Adaptive Versioning in Transactional Memories

Pavan Poudel and Gokarna Sharma(✉)

Department of Computer Science, Kent State University, Kent, OH 44242, USA
{ppoudel,sharma}@cs.kent.edu

Abstract. Transactional memory has been receiving much attention from both academia and industry. In transactional memory, program code is split into *transactions*, blocks of code that appear to execute atomically. Transactions are executed speculatively and the speculative execution is supported through data versioning and conflict detection and resolution mechanisms. *Lazy* versioning makes aborts fast but penalizes commits, whereas *eager* versioning makes commits fast but penalizes aborts. In this paper, we present an *adaptive* versioning approach that dynamically switches between eager and lazy versioning at runtime based on appropriate system parameters so that the performance of a transactional memory system is always better than that is obtained using either eager or lazy versioning individually. We implemented our adaptive versioning approach in the latest TinySTM distribution and extensively evaluated it through 5 micro-benchmarks and 8 complex benchmarks from STAMP and STAMPEDE suites. The results show significant benefits of our approach, giving performance improvements as much as 6.3x for execution time and as much as 170x for number of aborts.

1 Introduction

Concurrent processes (threads) need to synchronize to avoid introducing inconsistencies while accessing shared data objects. Traditional synchronization mechanisms such as locks and barriers have well-known limitations and pitfalls, including deadlock, priority inversion, reliance on programmer conventions, and vulnerability to failure or delay. *Transactional memory* (TM) [16,27] has emerged as an attractive alternative. Several commercial processors provide direct hardware support for TM, including Intel's Haswell [17] and IBM's Blue Gene/Q [14], zEnterprise EC12 [23], and Power8 [6]. There are proposals for adapting TM to clusters of GPUs [5,12,20].

Using TM, program code is split into *transactions*, blocks of code that appear to execute atomically. Transactions are executed *speculatively*: synchronization conflicts or failures may cause an executing transaction to *abort*: its effects are rolled back and the transaction is restarted. In the absence of conflicts or failures, a transaction typically *commits*, causing its effects to become visible. Supporting

This work is supported by the National Science Foundation grant CCF-1936450.

M. Ghaffari et al. (Eds.): SSS 2019, LNCS 11914, pp. 277–295, 2019.
https://doi.org/10.1007/978-3-030-34992-9_22

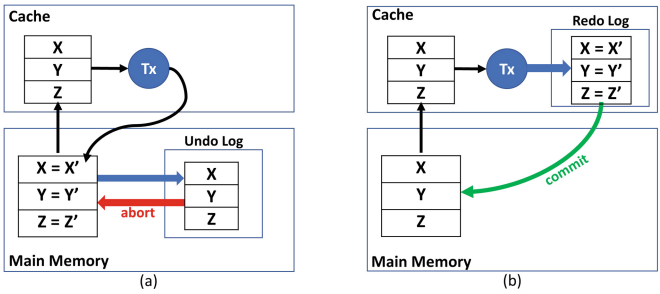

Fig. 1. An illustration of how a transaction T_x is executed using (a) eager versioning and (b) lazy versioning. Figure (a) depicts two kinds of operations in eager versioning, the first to copy the data from original memory locations to a log area (called *undo* log) in main memory and the second to copy the data back from the log area to the original memory locations, in case T_x aborts. If T_x commits, the data in the log area is simply discarded. Figure (b) depicts two kinds of operations in lazy versioning, the first to copy the updated values in cache to a log area (called *redo* log) in cache and the second to copy data from the log area to the original memory locations, in case T_x commits. If T_x aborts, the data in the log area is simply discarded. (Color figure online)

this speculative execution requires *data version management* and *conflict detection and resolution* mechanisms. The majority of the existing TM systems can be distinguished on how they implement these concepts. This is true for TM systems in hardware, called hardware TMs (HTMs) [4,13,22,26], as well as in software, called software TMs (STMs) [2,8,9].

Versioning handles the simultaneous storage of both *new* data (to be visible if transaction commits) and *old* data (retained if transaction aborts). At most one of these values can be stored "in place" (the original memory location), while the other value must be stored "on the side" (e.g., in cache or main memory). On a store, a TM system can either use *eager* versioning and put the new value in place or use *lazy* versioning to (temporarily) leave the old value in place. Figure 1 depicts how a transaction T_x is executed using eager and lazy versioning. Due to the working principle, lazy versioning makes aborts fast, but penalizes (the most frequent) commits, whereas eager versioning makes commits fast, but penalizes (the most frequent) aborts [22].

Conflict detection signals an overlap between the *write set* (data written) of one transaction and the write set or *read set* (data read) of other concurrent transactions. Conflict detection is called *eager* if it detects offending loads or stores immediately and *lazy* if it defers detection until later when transactions commit. Table 1 illustrates some existing TM systems that use lazy versus eager versioning and lazy versus eager conflict detection. *Conflict resolution* (or management) strategies are then used to decide on which conflicting transaction(s) to continue and which transaction(s) to wait (or abort and restart) the execution.

Table 1. Versioning and conflict detection mechanisms used in some TM systems.

		Versioning	
		Lazy	Eager
Conflict	Lazy	TCC [13], Norec [8], RSTM [2], SwissTM [9]	None
	Eager	LTM [4], VTM [26], RSTM [2], SwissTM [9]	UTM [4], LogTM [22], RSTM [2]

Both eager and lazy versioning along with both eager and lazy conflict detection and resolution have been studied heavily in the past for TM systems [2,4,8,9,13,22,26]. However, which versioning is better is still not clear and the studies provide contradictory conclusions. For example, consider two widely popular HTM implementations LogTM [22] and UTM [4]. They advocate that TM should ideally use eager versioning and eager conflict detection since in eager versioning transaction commits are faster than transactions aborts. Moreover, commits are much more common than aborts in practical applications. In addition, eager conflict detection finds conflicts early and reduces the wasted work by conflicting transactions. On the other hand, consider again widely popular HTM implementation TCC [13]. They use lazy versioning and lazy conflict detection. Other HTMs such as VTM [26] and LTM [4] advocate lazy versioning with eager conflict detection. This is also the case in STMs as some use eager, some use lazy, and some use the combination of eager and lazy approaches [2,8,9].

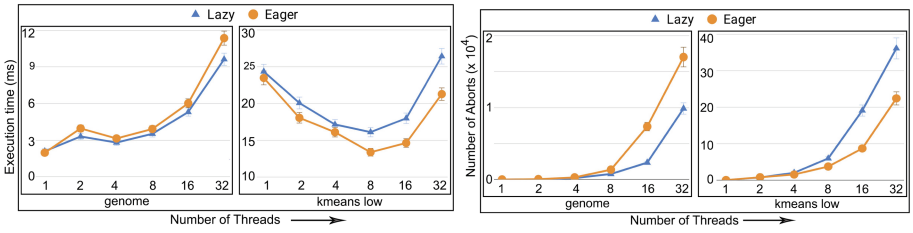

Fig. 2. An illustration of performance discrepancies in execution time (left) and number of aborts (right) in *genome* and *kmeans* benchmarks using eager and lazy versioning.

There is no study that elaborates the performance gap between eager and lazy versioning for TM systems. Figure 2 illustrates the performance discrepancies using eager and lazy versioning while executing *genome* and *kmeans* benchmarks from STAMP benchmark suite [21]. Lazy versioning performs well for *genome* where as for *kmeans* the opposite is true. This is mainly because of the fact that the versioning used is not appropriate for the workload and caused more number of aborts, subsequently increasing the execution time. Nevertheless, there are two major issues in selecting an appropriate versioning for TM systems. First, to select an appropriate versioning, a priori knowledge on the workload

(write-dominated or read-dominated) and contention scenario (low or high) is needed. Second, such knowledge is difficult to obtain prior to runtime.

Contributions. In this paper, we demonstrate that we can obtain the best of both worlds without any a priori knowledge on the workload and contention scenario. Particularly, we present an *adaptive* versioning for TM systems, which we call ADAPTIVE, that dynamically switches the execution using either lazy or eager versioning at runtime, always achieving performance on any workload and contention scenario better than that is obtained using either lazy or eager versioning individually. For the experimental evaluation, we incorporated ADAPTIVE in the latest TinySTM implementation [10,11] and ran experiments against a diverse set of TM benchmarks [10,11,15]. Specifically, we used 5 micro-benchmarks (bank, red black tree, hash set, linked list, and skip list) and 8 complex benchmarks (yada, vacation, ssca2, labyrinth, kmeans, intruder, genome, and bayes) from STAMP and STAMPEDE benchmarks [21,24]). We measured the performance of ADAPTIVE w.r.t. two crucial performance metrics.

- **execution time:** the total time to complete executing a set of transactions. This is the time interval from the beginning of the first transaction executed until the last transaction finishes and commits. In a dynamic setting, the execution time translates to *throughput*, the number of committed transactions per time step.
- **number of aborts:** the total number of transaction aborts until the current time. If compared with the total number of transaction commits until the current time, it provides *abort-to-commit ratio* (ACR), a useful metric.

Both metrics are fundamental and used extensively in evaluating TM systems. The number of aborts directly affect execution time since it is likely that the execution time increases with the increasing number of aborts requiring more number of transaction restarts.

The results show that, when using lazy versioning with eager conflict detection, ADAPTIVE achieves up to 6.3× better performance than lazy versioning and up to 5.5× better performance than eager versioning. When using lazy versioning with lazy conflict detection, ADAPTIVE achieves up to 3.7× better performance than lazy versioning and up to 5× better performance than eager versioning. The minimum performance gain for ADAPTIVE is 1.12. These results suggest that switching between eager and lazy versioning dynamically at runtime provides a way to exploit the positive aspects of both versioning methods for TM systems. ADAPTIVE is general enough to be applied to both HTMs and STMs, although we only report results from a STM implementation. In summary, we have the following three contributions.

- (**Section** 4) We introduce a novel versioning approach, ADAPTIVE, that switches between eager and lazy versioning dynamically at runtime.
- (**Section** 5) We discuss implementation issues related to ADAPTIVE and present two optimizations.

– (**Section 6**) We evaluate experimentally the performance of ADAPTIVE using five micro-benchmarks and 8 complex benchmarks from STAMP and STAMPEDE, report the results obtained, and provide observations on the obtained results.

2 Related Work

The previous studies on TM mostly supported speculative execution of transactions using either eager or lazy versioning. There is no work that elaborates on the impact of using eager and lazy versioning on the performance of TM systems. In fact, as outlined in Table 1, the majority of well-known TM systems make contradictory conclusions on whether to use eager or lazy versioning. We focus in this paper on the impact of the eager and lazy versioning on the performance of TM systems. Particularly, we propose an adaptive versioning (switching between eager and lazy versioning at runtime without needing any a priori knowledge on workload and contention scenarios) and achieve significant improvements in execution time and number of aborts (two crucial performance metrics for evaluating a TM system) compared to that of using either eager or lazy versioning individually. Our approach is simple and may provide insights into future TM system designs and implementations.

The performance gap of using eager and lazy versioning is relatively well-studied for crash consistency in non-volatile memories. One recent work is [25] where they presented an adaptive versioning approach like the one presented here but specifically tailored to non-volatile memories. In particular, they focused on minimizing *the number of data movements* while running workloads through these versioning methods. However, their approach increased the execution time in several benchmarks. The approach we study here is tailored for TM systems in volatile memories.

The other closely related works are as follows. Wan *et al.* [29] empirically evaluated eager and lazy versioning on the open source non-volatile memory library (NVML) [1] for some constrained workloads, and suggested that "*one logging method does not fit all workloads*". Particularly, they reported that (i) lazy versioning significantly outperforms eager versioning for workloads in which a transaction updates large number of different objects, while it underperforms eager versioning for read-dominated workloads, and (ii) eager versioning is more sensitive to *read-to-write* ratios whereas lazy versioning is less sensitive to those ratios [29]. The other works mostly proposed methods to provide crash consistency through either eager or lazy versioning, and there is no work that elaborates the performance gap between eager and lazy versioning. Coburn *et al.* [7] suggested a STM implementation for persistent memory NV-HEAPS using eager versioning. Volos *et al.* [28] suggested a TinySTM [10,11] variation MNEMOSYNE for persistent memory using lazy versioning. NV-HEAPS [7] and MNEMOSYNE [28] drew absolutely opposite conclusions on whether eager or lazy versioning is better for persistent memory. The former prefers to use eager versioning, and

the latter opts to use lazy versioning. Recently, Alistarh *et al.* [3] studied two variants of the transactional conflict problem and provided optimal solutions for both the variants.

3 Preliminaries

Model. We consider a computer system with volatile shared main memory, many processing cores, and hard disk drive. All shared main memory is cacheable and caches are volatile and coherent. We assume that all the writes of a committed transaction can be accommodated in the cache, i.e., once a transaction commits but before the commit is reflected in original memory locations in main memory, all its newly modified data is in volatile cache. We run workloads using the TinySTM execution model [10,11]. We assume that the execution starts at time $t_0 = 0$. We measure in execution time the time for all the transactions within a benchmark to finish execution and commit, except for micro-benchmarks where we consider time to execute and commit 10,000 transactions. We also assume that only a single-version of data is stored in each eager, lazy, and adaptive versioning, which is essentially different from techniques, such as those given in [18], of storing multiple versions.

Eager Versioning. Eager versioning is supported through so-called *undo logs*. Undo logs are stored in cacheable main memory. In this method, a transaction works by first copying the data in original memory locations to a undo log area and then performs updates in-place in the original data locations (in main memory). In the event the transaction aborts, any modifications to the original memory locations are *rolled back* using the old data stored in the undo log. The left of Fig. 1 illustrates eager versioning.

Lazy Versioning. Lazy versioning is supported through so-called *redo logs*. Redo logs are stored in cache. In this method, a transaction copies data in each memory location that it is going to read/write to a redo log area, appends all its data updates to that log area, and then writes the data back to original memory locations when transaction commits. If the transaction fails, the updates in log area are simply discarded. Therefore, the writing of data in redo log back to the original memory locations happens only when transaction commits. The right of Fig. 1 illustrates lazy versioning.

4 Basic Adaptive Versioning

We now describe our approach, ADAPTIVE, that runs transactions using either eager or lazy versioning, switching between them dynamically at runtime. Figure 3 compares ADAPTIVE with eager and lazy versioning. We will introduce techniques to improve ADAPTIVE in Sect. 5.

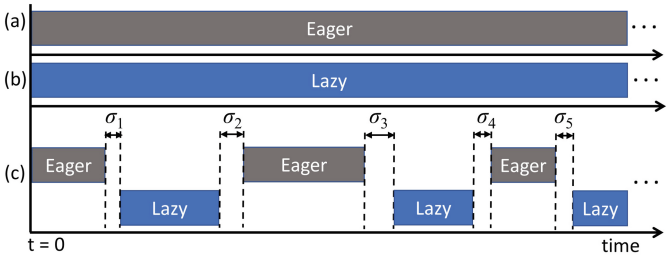

Fig. 3. An illustration of (a) eager, (b) lazy, and (c) basic adaptive versioning. The time gap σ_* while switching from eager (lazy) to lazy (eager) is to let finish executing in-flight transactions. This helps in avoiding potential data versioning inconsistencies.

High Level Overview. The high level idea in ADAPTIVE is to switch the versioning method depending on performance. That is, if the versioning currently used is hampering the performance, then we switch the versioning to improve the performance.

Now a fundamental question is how to identify and measure an indicator that reflects appropriately the effect of the versioning method on performance. Fortunately, in TM, if the number of aborts are increasing compared to the number of commits, then it might be a valid indicator of performance degradation due to the versioning method currently used. Therefore, we pick abort to commit ratio (ACR) as a performance indicator for any versioning method. ACR has also been used quite heavily in the TM literature as a vital indicator of performance, for example, see [19]. Ideally, the goal is to have no aborts, i.e., $ACR = 0$. However, in practice, this may not be feasible and the goal is to minimize ACR as much as possible.

Formally, ACR can be defined as follows: $ACR = \frac{N_{abort}}{N_{commit}}$, where N_{abort} is the total number of aborted transactions and N_{commit} is the total number of committed transactions from time 0 up to t. For eager (and lazy) versioning, we can compute ACR_{Eager} (and ACR_{Lazy}) based on the number of transactions committed and aborted using eager (lazy) versioning. To facilitate when to switch from one to another, we identify a *threshold* on ACR for both eager and lazy. We denote them by $Threshold_{Eager}$ and $Threshold_{Lazy}$, respectively. Let a transaction T be running at current time t using lazy versioning. If $ACR_{Lazy} < Threshold_{Lazy}$, then the versioning method is switched to Eager for transactions that start (or restart) execution after time $t' > t$. An analogous approach is used if currently T is executing using eager versioning.

Detailed Description. Let $N_{Ecommit}(N_{Lcommit})$ be the number of transaction commits in ADAPTIVE from time $t_0 = 0$ until the current time $t > t_0$ executed using eager (lazy) versioning. Similarly, let $N_{Eabort}(N_{Labort})$ be the total number of transaction aborts in ADAPTIVE from time $t_0 = 0$ until time $t > t_0$ executed using eager (lazy) versioning. Furthermore, let N_{commit} and N_{abort} be the total

number of commits and aborts in ADAPTIVE from $t_0 = 0$ until time $t > t_0$. Notice that $N_{commit} = N_{Ecommit} + N_{Lcommit}$ and $N_{abort} = N_{Eabort} + N_{Labort}$.

Based on $N_{Ecommit}, N_{Lcommit}, N_{Eabort}$, and N_{Labort}, we compute ACR_{Eager} and ACR_{Lazy} at each time step $t > t_0$. These ratios ACR_{Eager} and ACR_{Lazy} are then compared with $Threshold_{Eager}$ and $Threshold_{Lazy}$ parameters (computed in the next section). Therefore, at any time $t > t_0$, the transaction T that is ready-to-execute will be executed as follows.

- Suppose the versioning currently used is $L_{cur} = Eager$. If $ACR_{Eager} > Threshold_{Eager}$, then L_{cur} is switched to $Lazy$ (i.e., $L_{cur} \leftarrow Lazy$) and T will be executed using lazy versioning.
- Suppose the versioning method currently used is $L_{cur} = Lazy$. If $ACR_{Lazy} < Threshold_{Lazy}$, then L_{cur} is switched to $Eager$ (i.e., $L_{cur} \leftarrow Eager$) and T will be executed using eager versioning.

In the special case of $t_0 = 0$, $N_{Ecommit}, N_{Lcommit}, N_{Eabort}$, and N_{Labort} are all zero. Therefore, a simple approach is to execute T using either lazy or eager versioning. However, if some information regarding the workload is available, then we can decide on which versioning method to use. Suppose, the read and write sets of T are available. Let $Wset(T)$ be the *write set* of T which is essentially the memory locations that T would modify while executing. Similarly, let $Rset(T)$ be the *read set* of T which is essentially the memory locations that T would read (but not modify) while executing. $RW(T) = Rset(T) + Wset(T)$, where $RW(T)$ denotes the total number of memory locations that T reads and modifies while executing. If $|Wset(T)| > |Rset(T)|$, then T is executed using lazy versioning, otherwise using eager versioning.

Computing Switching Thresholds $Threshold_{Eager}$ and $Threshold_{Lazy}$. Let N be the total number of transactions in any workload. When the workload finishes execution and all transactions commit, we have that $N_{commit} = N$ and $N_{abort} \geq 0$ (if each transaction commits without aborting, then $N_{abort} = 0$, otherwise $N_{abort} > 0$).

Suppose, each transaction T spends α amount of time while moving data from one memory location to other. Consider the case of executing T using eager versioning. Let τ_{Eager} be the total amount of time spent while (i) versioning data from the original memory locations to the undo log area and (ii) updating data from the undo log area back to the original memory locations. The first kind of operations are shown as a blue arrow in Fig. 1(a) and the second kind of operations are shown as a red arrow in Fig. 1(a). The first kind of operations are always done in eager versioning and the second kind of operations are done only when the transaction aborts. That means, for an aborted transaction, data movement is performed two times, one for versioning, other for rollback. Therefore, for eager versioning, $\tau_{Eager} = (N_{commit} + 2N_{abort}) \cdot \alpha$.

Similarly, for lazy versioning, $\tau_{Lazy} = (2N_{commit} + N_{abort}) \cdot \alpha$.

Based on 3 different cases below, we can see 3 scenarios for τ_{Eager} and τ_{Lazy}:

- Case 1: **If** $N_{commit} = N_{abort}$, **then** $\tau_{Eager} = \tau_{Lazy}$

– Case 2: **If** $N_{commit} > N_{abort}$, **then** $\tau_{Eager} < \tau_{Lazy}$
– Case 3: **If** $N_{commit} < N_{abort}$, **then** $\tau_{Eager} > \tau_{Lazy}$

Moreover, equation for τ_{Eager} suggests that in eager versioning, total time spent for an aborted transaction is twice as much as the time spent for a committed transaction. Then it is immediate that the eager versioning performs better until $N_{commit} \geq 2N_{abort}$; i.e. $\frac{N_{abort}}{N_{commit}} \leq \frac{1}{2}$. Thus, we get $Threshold_{Eager} = \frac{1}{2}$ and switch to lazy versioning when $ACR_{Eager} > \frac{1}{2}$. Similarly, equation for τ_{Lazy} suggests that the lazy versioning performs better until $2N_{commit} \leq N_{abort}$; i.e. $\frac{N_{abort}}{N_{commit}} \geq 2$. Then, we get $Threshold_{Lazy} = 2$ and switch to eager versioning when $ACR_{Lazy} < 2$.

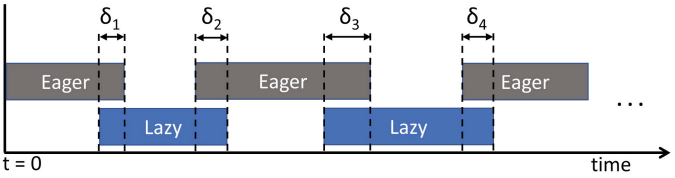

Fig. 4. An illustration of the better time barrier design. The interval δ_* between *Eager* and *Lazy* represents the time taken by in-flight transactions to finish their executions after versioning method is switched. The new transaction that do not conflict with transactions using previous versioning can execute concurrently with in-flight transactions.

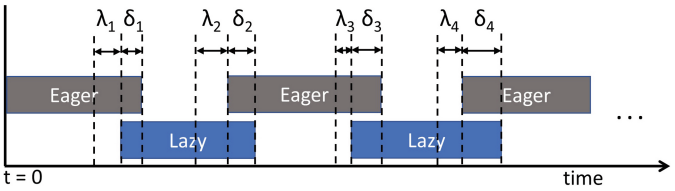

Fig. 5. An illustration of the better switching mechanism. λ_* represents the time interval in which versioning is not switched. δ_* resembles better time barrier of Fig. 4.

Time Barrier Requirement and Design. The ideal scenario in ADAPTIVE is to let each transaction T run Algorithm and decide which versioning (eager or lazy) to use for it to execute individually based on the parameters obtained at runtime. Let S_j be a set of transactions arrived before T. Suppose $L_{cur} = Eager$, which means that $L_{prev} = Lazy$. Suppose the versioning changed to *Eager* from *Lazy* after the transactions in S_j started execution but before T. If we run T using *Eager* immediately and T conflicts with any of the transaction $T_j \in S_j$, then the conflict detection and resolution mechanisms interfere, hampering

consistency. A simple approach to handle this situation is to ask T to wait until all transactions in S_j finish execution, which we call a *basic time barrier* (as shown in Fig. 3). The barrier reduces total number of aborts but due to a time delay before switching, it increases total execution time [25]. We provide a *better time barrier* design (described in Sect. 5) that will minimize this overhead.

5 Optimizations on Basic Adaptive Versioning

We provide two optimizations to basic ADAPTIVE. The first optimization is on time barrier design. The second optimization is on switching mechanism.

Better Time Barrier Design. Figure 4 illustrates the idea of better time barrier design. Consider a transaction T. Let S_j be a set of transactions arrived before T. Suppose $L_{cur} = Eager$, which means $L_{prev} = Lazy$. Suppose the versioning changed to $Eager$ from $Lazy$ after the transactions in S_j started execution but before T starts execution. In the basic time barrier design, T has to wait until all transactions in S_j finish execution. In this design, we ask T to start execution as soon as it is ready. If T does not conflict with transactions in S_j, we are done, otherwise, T aborts. If T conflicts with $T' \notin S_j$, it is handled as per the conflict resolution strategy used.

Better Switching Mechanism. Let $L_{cur} = Eager$. Suppose at time t, ADAPTIVE decides to switch to $Lazy$. We discuss here a mechanism so that ADAPTIVE does not switch to $Lazy$ at t but waits until a switching interval threshold SW_INT. We define SW_INT as the number of transactions after t for which the decision is to execute using $Lazy$. Let λ be the execution time interval during which all transactions in the interval SW_INT finish execution. Execution switches from $Eager$ to $Lazy$ at time $t + \lambda$. Figure 5 illustrates the design of better switching mechanism.

6 Experimental Evaluation

In this section, we evaluate the performance of optimized[1] ADAPTIVE (better time barrier and switching mechanism). The evaluation is performed in a STM implementation using TinySTM [10,11] modified appropriately to incorporate ADAPTIVE. The tests were executed on an Intel Xeon(R) E5-2620 v4 @ 4.20 GHz, 64-bit processor with 32 cores. Each core has private L1 and L2 caches, whose sizes are 64 KB and 256 KB, respectively. There is also an 20 MB L3 cache shared by all 32 cores and 32 GB main memory. The results are the average of 10 experimental runs. The results are for varying number of threads from 1 to

[1] The experimental results conducted on basic ADAPTIVE showed that the number of aborts always decrease in all the benchmarks but execution time for some benchmarks increase compared to the execution times obtained using eager and lazy versioning individually.

32. First, we present the experimental results for optimized ADAPTIVE with better time barrier using *suicide* conflict resolution strategy. Later, we extend the results using both better time barrier and switching mechanism. We also compare the performance of optimized ADAPTIVE against four different conflict resolution strategies.

Compared Versioning Methods. We developed a STM-based implementation using TinySTM [10,11]. TinySTM has implemented separately both lazy and eager versioning (called *Lazy* and *Eager*) through *Write_Back* and *Write_Through* designs, respectively. With *Write_Through* design, transactions directly write to original memory locations and revert their updates in case the transactions abort. However, with *Write_Back* design, transactions work on a copy of data and delay their updates to the original memory locations until commit [10,11]. Furthermore, *Write_Back* design has two different implementations: *Write_Back_ETL* and *Write_Back_CTL*. *Encounter-time locking* (ETL) detects conflicts early at the time of write and acquires the lock on the memory address before it is written. *Commit-time locking* (CTL) defers conflict detection on memory address until commit, i.e., the lock is acquired on the memory address at the commit time. Therefore, there are two different implementations of *Lazy* in TinySTM: one based on *ETL* called *Lazy_ETL* and another based on *CTL* called *Lazy_CTL*. We obtain adaptive design *Adaptive_ETL* using *Lazy_ETL* and *Eager* versioning. Similarly, we obtain adaptive design *Adaptive_CTL* using *Lazy_CTL* and *Eager* versioning. We run experiments with five different designs *Lazy_ETL*, *Lazy_CTL*, *Eager*, *Adaptive_ETL*, and *Adaptive_CTL*.

Results on Micro-benchmarks. The execution time results in 5 different micro-benchmarks are provided in Fig. 6. Figure 7 provides the result for the number of aborts. The results are for 10,000 transactions, each executed with *update rate* of 20%. Figure 6 shows that the execution time decreases notably in ADAPTIVE as compared to the other versioning methods with the increase in number of threads for all the micro-benchmarks. Specifically, *Adaptive_ETL* achieved up to 6.3× better execution time than *Lazy_ETL* and *Adaptive_CTL* achieved up to 3.7× better execution time than *Lazy_CTL*. Compared to *Eager*, *Adaptive_ETL* achieved up to 5.5× better execution time and *Adaptive_CTL* achieved up to 5× better execution time. The minimum execution gain for *Adaptive_ETL* beyond 4 number of threads is 1.23 and for *Adaptive_CTL* is 1.20. Due to high contention for memory access when transactions are executed with more number of threads, the number of aborts increases with the increasing number of threads. Figure 7 shows that ADAPTIVE minimizes number of aborts. Specifically, *Adaptive_ETL* achieved up to 2.6× less number of aborts than *Lazy_ETL* and up to 5.8× less number of aborts than *Eager*. *Adaptive_CTL* achieved up to 2.2× less number of aborts than *Lazy_CTL* and up to 8× less number of aborts than *Eager*.

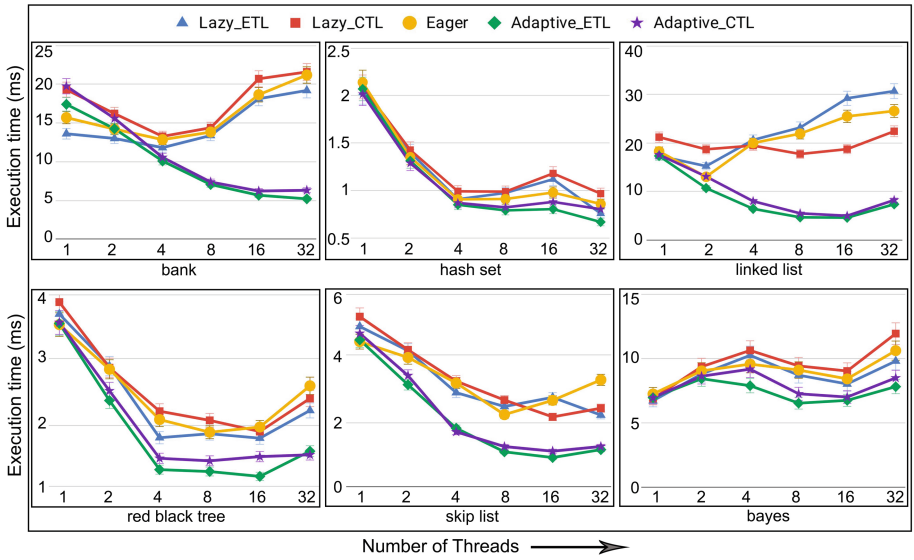

Fig. 6. Execution time in micro and bayes benchmark using better time barrier.

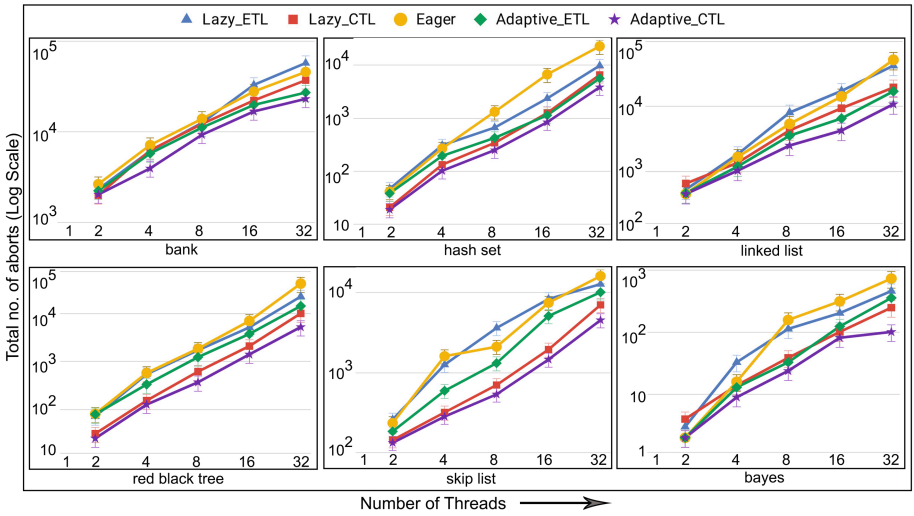

Fig. 7. Number of aborts in micro and bayes benchmark using better time barrier.

Results on STAMP Benchmarks. Figures 8 and 9, respectively, provide the execution time and number of aborts results for STAMP benchmarks. Regarding execution time, *Adaptive_ETL* has up to 1.78× better time than *Lazy_ETL* and *Adaptive_CTL* has up to 1.74× better time than *Lazy_CTL*. Compared to *Eager*, the execution time improvement in *Adaptive_ETL* and *Adaptive_CTL*

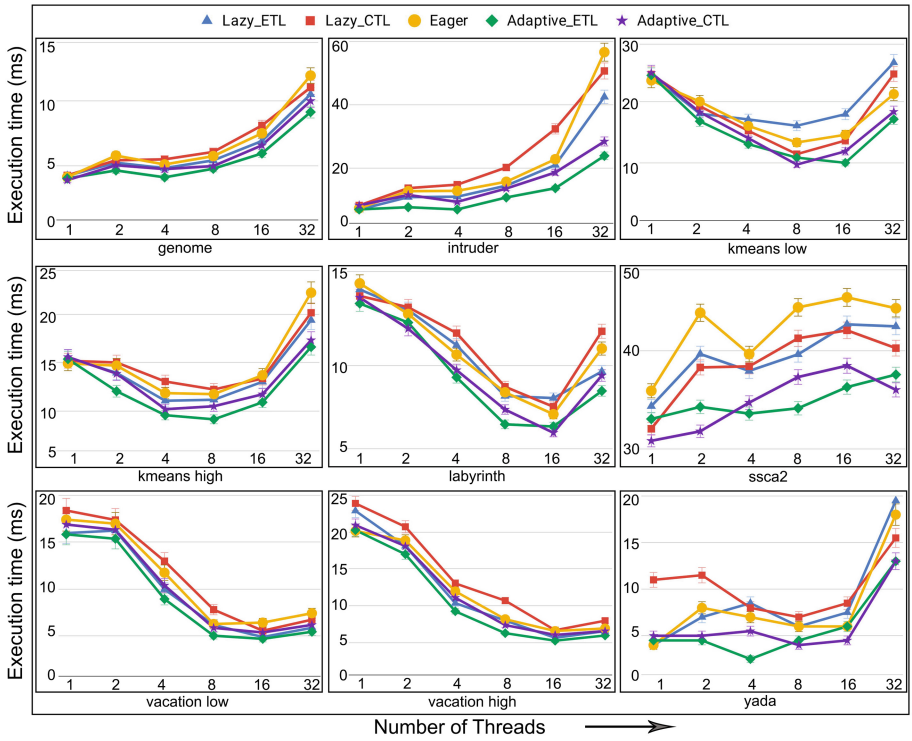

Fig. 8. Execution time in STAMP benchmarks using better time barrier.

is up to 2.36× and 2×, respectively. The minimum execution gain obtained in *Adaptive_ETL* is 1.13 and in *Adaptive_CTL* is 1.12 with the threads grater than 4. From Fig. 9, we observed that the number of aborts significantly increases in all the applications of STAMP benchmark when transactions are executed in more than 8 number of threads. Still, ADAPTIVE has significantly less aborts compared to *Lazy* and *Eager*. *Adaptive_ETL* has up to 16× less aborts than *Lazy_ETL* and up to 13× less aborts than *Eager*. Similarly, *Adaptive_CTL* has up to 2.5× less aborts than *Lazy_CTL* and up to 170× less aborts than *Eager*.

Results on STAMPEDE Benchmarks. Similar to micro and STAMP benchmarks, ADAPTIVE has better performance compared to *Lazy* and *Eager* in STAMPEDE benchmarks, for both execution time and number of aborts (Fig. 10). For execution time, *Adaptive_ETL* performed up to 1.72× better than *Lazy_ETL* and *Adaptive_CTL* performed up to 1.54× better than *Lazy_CTL*. Compared to *Eager*, *Adaptive_ETL* performed up to 1.68× better and *Adaptive_CTL* performed up to 1.91× better. The minimum execution gain obtained in *Adaptive_ETL* is 1.14 and in *Adaptive_CTL* is 1.12 with the threads greater than 4. For number of aborts, *Adaptive_ETL* performed up to

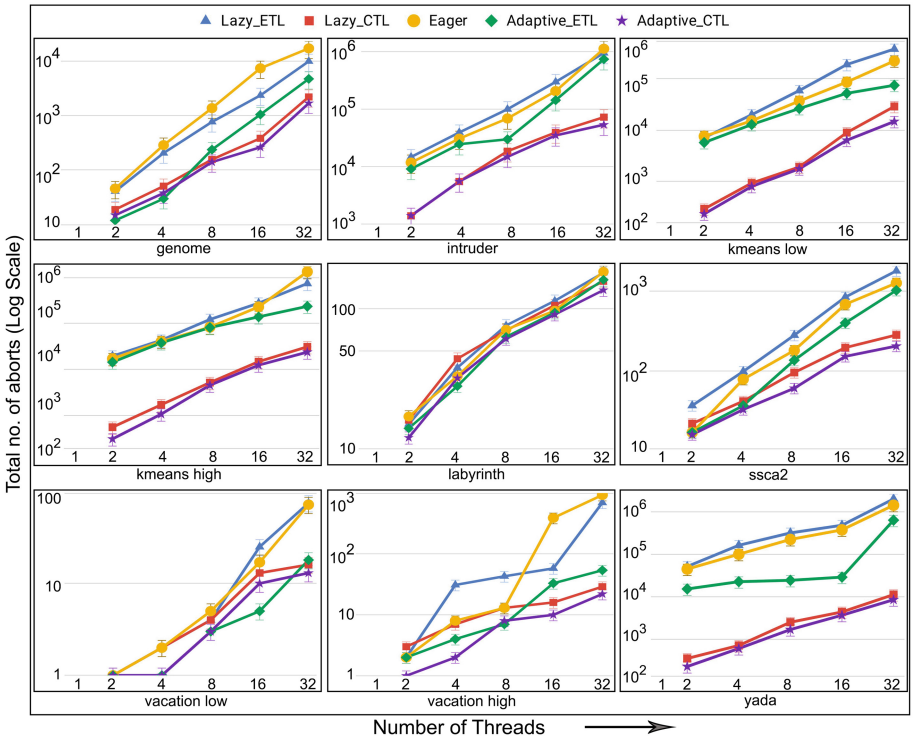

Fig. 9. Number of aborts in STAMP benchmarks using better time barrier.

4.1× better than *Lazy_ETL* and *Adaptive_CTL* performed up to 72× better than *Lazy_CTL*. Compared to *Eager*, *Adaptive_ETL* performed up to 10× better and *Adaptive_CTL* performed up to 124× better.

In all the benchmarks, the minimum execution gain for ADAPTIVE ranges between 1 and 1.16 when running with threads up to 4 numbers.

Further Results. The results in Figs. 6, 7, 8, 9 and 10 only considered optimized ADAPTIVE w.r.t. better time barrier. We also performed experiments for ADAPTIVE using both, better time barrier and better switching mechanism. We varied the switching interval threshold (SW_INT) from 2 up to 10. The results indicate that instead of switching versioning immediately, using the better switch mechanism increases the performance. However, for $SW\_INT > 2$, the performance gradually reduces and becomes worse while reaching $SW\_INT = 10$. Figure 11 shows the execution time for STAMP benchmarks when executed with both better time barrier and better switch mechanism ($SW\_INT = 2$). The improvement is up to 1.09× compared to ADAPTIVE with better time barrier. Alongwith decreasing the total number of aborts, the better switch mechanism decreases the total number of switches between the versioning methods which helps to get

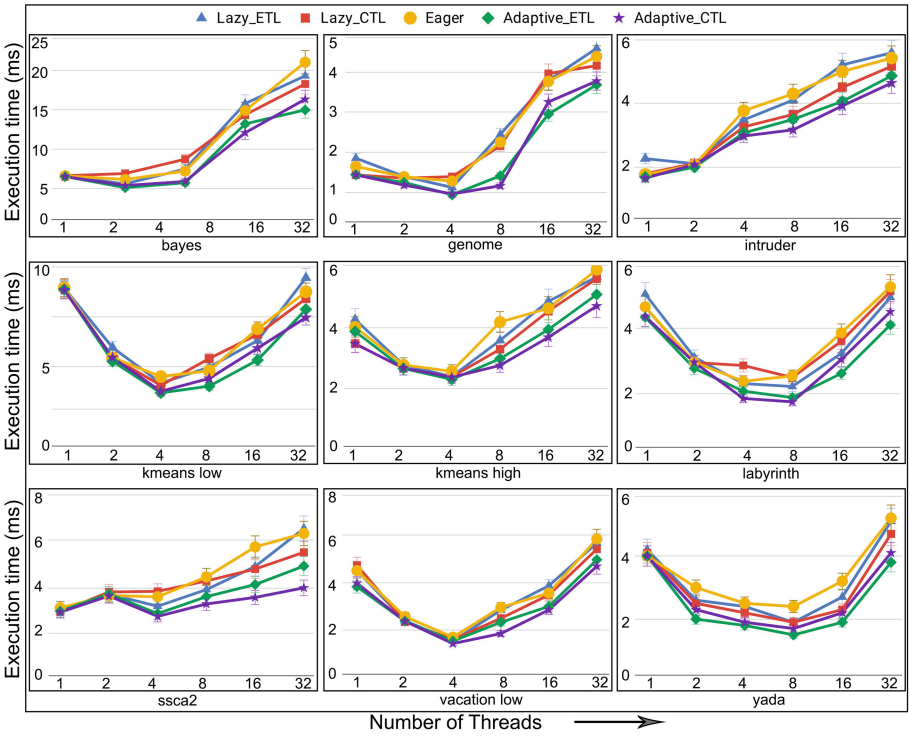

Fig. 10. Execution time in STAMPEDE benchmarks using better time barrier.

the improvement on execution time. Figure 12 illustrates the reduction of total number of switches using better switch mechanism for STAMP benchmarks. The experiments on micro-benchmarks and STAMPEDE showed similar results.

The experiments so far use $Threshold_{Eager} = \frac{1}{2}$ and $Threshold_{Lazy} = 2$ as computed in Sect. 4. It is natural to ask whether these are the ideal threshold values. Therefore, for $Threshold_{Eager}$, we used $\frac{1}{4}$ and $\frac{3}{4}$, whereas for $Threshold_{Lazy}$, we used 1 and 3. We performed experiments by using two different combinations of $Threshold_{Eager}$ and $Threshold_{Lazy}$, $(\frac{1}{4}, 1)$ and $(\frac{3}{4}, 3)$. We noticed the increase in both execution time and number of aborts in all the benchmarks for both the combinations. This suggests that the threshold values computed in Sect. 4 are appropriate.

The results reported in Figs. 6, 7, 8, 9, 10, 11 and 12 use *suicide* as a conflict resolution strategy. We were interested to see whether other strategies perform better than *suicide*. Therefore, we performed experiments using 4 different conflict resolution strategies *suicide, delay, back-off*, and *kill*. The results showed not significant change on performance in some of the benchmarks, while in the rest, the selection of conflict resolution strategy affected the performance. For example, *genome* and *intruder* performed better with *suicide* whereas, *kmeans*

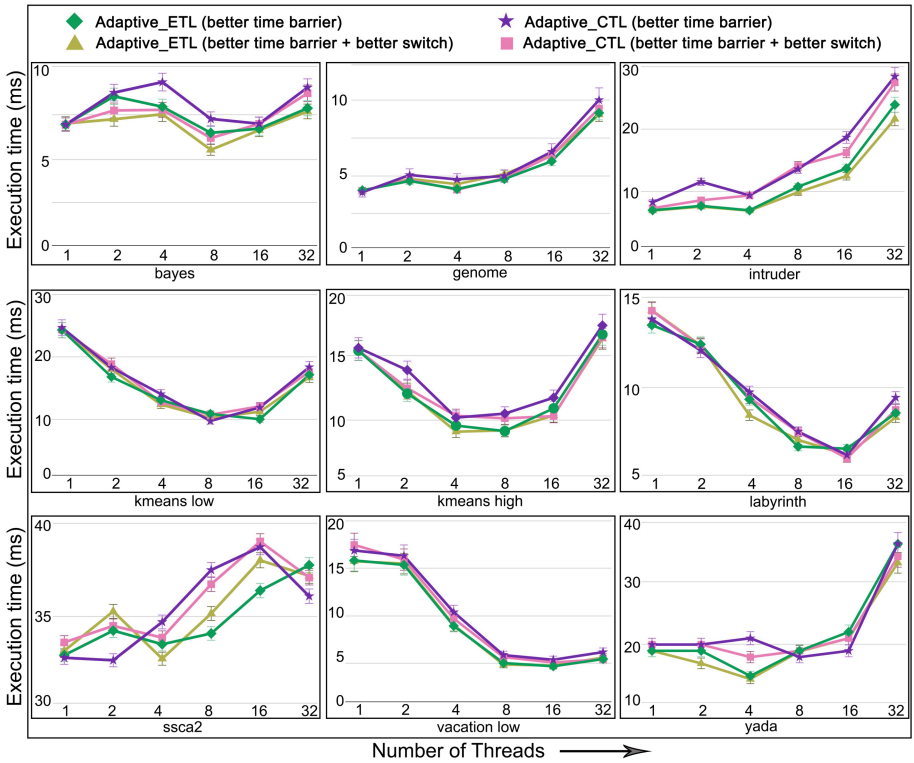

Fig. 11. Execution time in STAMP benchmarks using better barrier and better switch.

performed better with *back-off*. In overall, *suicide* performed better than the rest in most of the benchmarks.

Finally, we performed experiments starting the execution initially using eager and lazy versioning. We observed that the initial selection of versioning does not affect performance significantly in both micro and complex benchmarks except *intruder* and *kmeans* from STAMP in which ADAPTIVE performed better when starting with *Eager* than *Lazy* for up to 4 threads. This is mainly because transactions have almost constant abort rate and versioning method change is not necessary.

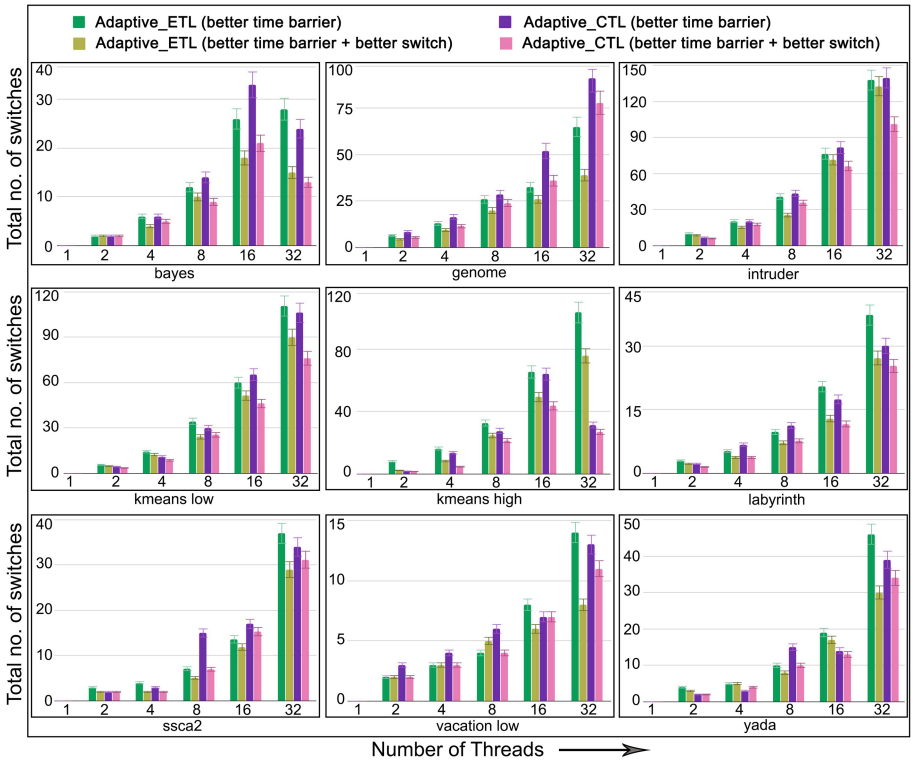

Fig. 12. Illustration of decrease in total number of switches between versioning methods using better switch mechanism.

7 Concluding Remarks

TM has been receiving much attention from both academia and industry. One of the most challenging issues in TM is how to ensure consistency of the shared data through speculative execution. Eager and lazy versioning have been used individually to support speculative execution in existing TM systems. However, whether to use eager or lazy versioning is better is not clear and previous studies contradict on the recommendations. In this paper, we have presented an adaptive framework that dynamically switches between eager and lazy versioning at runtime. Our framework is quite simple and achieves significantly better performance for execution time and number of aborts compared to eager and lazy versioning running individually in 5 micro-benchmarks and 8 applications from STAMP and STAMPEDE benchmarks. We believe that our results and techniques will be helpful in choosing proper versioning for TM systems.

References

1. The Persistent Memory Development Kit (PMDK). https://github.com/pmem/pmdk/. Accessed 14 Feb 2019
2. RSTM. http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml. Accessed 14 Feb 2019
3. Alistarh, D., Haider, S.K., Kübler, R., Nadiradze, G.: The transactional conflict problem. In: SPAA, pp. 383–392 (2018)
4. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: HPCA, pp. 316–327 (2005)
5. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: PPoPP, pp. 247–258 (2008)
6. Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le, H.Q.: Robust architectural support for transactional memory in the power architecture. In: ISCA, pp. 225–236 (2013)
7. Coburn, J., et al.: NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: ASPLOS, pp. 105–118 (2011)
8. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. In: PPOPP, pp. 67–78 (2010)
9. Dragojevic, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: PLDI, pp. 155–165 (2009)
10. Felber, P., Fetzer, C., Marlier, P., Riegel, T.: Time-based software transactional memory. IEEE Trans. Parallel Distrib. Syst. **21**(12), 1793–1807 (2010)
11. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPOPP, pp. 237–246 (2008)
12. Fung, W.W.L., Singh, I., Brownsword, A., Aamodt, T.M.: Hardware transactional memory for GPU architectures. In: MICRO, pp. 296–307 (2011)
13. Hammond, L., et al.: Transactional memory coherence and consistency. SIGARCH Comput. Archit. News **32**(2), 102 (2004)
14. Haring, R., et al.: The IBM Blue Gene/Q compute chip. IEEE Micro **32**(2), 48–60 (2012)
15. Herlihy, M., Luchangco, V., Moir, M., Scherer III., W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101 (2003)
16. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA, pp. 289–300 (1993)
17. Intel (2012). http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell
18. Keidar, I., Perelman, D.: Multi-versioning in transactional memory. In: Guerraoui, R., Romano, P. (eds.) Transactional Memory. Foundations, Algorithms, Tools, and Applications. LNCS, vol. 8913, pp. 150–165. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14720-8_7
19. Keidar, I., Perelman, D.: On avoiding spare aborts in transactional memory. Theory Comput. Syst. **57**(1), 261–285 (2015)
20. Manassiev, K., Mihailescu, M., Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In: PPoPP, pp. 198–208 (2006)
21. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC, pp. 35–46 (2008)
22. Moore, K.E.: LogTM: log-based transactional memory. In: HPCA, pp. 258–269 (2006)

23. Nakaike, T., Odaira, R., Gaudet, M., Michael, M.M., Tomari, H.: Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In: ISCA, pp. 144–157 (2015)
24. Nguyen, D., Pingali, K.: What scalable programs need from transactional memory. In: ASPLOS, pp. 105–118 (2017)
25. Poudel, P., Sharma, G.: An adaptive logging framework for persistent memories. In: Izumi, T., Kuznetsov, P. (eds.) SSS 2018. LNCS, vol. 11201, pp. 32–49. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03232-6_3
26. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: ISCA, pp. 494–505. IEEE Computer Society, Washington, DC (2005)
27. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
28. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: ASPLOS, pp. 91–104 (2011)
29. Wan, H., Lu, Y., Xu, Y., Shu, J.: Empirical study of redo and undo logging in persistent memory. In: NVMSA, pp. 1–6 (2016)