct, and it may also suffer from the known pathologies on mis-prediction. To better study the DynTM scheme, we make a case study on the prediction accuracy about its transaction selector to learn the impact on performance in the following subsection.

B. A Case Study on DynTM

Though DynTM achieves the best performance among existing schemes, we would like to learn how sensitive the performance of such scheme is to the predictor's prediction accuracy. We analysis the collected execution traces and find that some transactions can benefits from the eager scheme, some from the lazy scheme and some are independent to these schemes. We model a simple predictor instead of the execution mode selector to choose either the eager or lazy scheme with a preset prediction accuracy when replaying the traces to make the sensitivity study. For example, 80% prediction accuracy means that 80% of the total transactions either choose the proper scheme or the transaction is indifferent to the choice, and the remaining 20% transactions choose the opposite scheme. Figure 5 presents the execution time speedup of a 16-core processor over a single-core processor with various prediction accuracies under the DynTM scheme, from which we find that the application's performance is sensitive with the prediction accuracy especially under the high-contention and coarse-grained application such as $bayes$, $genome$, $labyrinth$ and $yada$. The DynTM scheme is difficult to optimize the coarse-grained transactions that may overflow from the L1 data cache, and this situation will worsen on high-contention applications that may suffer from trapping into the software library to restore the old values prior to the transaction in the undo log on abort as the eager scheme does. Moreover, not every transaction presents even impact on the performance that imposes a challenge to the predictor to recognize the high priority transactions and assign them the proper running scheme, the various transactions with different length and contention further adding the difficult to this problem. At last, we can see the 100% prediction accuracy obtains significant performance improvement, but the performance benefits decline rapidly as the prediction accuracy decreases, thus meaning this scheme is heavily depended on a high accuracy predictor, which is hard to implement considering the current dynamic branch predictor in contemporary superscalar processor.

From the above sensitive study, though the DynTM scheme provides a flexible framework to enable both the eager and lazy transactions to execute concurrently, we have observed several limitations in this scheme. When we further analysis the overheads incurred by various version management schemes, most overheads lie in the overheads incurred by the interplay between conflict management and version management. To overcome these side effects, we propose a software runtime environment to recognize the dynamic conflicting behaviors of the running transactions and guide the conflict resolution to assist the DynTM scheme to reduce the possible pathologies.

C. Runtime Environment on DynTM

To make the underlying hardware component aware the upper transaction's behavior, we construct several software structures and collect transactional information to shepherd the transaction schedule and conflict resolution.

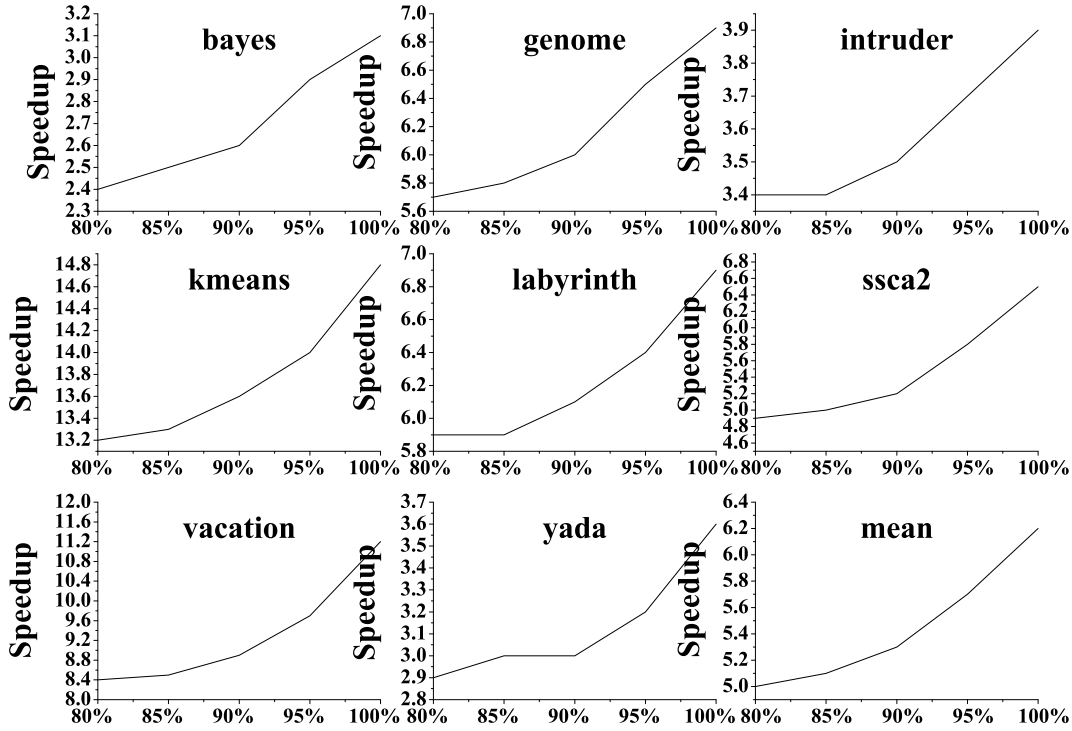

Fig. 5: The Execution Time Speedup of A 16-Core DynTM Scheme with Various Prediction Accuracies vs A Single-Core Machine

There is a simple transaction running table to indicate which transaction runs on which core. During the execution, our runtime collects the conflicted addresses and organizes these addresses into several sets that represent the minimum conflict unit to predict the possible conflict. As shown in Figure 6, the static transactions labeled in the source code can presents a lot of valuable information to schedule its execution, currently we also collect these dynamic information, such as transactional read/write set, static transaction length and conflict level. At the same time, it dynamically updates the length of executing transaction's length to indicate how much work is left to run. In what follows, we will use an example to introduce the work scheme of the runtime environment.

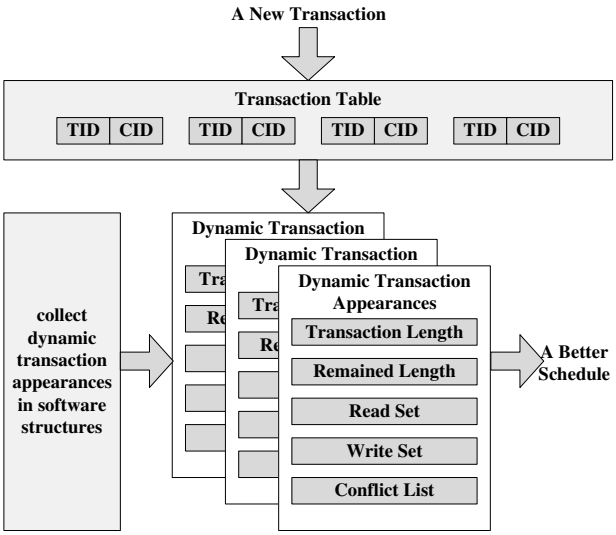

Fig. 6: Software Structure of Runtime in Improved DynTM

When a new transaction begins its work on a core, it firstly checks the transaction table to obtain the surrounding transactions; then it compares the transactional read/write set with currently running transactions' minimum conflict units to predict the possible conflict, once the possible conflict is find in this step, it will decide whether continue to execute this transaction that is much depended on the transaction length and conflict level of this transaction and the dynamically length of the competitor transactions. When it is a conflict between two coarse-grained and high-contention transactions, the system will schedule the new transaction on to the competitor thread. When a new long transaction conflicts with a short already executing transaction, it will guarantee the competitor commit first to avoid wasting the computation by stall on conflict. On the other side, a new short transaction conflicts with a long already executing transaction will be schedule on the competitor thread. At last, when it is a conflict between two short transactions, the transactions can be schedule to the competitor or continue execute on this thread that is determined by the already scheduled transactions on the competitor threads.

Specifically, we have divided various transactions into 8 categories by their characteristics on contention, grained and length. To label these 8 different kinds of transactions, we use a triad such as "(H, C, L) "to represent the high-contention, coarse-grained and long length transactions, where the first parameter indicates either high-contention (i.e., H) or low-contention (i.e., L), the second parameter indicates either coarse-grained (i.e., C) or fine-grained (i.e., F), and the third parameter indicates either long-length (i.e., L) or short-length (i.e., S). To solve the transactional conflict, it usually stalls one of the conflicting transactions to wait the competitor transaction to

TABLE III: Basic Schedule Policy of The Runtime Environment, Where A, R, S and W Are Short for Abort, Restart, Schedule and Wait Respectively

	(H,C,L)	(H,C,S)	(H,F,L)	(H,F,S)	(L,C,L)	(L,C,S)	(L,F,L)	(L,F,S)
(H,C,L)	(A,S)	(W)	(A,S)	(W)	(A,R)	(W)	(A,R)	(W)
(H,C,S)	(W)	(W)	(A,S)	(W)	(A,R)	(A,R)/(W)	(A,R)	(A,R)/(W)
(H,F,L)	(A,S)	(A,S)	(A,S)	(W)	(A,R)	(A,R)/(W)	(A,R)	(A,R)/(W)
(H,F,S)	(W)	(W)	(W)	(W)	(A,S)	(W)	(A,R)/(W)	(W)
(L,C,L)	(A,R)	(A,R)	(A,R)	(A,S)	(A,R)	(W)	(A,R)	(W)
(L,C,S)	(W)	(A,R)	(A,R)/(W)	(W)	(W)	(W)	(A,R)	(W)
(L,F,L)	(A,R)	(A,R)	(A,R)	(A,R)/(W))	(A,R)	(A,R)	(A,R)	(W)
(L,F,S)	(W)	(A,R)	(A,R)/(W)	(W)	(W)	(W)	(W)	(W)

TABLE IV: The Execution Time Reduction of DynTM with Runtime Support over the Original Scheme

	NoTrans	Trans	Serializing	Wasted	Total Time Reduction
bayes	99%	100%	59%	56%	80.6%
genome	100%	102%	63%	78%	82.8%
intruder	98%	124%	62%	86%	83.5%
kmeans	99%	151%	89%	94%	100.2%
labyrinth	99%	100%	56%	71%	72.9%
ssca2	98%	231%	99%	96%	108.2%
vacation	98%	102%	85%	79%	99.5%
yada	99%	101%	73%	83%	84.3%
geo-mean	98.7%	120.8%	71.8%	79.4%	88.3%

finish its work, aborts one of the conflicting transactions to guarantee the data consistency while it can restart the aborted transaction after a random back off time or reschedule it onto the competitor transaction's thread. So the basic schedule policy can be concluded in a 8x8 table to solve the transactional conflicts, as what shown in Table III. Note the slots with the "(A,R)/(W)"policy means the runtime system can either abort the conflicting transaction and restart it later or wait for the conflicting transaction to finish its work, it is determined upon the number of waiting threads and the runtime environment will choose the right choice.

D. Evaluate The DynTM with Runtime Support

Due to the complex behaviors of the upper transactions, DynTM's performance is very sensitive with the accuracy of the transaction mode selector. We have proposed the runtime support to guide transaction schedule and conflict resolution. Here we will present the evaluation results of DynTM with runtime support to learn its benefit.

In Table IV, we also break down the execution time into four components, whose definitions are the same as what are defined in Figure 4, to learn the runtime environment's performance gain. Though the software overheads of the runtime may lengthen the Trans component in the fine-grained transactional applications such as intruder, kmeans and ssca2, it can significantly reduce the overheads spent on the Serializing and Wasted components that will be the most overheads of the conflicted transactions. In particular, the runtime environment works well under the high-contention and coarse-grained applications. And finally, the improved DynTM with runtime support obtains an average speedup of 11.7% over the original DynTM scheme under the STAMP benchmark suite, which indicates it is an efficient optimization on the existing DynTM scheme.

E. One Optimization on The Runtime Environment

Although the runtime augmented DynTM scheme achieves an average speedup of 11.7% over the original scheme, its scalability still rather low especially under the high-contention, coarse-grained and long-length applications such as bayes, labyrinth and yada. Most of time is spent on waiting the conflicting transactions that are aborted later. This is because the transactions' lengths in these applications are very long that the system can't avoid transactional conflicts happen once again on the restarted or rescheduled transactions. To overcome this shortcoming, we have optimized the runtime environment to collect the information about these high-contention, coarse-grained and long-length transactions, exploit the internal parallelism of the coarse-grained and long-length transactions.

During the comprehensive evaluation on the high-contention, coarse-grained and long-length transactional workloads on existing HTM platforms, we have already found the potential of exploiting the parallelism in the high-contention, coarse-grained and long-length transactions and proposed TMTLS [23] as the basic hardware architecture to support this function. Here we incorporate this method in our improved DynTM as an optimization to further reduce the transactional overheads. From the hardware view, no special component is needed to add because the hardware transactional memory (HTM) and thread level speculation (TLS) share similar underline hardware support components such as the processor's checkpoint, version management scheme and conflict detection method. We only to guarantee the sequential ordering of the spawned threads with the non-speculative threads because it usually extracts potential parallel block in programs and speculatively executes the sequential sections of code in parallel without violating the sequential semantics in the original codes, and fortunately it can be solved by embedding the timestamp information in the cache coherence messages to order these threads.

Currently, we only exploit the parallelisms of function calls and loop iterations in the high-contention, coarse-grained and long-length transactions. In Simics simulator platform, the magic instruction enables the researchers to add custom instructions and implement demanded functions. Like labeling transactional area in TM, thread level speculation also uses some micro instructions to label the TLS area. Ideally, there should be a compiler to assist identifying the speculative threads in programs which is left as our future work. To evaluate the effeteness

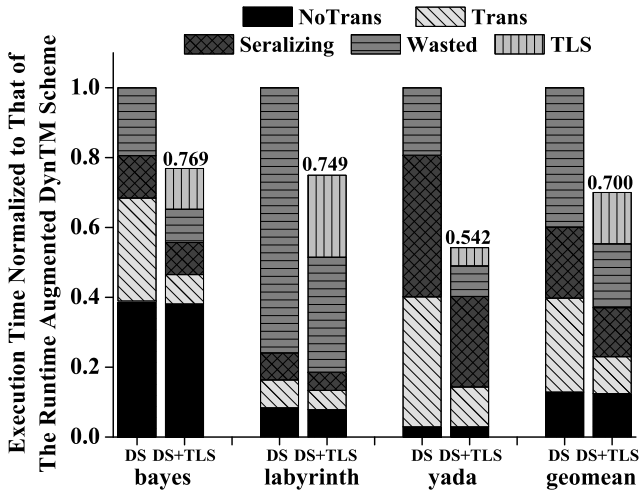

Fig. 7: The Execution Times of 3 High-contention Coarse-grained and Long-length Applications on The Runtime Augmented DynTM Scheme with the TLS Optimization and All Their Execution Times Are Normalized to That of The Runtime Augmented DynTM Scheme without The TLS Optimization

of our proposal, currently we have transformed the high-contention, coarse-grained and long-length transactions by inserting the macro into the source code. When we obtain the program's profile of bayes, labyrinth and yada, we find that more than 80% transactions in bayes, 100% transactions in labyrinth and 66.7% transactions in yada are long-length transactions which are the target to apply this optimization to exploit the parallelisms.

As shown in Figure 7, NoTrans, Trans, Serializing and Wasted components are defined the same as that in Figure 4 while the TLS component is defined as the time spent on thread level speculation. When we apply the TLS optimization on the runtime augmented DynTM scheme, these 3 high-contention, coarse-grained and long-length applications can reduce about 30% execution in geometric mean, thus meaning this optimization can achieve an average speedup of 42.9% throughout these 3 applications, which is a big leap ahead over than the original scheme.

IV. Conclusion

Through the comprehensive evaluation of high-contention and coarse-grained workloads on the existing hardware transactional memory systems, we find two pathologies stemmed from the interplay between the conflict management and version management that may impact thread level parallelism significantly. Even though DynTM, one of the latest proposals in HTMs, exploits the complementary characteristics between the eager and lazy transaction schemes, its performance is still limited by the predictor's accuracy and the size of the speculative modification during the transaction execution. Otherwise, the DynTM scheme will also suffer from the existing pathology especially when it degenerates to the eager scheme. Moreover, the complexity behaviors require the predictor to schedule the transactions at some form of priority, which make it hard to implement in hardware at an economic cost. To make a better transactional memory system, we propose a runtime environment to recognize the dynamic conflicting behaviors of the running transactions and guide the conflict resolution that can significantly boost the performance. After evaluation, we find that our runtime augmented DynTM scheme obtains an average speedup of 11.7% across the 8 transactional applications selected from the STAMP benchmark suite over the DynTM proposal. In order to further reduce the transactional overheads on the high-contention, coarse-grained and long-length transactions, we have incorporated the thread level speculation optimization in the runtime augmented DynTM scheme and found this optimization can achieve an average speedup of 42.9% over the original scheme throughout the 3 high-contention, coarse-grained and long-length transactional workloads such as bayes, labyrinth and yada.

Acknowledgements

This work was supported by the National Basic Research 973 Program of China under Grant No. 2011CB302301, the National High Technology Research and Development Program ("863" Program) of China under Grant No. 2009AA01A402, National Natural Science Foundation of China (NSFC) under Grant No. 61025008, 60933002,60873028, Changjiang innovative group of Education of China No. IRT0725.

References

[1] AMD. Advanced Synchronization Facility Proposed Architectural Specification. 2009.

[2] A. Armejach, A. Seyedi, and et.al. Using a Reconfigurable L1 Data Cache for Efficient Version Management in Hardware Transactional Memory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 361 –371, 2011.

[3] G. Blake, R. Dreslinski, and T. Mudge. Proactive Transaction Scheduling for Contention Management. In *Proceedings of the 42nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 156–167. 2009.

[4] J. Bobba, K. Moore, and et.al. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 81–91. 2007.

[5] S. Chaudhry. Rock: A third Generation 65nm, 16-Core, 32 Thread + 32 Scout-Threads CMT SPARC Processor. In *Proceedings of the 20th IEEE International Symposium on High Performance Chips (HotChips)*. 2008.

[6] C. Click. Azul's Experiences with Hardware Transactional Memory. In *Transactional Memory Workshop*. 2009.

[7] L. Hammond, V. Wong, and el.al. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 102–113. 2004.

[8] R. Haring. The IBM Blue Gene/Q Compute Chip+SIMD Floating-point Unit. In *Proceedings of the 23th IEEE International Symposium on High Performance Chips (HotChips)*. 2011.

[9] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory (2nd edition)*. Morgan & Claypool, 2010.

[10] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300. 1993.

[11] Intel. Intel Architecture Instruction Set Extensions Programming Reference. 2012.

[12] M. Lupon, G. Magklis, and A. González. FasTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 293–302. 2009.

[13] M. Lupon, G. Magklis, and A. González. A Dynamically Adaptable Hardware Transactional Memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 27–38. 2010.

[14] P. Magnusson, M. Christensson, and et.al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35:50–58, 2002.

[15] M. Martin, D. Sorin, and et.al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33:92–99, November 2005.

[16] C. Minh, J. Chung, and et.al. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the 4th IEEE International Symposium on Workload Characterization (IISWC)*, pages 35–46. 2008.

[17] K. Moore, J. Bobba, and et.al. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 254–265. 2006.

[18] S. H. Pugsley, M. Awasthi, and et.al. Scalable and Reliable Communication for Hardware Transactional Memory. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 144–154, 2008.

[19] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 240–248, 2005.

[20] A. Shriraman and S. Dwarkadas. Refereeing Conflicts in Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, pages 136–146. 2009.

[21] S. Tomic, C. Perfumo, and et.al. EazyHTM: EAger-LaZY Hardware Transactional Memory. In *Proceedings of the 42nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 145–155. 2009.

[22] Z. Yan, D. Feng, and Y. Tan. Poster: Dissection The Version Management Schemes in Hardware Transactional Memory Systems. *SIGMETRICS Perform. Eval. Rev.*, 39:78–78, September 2011.

[23] Z. Yan, D. Feng, and Y. Tan. TMTLS: Combine TM with TLS to Limit The Memory Contentions and Exploit The Parallelism in The Long-Running Transactions. In *Proceedings of the 6th International Conference Networking, Architecture, and Storage (NAS)*, pages 140–148. 2011.

[24] L. Yen, J. Bobba, and et.al. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–272. 2007.