Article

Adaptive Versioning in Transactional Memory Systems †

Pavan Poudel and Gokarna Sharma *

Department of Computer Science, Kent State University, Kent, OH 44240, USA; ppoudel@cs.kent.edu
* Correspondence: sharma@cs.kent.edu

† This paper combines preliminary results that appeared in SSS'18 and SSS'19.

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 mechanism. *Lazy* versioning makes aborts fast but penalizes commits, whereas *eager* versioning makes commits fast but penalizes aborts. However, whether to use eager or lazy versioning to execute those transactions is still a hotly debated topic. Lazy versioning seems appropriate for write-dominated workloads and transactions in high contention scenarios whereas eager versioning seems appropriate for read-dominated workloads and transactions in low contention scenarios. This necessitates a priori knowledge on the workload and contention scenario to select an appropriate versioning method to achieve better performance. In this article, we present an adaptive versioning approach, called ADAPTIVE, that dynamically switches between eager and lazy versioning at runtime, without the need of a priori knowledge on the workload and contention scenario but 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 provide ADAPTIVE for both persistent and non-persistent transactional memory systems using performance parameters appropriate for those systems. We implemented our adaptive versioning approach in the latest software transactional memory distribution TinySTM 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. Specifically, in persistent TM systems, our approach achieved performance improvements as much as 1.5× for execution time and as much as 240× for number of aborts, whereas our approach achieved performance improvements as much as 6.3× for execution time and as much as 170× for number of aborts in non-persistent transactional memory systems.

Keywords: transactional memory; persistent memory; versioning; conflicts; concurrency; switching

Citation: Poudel, P.; Sharma, G. Adaptive Versioning in Transactional Memory Systems. *Algorithms* **2021**, *14*, 171. https://doi.org/10.3390/a14060171

Academic Editors: Frank Werner and Marina Papatriantafilou

Received: 4 February 2021
Accepted: 28 May 2021
Published: 31 May 2021

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) [1,2] has emerged as an attractive alternative. Several commercial processors provide direct hardware support for TM, including Intel's Haswell [3] and IBM's Blue Gene/Q [4], zEnterprise EC12 [5], and Power8 [6]. There are proposals for adapting TM to clusters of GPUs [7–9].

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 speculative execution requires *data version management* mechanism.

Many TM systems have been designed using the transactional concept in persistent memories as well as non-persistent (volatile) memories. Those designed TM systems can be distinguished on how they implement data version management. This distinction is true for TM systems implemented in hardware, called hardware TMs (HTMs) [10–13], as well as implemented in software, called software TMs (STMs) [14–16].

In this paper, we present a technique that improves on the existing data version management mechanisms used in both persistent and non-persistent TM systems. Essentially, a *versioning* mechanism handles *data versions*, i.e., 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 persistent/non-persistent 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. Figures 1 and 2 depict how a transaction T_x is executed using eager and lazy versioning in persistent and non-persistent (volatile) TM systems, respectively. Due to the working principle, in both the systems, lazy versioning makes aborts fast but penalizes commits, whereas eager versioning makes commits fast but penalizes aborts.

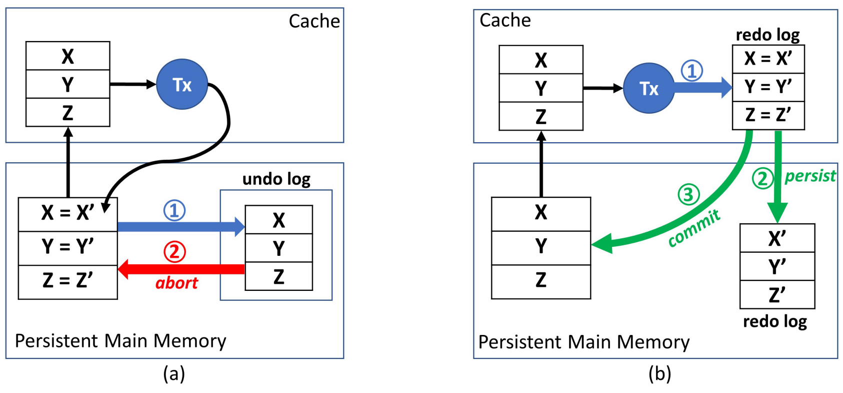

Figure 1. An illustration of how a transaction T_x is executed in persistent memory using (**a**) eager versioning and (**b**) lazy versioning. Figure (**a**) depicts two kinds of operations in eager versioning. The first operation is to copy the data from original memory locations to a log area (called *undo* log) in the persistent main memory and the second is 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 three kinds of operations in lazy versioning. The first operation is to copy the data from original memory locations to a log area (called *redo* log). Transaction T_x updates on this redo log. The second operation is to persist the redo log in persistent memory and the third operation is to copy the updated data back from redo log to the original memory locations. The second and third operations are required only in case T_x commits. If T_x aborts, the data in the redo log area in cache is simply discarded.

Although both eager and lazy versionings are studied heavily in the literature (details in Section 2) for both persistent [17–20] and non-persistent TM systems [10–16,21], whether to use eager or lazy versioning is still in hot debate for both the systems. In fact, there is no study in persistent/non-persistent TM systems that elaborates the performance gap between eager and lazy versioning with comprehensive practical evaluations, with one notable exception [22] which elaborates on the performance gap partially only for persistent TM systems. The conclusion from [22] is that lazy versioning is appropriate for write-dominated workloads and high contention scenarios whereas eager versioning is appropriate for read-dominated workloads and low contention scenarios. However, to improve performance using lazy or eager versioning, a priori knowledge on the workload as well as the contention scenario is needed.

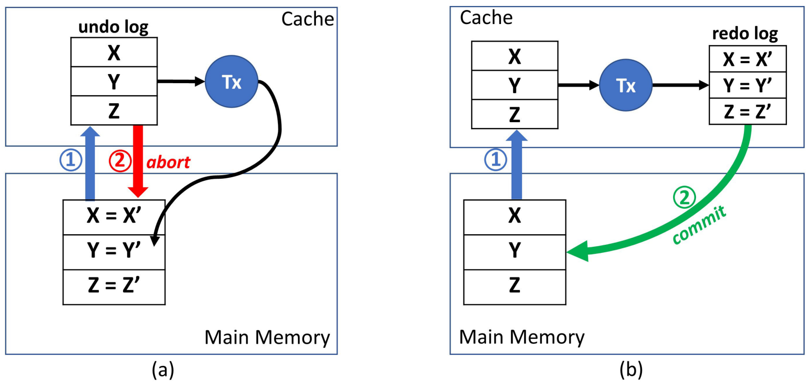

Figure 2. An illustration of how a transaction T_x is executed in non-persistent memory using (**a**) eager versioning and (**b**) lazy versioning. Figure (**a**) depicts two kinds of operations in eager versioning. The first operation is to copy the data from original memory locations to a log area (called *undo* log) in cache. Transaction T_x then performs in-place updates to the original memory locations. Now, the second operation is 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 operation is to copy the data from original memory locations to a log area (called *redo* log) in cache. Transaction T_x updates on this redo log. Then, the second operation is to copy updated 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.

We conducted a study to validate whether the conclusion from [22] also applies to non-persistent TM systems. Particularly, we executed *genome* and *kmeans* benchmarks from STAMP benchmark suite [23] using lazy and eager versioning and measured performance through execution time and number of aborts under varying number of threads (Refer to Section 6 for details on the experimental setup and implementation). The results obtained are shown in Figure 3.

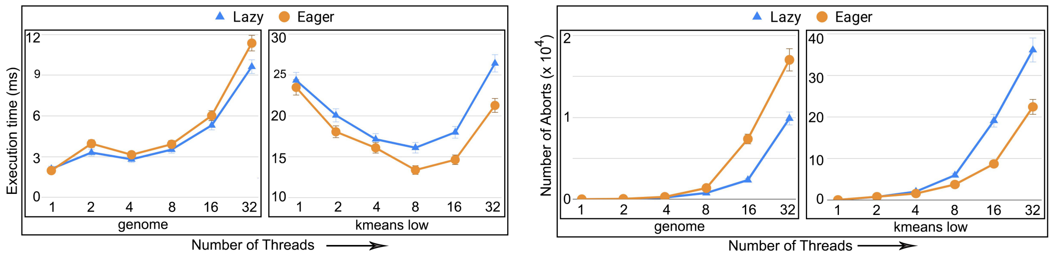

Figure 3. An illustration of performance discrepancies in execution time (**left**) and number of aborts (**right**) in *genome* and *kmeans* benchmarks using eager and lazy versioning in non-persistent TM systems.

The results show that lazy versioning performs well for *genome* whereas the opposite is true for *kmeans*, which is in line of the conclusion drawn in [22]. Again, this discrepancy in performance 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. This raises the question of how to choose the versioning method that is appropriate for an application, without a priori knowledge on the workload and contention scenario. However, this is a challenging issue, particularly due to the fact that (i) to select an appropriate versioning, a priori knowledge on the workload (write-dominated or read-dominated) and contention scenario (low or high) is needed, and (ii) such knowledge is difficult to obtain prior to runtime.

1.1. Contributions

In this article, 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. We provide two different models of ADAPTIVE, one for persistent TM systems and another for non-persistent TM systems. We reported a preliminary version of ADAPTIVE for persistent TM systems in [24]. We reported a preliminary version of ADAPTIVE for non-persistent TM systems in [25]. This article combines and extends those preliminary versions with a full set of experimental results. For the experimental evaluation in both the systems, we incorporated ADAPTIVE in the latest TinySTM implementation [26,27] and ran experiments against a diverse set of TM benchmarks [26–28]. 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 [23,29]). We measured the performance of ADAPTIVE w.r.t. four 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. 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.
- **total number of data movements (for persistent TM systems only):** the total number of movements of data to and from the original memory addresses. The execution time of transactions in persistent memory is directly affected by the number of reads and writes to the PM. The total number of reads and writes to the persistent memory addresses can also be defined as the total number of movements of data to and from the memory addresses. Thus, minimizing the total number of data movements decreases the total execution time of the transactions in PM.
- **total number of stores to persistent memory (for persistent TM systems only):** the total number of writes to the persistent memory addresses. The motivation behind this performance metric is as follows. It has been heavily advocated that persistent memories significantly outperform traditional dynamic random access memories (DRAMs) due to low standby power, higher memory density, and much lower cost/bit [30,31]. However, persistent memories suffer from the *write endurance* problem, i.e., every persistent memory (PM) unit can sustain a very limited number of writes (i.e., stores) before it wears-out. Minimizing the total number of stores to the PM helps in mitigating the write-endurance problem in PM.

In persistent TM systems, the results suggest that, when using *lazy* versioning with encounter time locking (the two variants *encounter-time-locking* and *commit-time-locking* of the lazy versioning are described in Section 6.1), ADAPTIVE achieves up to 1.21× better performance (i.e., 17% less execution time) than eager versioning and up to 1.27× better performance (i.e., 21% less execution time) than lazy versioning. When using *lazy* versioning with commit time locking, ADAPTIVE achieves up to 1.39× better performance (i.e., 28% less execution time) than eager versioning and up to 1.5× better performance (i.e., 33% less execution time) than lazy versioning. Moreover, ADAPTIVE has up to 240× and 17× less number of aborts compared to that in *eager* and *lazy* versioning, respectively.

In non-persistent TM systems, the results show that, when using lazy versioning with encounter time locking, 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 commit time locking, 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 both persistent memory and non-persistent TM systems. 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, and provide two models of ADAPTIVE that are suitable for persistent memory and non-persistent TM systems, respectively.
- (**Section** 5) We discuss the limitations of basic design of ADAPTIVE for non-persistent TM system and present two optimizations.
- (**Section** 6) We evaluate experimentally the performance of ADAPTIVE in both persistent and non-persistent TM systems using five micro-benchmarks and 8 complex benchmarks from STAMP and STAMPEDE suites, report the results, and provide observations.

1.2. Organization

We discuss related work in Section 2. We discuss the memory model and some preliminaries in Section 3. We present our basic adaptive versioning approach in Section 4 and provide two models suitable for persistent and non-persistent TM systems, respectively. We discuss the limitation of the design of basic ADAPTIVE in a non-persistent TM system in Section 5 and present some optimizations. We evaluate the performance of ADAPTIVE in both the systems in Section 6. Finally, we provide concluding remarks in Section 7 with a short discussion on possible future work.

2. Related Work

We discuss here the persistent and non-persistent TM systems proposed in the literature, the use of eager and lazy versioning in those systems, and the conflict detection and resolution mechanisms. Table 1 summarizes the advantages and disadvantages of eager and lazy versioning in persistent and non-persistent TM systems.

Table 1. A comparison of *eager* and *lazy* versioning in persistent and non-persistent TM systems *(the row in bold text is only for persistent TM systems)*.

Constraint	**Eager Versioning**	**Lazy Versioning**
Memory Update	performs in-place memory update	updates are written to memory at the commit time
Reading Overhead	allows to read most recent data directly from in-place memory	reads are intercepted and redirected to the redo log area to read recent uncommitted data
***Persist Ordering* (only for persistent TMs)**	**requires to ensure persist ordering for each memory write in a transaction [19]**	**requires only one persist ordering for each transaction [19,21]**
Data Movement	transaction aborts are costly as the memory updates need to be rolled back to consistent state using undo log	transaction commits are costly as the updates need to be written back to original memory using redo log

2.1. Non-Persistent (Volatile) TM Systems

There are several previous studies in non-persistent TM systems, e.g., [10–12,14–16]. Table 2 shows the versioning mechanisms used in some widely-popular non-persistent TM systems. The previous studies used either eager or lazy versioning individually. There is no work that elaborates on the impact of using eager and lazy versioning on the performance of non-persistent TM systems. In fact, the majority of well-known non-persistent TM systems make contradictory conclusions on whether to use eager versioning or lazy versioning. For example, consider two widely popular HTM implementations LogTM [12] and UTM [10]. They advocate that TM should ideally use eager versioning and eager conflict detection (discussed in Section 3.5) since in eager versioning transaction commits are faster than transaction 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 another widely popular HTM implementation TCC [11]. They use lazy versioning and lazy conflict detection. Other HTMs such as VTM [13] and LTM [10] 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 of versioning and conflict detection methods, e.g., [14–16].

The other line of work is Hybrid TM systems (HyTMs) [32–37] where transactions are dynamically executed either in HTM or STM implementation. However, it is challenging and complicated to manage the concurrent execution of both hardware and software transactions in HyTM [33]. Therefore, to address this, in 2007, Lev et al. [38] proposed Phased Transactional Memory (PhTM) system to allow the execution of transactions in phases such that each phase is run in the same mode (HTM or STM) and the switching between them is supported seamlessly. PhTM benefits as it does not require coordination between transactions running in different modes. Recently, Carvalho et al. presented an improved version (PhTM*) [39] and its effectiveness in [40] by avoiding unnecessary switches to software mode. Both approaches, HyTM and PhTM, focus on getting better performance by dynamically switching between the HTM and STM implementations. This is different from the approach we present in this article since we deal with dynamically switching between eager and lazy versioning method at runtime to improve performance of a TM implementation, whereas the HyTM and PhTM approaches deal with switching between HTM and STM implementations.

Table 2. Versioning and conflict detection mechanisms used in some non-persistent TM systems.

		Versioning	
		Lazy	**Eager**
Conflict	**Lazy**	TCC [11], Norec [15], RSTM [14], SwissTM [16]	None
	Eager	UTM [10], VTM [13], RSTM [14], SwissTM [16]	UTM [10], LogTM [12], RSTM [14]

2.2. Persistent TM Systems

The performance gap of using eager and lazy versioning is relatively well-studied in persistent TM systems. The most closely related work is due to Wan et al. [22], where they empirically evaluated eager and lazy versioning on the open source non-volatile memory library (NVML), PMDK, Ref. [41] for some constrained workloads, and suggested that "*one versioning 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 [22]. The other works mostly proposed methods through either eager or lazy versioning, and there is no work that elaborates the performance gap between eager and lazy versioning. For example, Coburn et al. [18] suggested an STM implementation for persistent memory, called NV-Heaps, using eager versioning. Volos et al. [19] suggested a TinySTM [26,27] variation, called Mnemosyne, for persistent memory using lazy versioning. NV-Heaps [18] and Mnemosyne [19] drew absolutely opposite conclusions on whether eager or lazy versioning is better for persistent memories. The former prefers to use eager versioning, and the latter opts to use lazy versioning. Furthermore, many other persistent TM systems such as [20,42] suggested using eager versioning.

Avni et al. [43] studied HTM-based transactions using lazy versioning. DUDETM [21] incorporates lazy versioning in their design where a transaction first runs in volatile memory using any HTM or STM implementation and produces a redo log for that transaction. The redo log is then flushed to persistent memory satisfying atomicity of data and then modify the original data in persistent memory according to the persistent redo log. Notice that this approach is different from ours and needs a shared shadow memory, besides persistent memory where that data is. Genc et al. [44] proposed a low overhead HTM compatible persistent transactional system, called Crafty, using eager versioning. Joshi et al. [45] proposed a persistent HTM system providing a hardware support for lazy versioning to reduce the performance overhead compared to software based implementations. Recently, Castro et al. [46] studied the scalability issues with the experimental results on Intel Optane DC [47] persistent memory and proposed scalable persistent hardware transactions (SPHT). Baldassin et al. [48] have introduced a phase based persistent TM system, NV-PhTM, where the transaction execution mode is switched between two phases, HTM and STM. The best execution mode among the two is selected according to the application's characteristics. This is different than our approach where we switch between the versioning methods within a TM system instead of switching between hardware or software transaction execution mode.

The other related works in persistent memory study latency, scalability and ordering constraints problems. Krishnan et al. [49] proposed a persistent TM system, called TimeStone, that has minimal writes and low memory footprints. Gu et al. [50] presented a read-friendly persistent TM system, called Pisces, that maintains up to two versions of the data using dual-version concurrency control (DVCC) protocol and provides non-blocking reads. Kolli et al. [51] studied the ordering constraint problem for transactions in persistent memories and proposed deferred commit transactions (DCT) to achieve minimal ordering constraints. Lu et al. [52] proposed a system for reducing ordering constraints among persistent writes by distributing the commit status of a transaction among the data blocks. In [53], Memaripour et al. studied the latency overhead in byte addressable non-volatile memories and propose Kamino-Tx without requiring copying of data in the critical path.

Other several recent papers, e.g., [17,20,54–58], provided techniques to improve the time to log the data (e.g., through coalescing, through persistent cache, through hardware support, through eager+lazy versioning methods, etc.) for both undo and redo logs. However, our focus is on taking a different approach of dynamically switching between eager and lazy versioning at runtime to exploit advantages of both the versioning methods and our extensive experimental evaluation detailed in Section 6 confirms this exploitation. Our approach obviates the need of a priori knowledge on the workload as well as contention scenario to decide whether to use eager versioning or lazy versioning.

3. Model and Preliminaries

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 [59], of storing multiple versions.

3.1. Persistent Memory Model

We consider a computer system with unlimited persistent memory, many processing cores, and no hard disk drive (HDD). All persistent memory is cache-able and caches are volatile and coherent. The system may include limited size DRAM (but we do not assume its necessity). We assume that all the writes of a committed transaction can be accommodated in the volatile cache, i.e., once a transaction commits but before the commit is reflected in original memory locations in persistent memory, all its newly modified data is in volatile cache. The system restarts and resumes its computation after experiencing failures/crashes. Therefore, the task after restart is to bring the data to a consistent state, removing effects of uncommitted transactions and applying the missing effects of the committed ones. In the experimental evaluation, we simulate crashes by periodically wiping out the volatile logs, and use the data stored in undo or redo logs in persistent memory to recover consistency. We employ a function that checks and maintains consistency while under execution.

3.2. Non-Persistent Memory Model

We consider a computer system with volatile shared main memory, many processing cores, and a HDD. All shared main memory is cache-able 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 [26,27].

3.3. Eager Versioning

Eager versioning is supported through so-called *undo logs*. Undo logs are stored in cachable 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. Figures 1a and 2a illustrate eager versioning in persistent and non-persistent TM systems, respectively.

3.4. Lazy Versioning

Lazy versioning is supported through so-called *redo logs*. The operation of lazy versioning is slightly different in persistent and non-persistent TM systems. In non-persistent memories, redo logs are stored only in cache. But, in persistent memories, the redo logs are also persisted in the persistent memory before updating on original memory locations.

Using lazy versioning in non-persistent memories, a transaction copies all the data that it is going to write from original memory location to a redo log area, appends all its data updates to that log area, and then writes the updated data back to the original memory locations when the transaction commits. In persistent memories, the transaction additionally copies the updated data from redo log area in cache to the redo log area in persistent memory before writing back to the original memory locations. If the transaction fails, the updates in log area in cache are simply discarded. Therefore, the writing of data in redo log back to the original memory locations happens only when transaction commits. Figures 1b and 2b illustrate lazy versioning in persistent and non-persistent memories, respectively.

3.5. Conflict Detection and Resolution

Conflict detection and resolution comes into play when concurrently executing transactions on both persistent/non-persistent TM systems read/write the same memory locations and certain combinations of read/write patterns cannot allow multiple transactions to proceed to commit. *Conflict detection* mechanism signals such 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. Conflict detection depends on whether lazy versioning is used or eager versioning. Table 2 illustrates some existing non-persistent TM systems that use lazy versus eager conflict detection with the versioning mechanism (lazy or eager) they use. For example, TCC [11] uses lazy conflict detection with lazy versioning and LogTM [12] uses eager conflict detection with eager versioning. *Contention management* technique is then used to decide on which conflicting transaction(s) to continue and which transaction(s) to wait (or abort and restart) the execution. This is typically done through a contention management strategy. There is a huge amount of work in this area giving many different strategies with and without provable properties on the guarantees they provide, e.g., [26,60–68].

3.6. Supporting Durable Transactions in TinySTM

We implemented durable transactions using TinySTM [26], a widely used lightweight STM implementation, as follows. For eager versioning, we maintain a undo log in persistent memory. When a transaction starts, each variable accessed by the transaction is added to the log before any modification is performed to it. Any update to the variable during the execution of the transaction is written directly to the variable's original address. When the transaction commits, respective log records for the transactions are freed and the memory is made available to use by other transactions. If the transaction aborts, all the changes made by the transaction to the variables are written back with the previous values stored in the respective undo logs. Then the log records are flushed.

For lazy versioning, when a transaction starts, all the variables accessed by the transaction are loaded to volatile cache and modified. The new (or updated) values written by the transactions are then added to a redo log in persistent memory and also buffered in the volatile cache before the transaction commits. When the transaction commits, the new values are written back to the original memory addresses and the log records are flushed. We attach a timestamp based version number to each transactional log to make sure that the last committed value is used in the restart process.

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 4 compares ADAPTIVE with eager and lazy versioning. The pseudocode of ADAPTIVE is given in Algorithm 1.

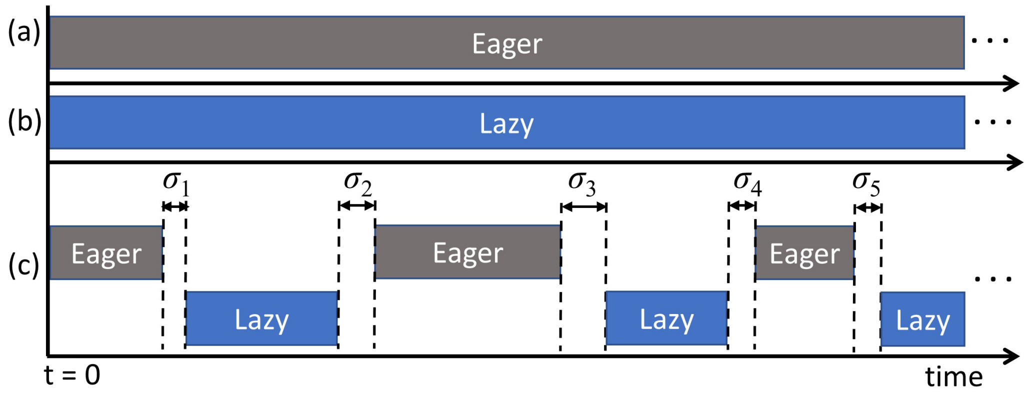

Figure 4. An illustration of (**a**) eager, (**b**) lazy, and (**c**) basic adaptive versioning. The time gap σ_* while switching from eager (lazy) to lazy (eager) is to let finish executing in-flight transactions. This helps in avoiding potential data versioning inconsistencies.

Algorithm 1: ADAPTIVE for a transaction T at any time $t \geq 0$

```
1  $N_{Ecommit} \leftarrow$ number of transaction commits until $t$ executed using *Eager* versioning;
2  $N_{Lcommit} \leftarrow$ number of transaction commits until $t$ executed using *Lazy* versioning;
3  $N_{Eabort} \leftarrow$ number of transaction aborts until $t$ executed using *Eager* versioning;
4  $N_{Labort} \leftarrow$ number of transaction aborts until $t$ executed using *Lazy* versioning;
5  $N_{commit} \leftarrow N_{Ecommit} + N_{Lcommit}$; $N_{abort} \leftarrow N_{Eabort} + N_{Labort}$;
6  $V_{cur} \leftarrow$ current versioning method for transactions running in a thread;
   // $V_{cur} \in \{Eager, Lazy\}$
7  $tms \leftarrow$ Transactional Memory System type; // $tms \in \{persistent, non\_persistent\}$
   /* Initially, at $t = 0$, choose the versioning method based on
      workload size (if available), otherwise randomly */
8  if $N_{commit} + N_{abort} = 0$ then
9      if information on read and write sets is available then
10         $Wset(T) \leftarrow$ write set of transaction $T$;
11         $Rset(T) \leftarrow$ read set of transaction $T$;
12         if $|Wset(T)| > |Rset(T)|$ then
13             $V_{cur} \leftarrow Lazy$; Execute $T$ using *Lazy* versioning;
14         else $V_{cur} \leftarrow Eager$; Execute $T$ using *Eager* versioning;
15     else Execute $T$ using either eager or lazy versioning;
   /* At any time $t > 0$, choose the versioning method based on
      abort-to-commit ratio (ACR) */
16 else if $tms == persistent$ then // ADAPTIVE for persistent memories
17     $AAR \leftarrow \frac{N_{abort}}{N_{commit}+N_{abort}}$, $AAR_{Eager} \leftarrow \frac{N_{Eabort}}{N_{Ecommit}+N_{Eabort}}$;
18     $ACR_{Eager} \leftarrow \frac{N_{Eabort}}{N_{Ecommit}}$, $ACR_{Lazy} \leftarrow \frac{N_{Labort}}{N_{Lcommit}}$;
19     if $(AAR \geq \frac{2}{3}) \vee ((ACR_{Eager} > ACR_{Lazy}) \wedge (AAR_{Eager} \geq \frac{2}{3}))$ then
20         Execute $T$ using *Lazy* versioning;
21     else
22         Execute $T$ using *Eager* versioning;
23 else if $tms == non\_persistent$ then // ADAPTIVE for non-persistent memories
24     $ACR_{Eager} \leftarrow \frac{N_{Eabort}}{N_{Ecommit}}$, $ACR_{Lazy} \leftarrow \frac{N_{Labort}}{N_{Lcommit}}$;
25     if $((V_{cur} = Eager) \wedge (ACR_{Eager} > \frac{1}{2}))$ then
26         $V_{cur} \leftarrow Lazy$;
27     else if $((V_{cur} = Lazy) \wedge (ACR_{Lazy} < 2))$ then
28         $V_{cur} \leftarrow Eager$;
29     Execute $T$ using $V_{cur}$ versioning method;
```

4.1. High Level Overview

The high level idea in ADAPTIVE is to switch the versioning method depending on performance. That is, if the versioning method currently used is hampering the performance, then switch the versioning to improve the performance. The fundamental question is how to identify and measure an indicator that reflects appropriately the effect of the versioning method on performance. Fortunately, in TM systems, if the number of aborts are increasing compared to the number of commits, then it is be a valid indicator of performance degradation due to the versioning method currently in use. Therefore, we pick abort to commit ratio (ACR) as a performance indicator. ACR has also been used quite heavily in the TM literature as a vital indicator of performance, for example, see [69].

Formally, ACR can be defined at any time $t > 0$ as follows:

$$ACR = \frac{N_{abort}}{N_{commit}}, \tag{1}$$

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. 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.

Let T be a transaction that comes to the system at time $t \geq 0$; we assume that the execution starts at time $t_0 = 0$. 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

$$N_{abort} = N_{Eabort} + N_{Labort}.$$

The concept in ADAPTIVE is to decide on which versioning method to use for executing T based on the parameters $N_{Ecommit}, N_{Lcommit}, N_{Eabort}$, and N_{Labort} learned from the system at runtime. However, if T comes to the system at time $t_0 = 0$, we have all $N_{Ecommit}, N_{Lcommit}, N_{Eabort}$, and N_{Labort} zero. We treat this as a special case. In the special case of $t_0 = 0$, 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.

For any transaction T arriving at time $t > t_0$, we have two different models of ADAPTIVE described in the following sub-sections, one for persistent memories and another for non-persistent memories.

4.2. ADAPTIVE *for Persistent Memories*

The idea we employ in ADAPTIVE for persistent memories is to compute the number of data movements for eager and lazy versioning, separately, and switch between these methods when the data movement increases. Ideally, we would like to use the versioning method that gives optimum data movement performance for any specific workload. We use the following notions. Let N be the total number of transactions in any workload. When the workload finishes execution and all transactions commit, we have $N_{commit} = N$ number of commits and $N_{abort} \geq 0$ number of aborts (if each transaction commits without even aborting a single time, then $N_{abort} = 0$, otherwise $N_{abort} > 0$). Suppose each transaction T has read write set $RW(T)$ of size S.

If T comes to the system at time $t > t_0$ after at least a transaction finishes executing one time (irrespective of whether that transaction aborts or commits), then it is executed based on the following parameters computed in ADAPTIVE from time $t = 0$ until time t.

i. $AAR = \frac{N_{abort}}{N_{commit}+N_{abort}}$; *average abort ratio* of transactions in ADAPTIVE (using total aborts in both eager and lazy versioning).

ii. $AAR_{Eager} = \frac{N_{Eabort}}{N_{Ecommit}+N_{Eabort}}$; *average abort ratio* of transactions in ADAPTIVE executed using eager versioning.

iii. $ACR_{Eager} = \frac{N_{Eabort}}{N_{Ecommit}}$; *abort to commit ratio* of transactions in ADAPTIVE executed using eager versioning.

iv. $ACR_{Lazy} = \frac{N_{Labort}}{N_{Lcommit}}$; *abort to commit ratio* of transactions in ADAPTIVE executed using lazy versioning.

At any time $t \geq 0, 0 \leq AAR \leq 1$ and $0 \leq AAR_{Eager} \leq 1$.

At any time $t > t_0$ in ADAPTIVE, T is executed using lazy versioning if (i) $AAR \geq \frac{2}{3}$ or (ii) $ACR_{Eager} > ACR_{Lazy}$ and $AAR_{Eager} \geq \frac{2}{3}$. Otherwise, T is executed using eager versioning. We call the value $\frac{2}{3}$ *switching threshold* and we describe later how this switching threshold $\frac{2}{3}$ is computed. The motivation behind using $\frac{2}{3}$ as switching threshold in ADAPTIVE for persistent memories is that it works on all the benchmarks we experimented our framework against. We now discuss how the switching threshold is computed.

Computation of Switching Threshold in Persistent Memories

The computation of switching threshold is based on the total movements of data from one memory location to the other (i.e., total number of loads and stores). The motivation behind using this metric for the computation of switching threshold is that the time spent by a transaction to load and store data from and to the memory addresses plays significant role in total execution time. Our objective in the design of ADAPTIVE is to dynamically switch between the two versioning methods to obtain less number of data movements, and in return minimize execution time.

Let W_{Eager} be the total number of operations of moving data in eager versioning (i) from the original persistent memory locations to the undo log area and (ii) from the undo log area back to the original persistent memory locations. The first kind of moves are shown as ① in Figure 1a and the second kind of moves are shown as ② in Figure 1a. The first kind of moves are always done in eager versioning and the second kind of moves are done only when the transaction aborts. Therefore,

$$W_{Eager} = (N_{commit} + 2N_{abort}) \cdot S \tag{2}$$

Let W_{Lazy} be the total number of operations of moving data in lazy versioning (i) from the original persistent memory locations to the redo log area (in volatile cache), (ii) from the redo log area (in volatile cache) to persistent memory locations to persist the redo log, and (iii) finally, writing the data back to the original persistent memory locations either from redo log area in persistent memory after restart or from redo log area in volatile cache. The first kind of moves are shown as ① in Figure 1b, and the second and third kind of moves are shown as ② and ③ in Figure 1b, respectively. The first kind of moves are always done in lazy versioning and the second and third kind of moves are done only when the transaction commits. Therefore,

$$W_{Lazy} = (3N_{commit} + N_{abort}) \cdot S \tag{3}$$

Notice that a transaction can run using either eager or lazy versioning when $W_{Eager} = W_{Lazy}$ as the selection of a versioning method does not have impact on the total number of movements. Therefore, from Equations (2) and (3), we have that

$$\begin{aligned} (N_{commit} + 2N_{abort}) \cdot S &= (3N_{commit} + N_{abort}) \cdot S && (4) \\ N_{commit} + 2N_{abort} &= 3N_{commit} + N_{abort} && (5) \\ N_{abort} &= 2N_{commit} && (6) \end{aligned}$$

Also, we have that $N \leq N_{abort} + N_{commit}$. This implies that

$$\frac{N_{abort}}{N} + \frac{N_{commit}}{N} \geq 1 \tag{7}$$

$$\frac{2N_{commit}}{N} + \frac{N_{commit}}{N} \geq 1 \tag{8}$$

$$\frac{N_{commit}}{N} \geq \frac{1}{3} \tag{9}$$

Therefore, $\frac{N_{abort}}{N} < \frac{2}{3}$. That is, if the value of N_{abort} is such that $\frac{N_{abort}}{N}$ is higher than or equals to $\frac{2}{3}$, then $W_{Eager} > W_{Lazy}$. Thus, ADAPTIVE switches execution to lazy versioning when $\frac{N_{abort}}{N} \geq \frac{2}{3}$ and stays with eager versioning, otherwise.

4.3. *Computation of Total Number of Stores to Persistent Memories*

The total number of writes (i.e., stores) to the persistent memories are different when transactions are executed using different versioning methods.

In eager versioning, transactional (undo) logs are stored in the persistent memory and in-place memory updates are performed. Each memory location accessed by a transaction is added to the log. Thus, in default, there are two stores for each memory location accessed by a transaction. Additionally, if the transaction aborts, it needs to be rolled back to the previous consistent state using the undo log, which requires two additional stores to the persistent memory. Let ST_{Eager} be the total number of stores to the persistent memory in eager versioning, N_{commit} be the total number of commits and N_{abort} be the total number of aborts, then,

$$ST_{Eager} = (2N_{commit} + 2N_{abort}) \cdot S \tag{10}$$

where S is the size of RW set of the transactions.

In lazy versioning, all the computations are performed in volatile cache. If a transaction commits, then the changes are first persisted to the transactional (redo) logs and then updated to the original memory locations. That means, in lazy versioning, only committing transactions account for PM stores. Let ST_{Lazy} be the total number of stores to the persistent memory in lazy versioning and N_{commit} be the total number of commits, then,

$$ST_{Lazy} = (2N_{commit}) \cdot S \tag{11}$$

From Equations (10) and (11), we can see that the total number of PM stores in lazy versioning is always less than that in eager versioning. So, from the perspective of minimizing total stores to persistent memories, lazy versioning seems better. However, this metric alone can not guarantee the better performance of transactions. Thus, we also consider other performance metrics such as execution time, abort rate and total data movements.

4.4. ADAPTIVE *for Non-Persistent Transactional Memories*

The idea in the design of ADAPTIVE for non-persistent memories is to compute the total time spent by transactions executing using eager and lazy versioning, separately, and switch between the versioning methods when the execution time increases. Ideally, again, we would like to use the versioning method in ADAPTIVE that gives optimum performance in terms of execution time for any specific workload. From the working principle of a TM system, we can see that a transaction spends significant amount of time on moving data between the original memory location and the log areas in addition to the constant computation time. Figure 2 illustrates the working principle of a TM system. Moreover, it is likely that the total execution time increases with the increase in total number of aborts requiring more number of transaction restarts. Thus, we use abort to commit ratio (ACR) in the design of ADAPTIVE for non-persistent memories.

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 as follows.

$$ACR_{Eager} = \frac{N_{Eabort}}{N_{Ecommit}} \tag{12}$$

$$ACR_{Lazy} = \frac{N_{Labort}}{N_{Lcommit}} \tag{13}$$

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.

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 $V_{cur} = Eager$. If $ACR_{Eager} > Threshold_{Eager}$, then V_{cur} is switched to *Lazy* (i.e., $V_{cur} \leftarrow Lazy$) and T will be executed using lazy versioning.
- Suppose the versioning method currently used is $V_{cur} = Lazy$. If $ACR_{Lazy} < Threshold_{Lazy}$, then V_{cur} is switched to *Eager* (i.e., $V_{cur} \leftarrow Eager$) and T will be executed 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 ① in Figure 2a and the second kind of operations are shown as ② in Figure 2a. 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,

$$\tau_{Eager} = (N_{commit} + 2N_{abort}) \cdot \alpha \tag{14}$$

Similarly, consider the case of executing T using lazy versioning. Let τ_{Lazy} be the total amount of time spent while (i) versioning data from the original memory locations to the redo log area and (ii) writing the data from the redo log area back to the original memory locations. The first kind of operations are shown as ① in Figure 2b and the second kind of operations are shown as ② in Figure 2b, respectively. The first kind of operations are always done in lazy versioning and the second kind of operations are done only when the transaction commits. That means, for a committed transaction, data movement is performed two times. Therefore, for lazy versioning,

$$\tau_{Lazy} = (2N_{commit} + N_{abort}) \cdot \alpha \tag{15}$$

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.,

$$\frac{N_{abort}}{N_{commit}} \leq \frac{1}{2} \tag{16}$$

Thus, we get $Threshold_{Eager} = \frac{1}{2}$ and switch to lazy versioning when $ACR_{Eager} > \frac{1}{2}$.

Equation for τ_{Lazy} suggests that the lazy versioning performs better until $2N_{commit} \leq N_{abort}$; i.e.,

$$\frac{N_{abort}}{N_{commit}} \geq 2 \tag{17}$$

Then, we get $Threshold_{Lazy} = 2$ and switch to eager versioning when $ACR_{Lazy} < 2$.

4.5. Contention Management

A transaction T is said to be *conflicted* with another transaction T_j in two cases: (i) $Rset(T)$ shares at least a memory location with $Wset(T_j)$, i.e., $Rset(T) \cap Wset(T_j) \neq \emptyset$, and (ii) $Wset(T)$ shares at least a memory location either with $Rset(T_j)$ (i.e., $Wset(T) \cap Rset(T_j) \neq \emptyset$) or with $Wset(T_j)$ (i.e., $Wset(T) \cap Wset(T_j) \neq \emptyset$). Any contention management technique requires at least $x - 1$ transactions out of the $x \geq 2$ conflicted transactions to abort. There has been an extensive study on contention management and several techniques with different performance properties are available, e.g., [26,27,60–68]. We use the following strategies for resolving conflict of a transaction T with transaction T_j, which are discussed in detail in [26,27,65].

- **suicide:** T aborts and restarts immediately.
- **kill** (aka **aggressive**): T kills T_j and continues its execution.
- **delay:** T aborts immediately but waits until the contended memory location is released before restarting. This increases the chance of the transaction to commit with no interruption upon retry, but may increase total execution time.
- **back-off:** T uses an exponential back-off mechanism to resolve conflict.

4.6. Time Barrier Requirement and Design

The ideal scenario in ADAPTIVE is to let each transaction T run Algorithm 1 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 current versioning method for executing the transactions in S_j is $V_{cur} = Lazy$ and the transaction T satisfies for switching the versioning method to $V_{new} = Eager$. 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 Figure 4). The pseudocode is given in Algorithm 2. The barrier reduces total number of aborts but due to a time delay before switching, it may increase total execution time [24]. We provide a *better time barrier* design appropriate for non-persistent TM systens (described in Section 5) that will minimize this overhead.

Algorithm 2: Basic Time Barrier Design

```
1  $T \leftarrow$ new transaction arrived for execution;
2  $S_j \leftarrow$ set of other transactions arrived before $T$;
3  $V_{cur} \leftarrow$ versioning method for currently running transactions ;
4  $V_{new} \leftarrow$ new versioning method computed for the transaction $T$;
5  if ($V_{new} \neq V_{cur}$) then        // i.e., $V_{cur}$ and $V_{new}$ are complement of each
   other in the set $\{Eager, Lazy\}$
6      if (there is no transaction $T_j \in S_j$ such that $T_j$ is executing using $V_{cur}$ when $T$ wants
       to execute) then
7          $V_{cur} \leftarrow V_{new}$;
8          Start executing $T$ using $V_{new}$;
9      else
10         Wait until each transaction $T_j \in S_j$ finish execution;
```

5. Optimizations on Basic Adaptive Versioning

5.1. Limitation of Basic ADAPTIVE in Non-Persistent Memories

In basic ADAPTIVE, no two transactions can execute simultaneously with different versioning methods, i.e., if a new transaction tries to execute with a different versioning method than the currently executing one, the *basic time barrier* prevents it to execute simultaneously. The design of *basic time barrier* in ADAPTIVE requires a transaction T to wait until all the previous transactions finish their executions if T wants to execute with different versioning method than the previous transactions. This also prevents to execute two non-conflicting transactions concurrently with different versioning methods. Thus, to alleviate these problems, we provide two optimizations to basic ADAPTIVE. The first optimization is on time barrier design. The second optimization is on switching mechanism.

5.2. Better Time Barrier Design

The pseudocode of the better time barrier design is given in Algorithm 3. The objective of better time barrier design is to increase concurrency as opposed to the basic time barrier design. For this, we allow each transaction to start its execution (with possibly new versioning method) as soon as it becomes ready rather than waiting for other in-flight transactions to complete. Figure 5 illustrates the idea of better time barrier design. Consider a transaction T. Let S_j be a set of transactions arrived before T. Suppose current versioning method for executing transactions in S_j is $V_{cur} = Lazy$ and new versioning method computed for transaction T is $V_{new} = Eager$. Suppose the versioning method changed to *Eager* from *Lazy* after the transactions in S_j started execution (but not completed yet) and before T starts execution. In the basic time barrier design, T has to wait until all the transactions in S_j finish execution. In the better time barrier design, we ask T to start execution as soon as it is ready. If T does not conflict with any transaction in S_j, then T continues its execution until it commits, otherwise, T aborts. In order to detect the possible conflict of T with the transactions in S_j, we add a 1-bit *modify flag* to each memory address that is going to be updated by the transactions in S_j. The modify bit associated to a memory address is set to 1 at the start of a transaction if it is going to be updated and is reset back to 0 at the time of commit. If T conflicts with $T' \notin S_j$, it is handled as per the contention management technique adapted in the design (e.g., suicide, kill etc).

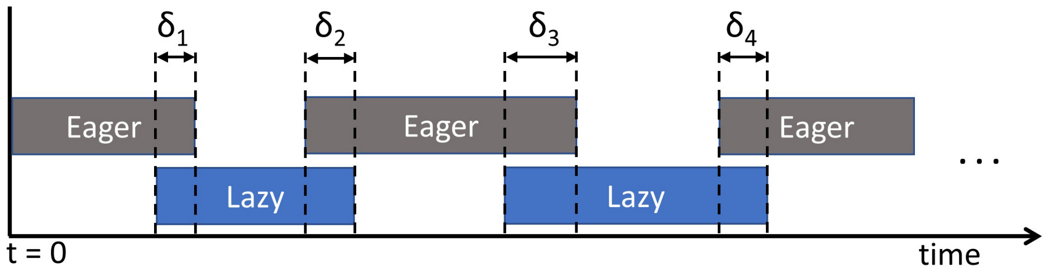

Figure 5. An illustration of the better time barrier design. The interval δ_* between *Eager* and *Lazy* represents the time taken by in-flight transactions to finish their executions after versioning method is switched. The new transaction that do not conflict with transactions using previous versioning can execute concurrently with in-flight transactions.

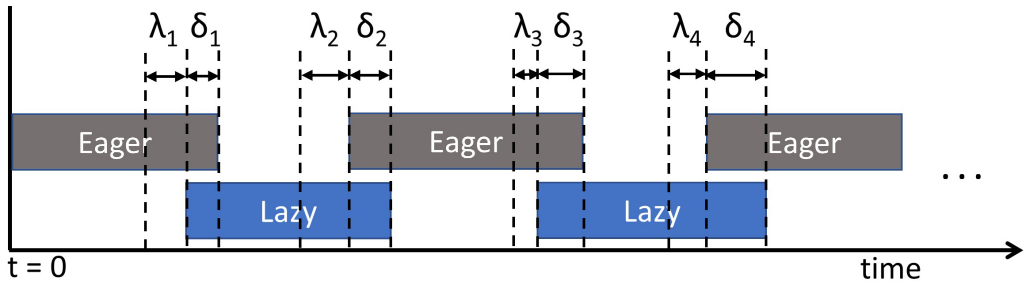

Figure 6. An illustration of the better switching mechanism. λ_* represents the time interval in which versioning is not switched. δ_* resembles better time barrier of Figure 5.

Algorithm 3: Better Time Barrier Design

```
1  $T \leftarrow$ new transaction arrived for execution;
2  $S_j \leftarrow$ set of other transactions arrived before $T$;
3  $V_{cur} \leftarrow$ versioning method for currently running transactions;
4  $V_{new} \leftarrow$ new versioning method computed for the transaction $T$;
5  if ($V_{cur} \neq V_{new}$) then
6      $V_{cur} \leftarrow V_{new}$;
7  Execute $T$ using $V_{cur}$;
8  if ($T$ conflicts with $T_j \in S_j$) then
9      Abort $T$;
10 else if ($T$ conflicts with $T' \notin S_j$) then
11     Handle conflict between $T$ and $T'$ using the contention management
        technique;
```

5.3. Better Switching Mechanism Design

Switching between *Eager* and *Lazy* versioning should ideally be done with no overhead. However, there might be a possibility of continuous switching between the versioning methods for every new transaction. This may result a significant amount of overhead in total execution time of the transactions. Thus, the idea is to minimize the total number of switching between the versioning methods without affecting the total execution time of the transactions. The better switching mechanism avoids the possible continuous switching between the versioning methods for every new transaction, thus helps to reduce the overhead. The versioning method is switched from one to another only if the switching condition is satisfied continuously for a specified number of times (which we call a switching interval threshold SW_INT). Let the current versioning method $V_{cur} = Eager$. Suppose at time t, ADAPTIVE decides to switch to $V_{new} = Lazy$. With better switching mechanism, ADAPTIVE does not switch to $V_{new} = Lazy$ at t but waits until a switching interval threshold SW_INT. We define SW_INT as the number of transactions that execute using the current versioning method V_{cur} before switching to the new versioning method V_{new} after t. When

$SW\_INT = 0$, ADAPTIVE does not wait for switching the versioning method. We use $SW\_INT > 0$ in the *better switching mechanism* design. Let λ be the time interval during which SW_INT number of transactions finish execution using the current versioning method $V_{cur} = Eager$. For every next (new or restarted) transaction arrived during the interval λ, if $V_{new} = Lazy$ satisfies (i.e., $V_{new} \neq V_{cur}$), then the versioning method switches to $V_{new} = Lazy$ at time $t + \lambda$. Otherwise, versioning method remains to V_{cur}. We determine the switching interval threshold SW_INT by using an empirical method, i.e., varying the value of SW_INT from 2 up to 10 and picking the one with the best performance. Note that the time interval λ denotes the time elapsed until the consecutive SW_INT number of transactions satisfy for the switching of the versioning method. So, the value of λ may not be necessarily the same for all switching events. Figure 6 illustrates the design of the better switching mechanism. The pseudocode is given in Algorithm 4.

6. Experimental Evaluation

We now evaluate the performance of ADAPTIVE using 5 micro-benchmarks and 8 complex benchmarks from STAMP and STAMPEDE benchmark suites. The evaluation is performed in an STM implementation using TinySTM [26,27] modified appropriately to incorporate ADAPTIVE. For persistent TM, the tests were executed on an Intel Core i7-7700K 4.20 GHz, 64-bit Haswell processor with 4 cores, each with 2 hyper threads. Each core has private L1 and L2 caches, whose sizes are 256 KB and 1 MB, respectively. There is also an 8 MB L3 cache shared by all 4 cores and 32 GB main memory. The results reported are the average of 10 runs varying the number of threads from 1 to 16. For non-persistent TM, 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 reported are the average of 10 experimental runs. The results are for varying number of threads from 1 to 32. First, we present the experimental results for basic ADAPTIVE in persistent memories. We also provide the execution time overhead in the basic ADAPTIVE. Later, we provide the experimental results for optimized ADAPTIVE with better time barrier using *suicide* contention management technique in non-persistent memories. And finally, we extend the results in non-persistent memories using both better time barrier and switching mechanism. We also compare the performance of optimized ADAPTIVE against four different contention management techniques.

6.1. Experimental Setup

We developed an STM-based implementation using TinySTM [26,27]. 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 [26,27]. 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*.

Table 3. Summary of five micro-benchmarks and eight complex benchmarks in STAMP (and STAMPEDE) benchmark suite, including some of their properties.

Benchmark	Application	Description	RW Set	Contention
bank	banking	implements banking transactions	small	low
red black tree	BST	balances the nodes of binary tree	small	low
hash set	data structure	stores information using hashing	medium	low
linked list	data structure	maintains linear collection of data	medium	medium
skip list	data structure	maintains linked hierarchy of subsequences	medium	low
bayes	machine learning	learns a structure of a Bayesian network	large	high
genome	bioinformatics	performs gene sequencing	medium	low
intruder	security	detects network intrusions	medium	high
kmeans	data mining	implements *k*-means clustering	small	low
labyrinth	engineering	routes paths in maze	large	high
ssca2	scientific	creates efficient graph representation	small	low
vacation	OLTP	emulates travel reservation system	medium	low/medium
yada	scientific	refines a Delaunay mesh	large	medium

Algorithm 4: Better Switching Mechanism Design

```
1 $T \leftarrow$ new transaction arrived for execution;
2 $V_{cur} \leftarrow$ versioning method for currently running transactions ;
3 $V_{new} \leftarrow$ new versioning method computed for the transaction $T$;
4 $t \leftarrow$ the timestamp at which the versioning satisfies to switch from $V_{cur}$ to $V_{new}$;
5 $SW\_INT \leftarrow$ switching interval threshold;
6 $\lambda \leftarrow$ time taken by the $SW\_INT$ transactions arrived on or after timestamp $t$ to
   finish their executions;
7 if ($V_{cur} \neq V_{new}$ for time $t + \lambda$) then
8     $V_{cur} \leftarrow V_{new}$;   // switch the versioning method to $V_{new}$ at timestamp
       $t + \lambda$
9 Execute $T$ using $V_{cur}$;
```

6.2. Persistent Memory Emulation

We emulate persistent memory using DRAM in our experiments following Avni et al. [43]. We separate 500 MB region of DRAM for the persistent memory emulation. All the original data are kept in this region. Moreover, we use this region for keeping the persistent undo log when a transaction runs using eager versioning and to persist the redo log when transaction runs using lazy versioning. To emulate the power failure and crash in persistent memory, we leave the power on and wipe out all the volatile log records so that the rollback (in case of abort in eager versioning) and update (in case of commit but not yet written to memory in lazy versioning) operations will be handled by those persistent log records. We use spin loop for this purpose that runs for around 200 ms (which is sufficient to load and store the log records).

6.3. Benchmarks

We use both micro and complex benchmarks from the TM literature. A summary of some prominent properties of these benchmarks such as targeted applications, short description of the applications, the size of the read write (RW) set, and the amount of contention is given in Table 3.

Micro-Benchmarks: We use five micro-benchmarks, namely *bank, red black tree, hash set, linked list*, and *skip list* used in several previous studies, e.g., [21,22,26–28]. These benchmarks simulate the basic concurrent execution scenario for transactions with (relatively) small read/write sets.
STAMP: STAMP is a well-known and widely-used benchmark. It consists of eight applications: *bayes, genome, intruder, kmeans, labyrinth, ssca2, vacation*, and *yada* of varying complexity. These applications span a variety of computing domains as well as runtime transactional characteristics such as varying transaction lengths, read and write set sizes, and amounts of contention [23].
STAMPEDE: Recently, Nguyen et al. [29] argued that the programming model and data structures used in STAMP benchmarks suffer from performance bottlenecks. They then modified the programming structure of these benchmarks in a way the bottlenecks can be removed. They finally provided a set of rewritten STAMP benchmarks calling them STAMPEDE benchmarks.

6.4. Evaluation of ADAPTIVE *in Persistent Memories*

For persistent memories, we ran the experiments using up to 16 threads and report the results accordingly with an average of 10 runs for each thread. We present 4 different types of results for each benchmark suite: (i) total data movements (including both loads from and stores to the persistent memory), (ii) total number of stores to the persistent memory, (iii) total number of aborts, and (iv) total execution time. We first discuss results for metrics (i)–(iii) separately for each benchmark suite and then we discuss results for metric (iv) together for all the benchmark suites.
Results on Micro-benchmarks. All the transactions in these benchmarks were run with *update rate* of 20%. When transactions were executed with less number of threads, we found that the transaction commit rate is higher than the transaction abort rate and the cost in lazy versioning is higher than the cost in eager versioning. With the increase in number of threads, the abort rate is also increased. Figure 7–9 provide the experimental results on all five micro-benchmarks for total data movements, total number of stores to PM and total number of aborts, respectively. We noticed that *Lazy_CTL* has consistently better performance than *Lazy_ETL* on all the five micro-benchmarks. This is because the early detection of conflict and locking the memory addresses has increased abort rate than detecting the conflicts and locking the memory addresses at the commit time. We observed that the total number of aborts in ADAPTIVE versioning has been decreased compared to that in both eager and lazy versioning. To be specific, *Adaptive_ETL* has up to 1.5× less number of aborts than *Lazy_ETL* and up to 1.7× less number of aborts than *Eager*. Similarly, *Adaptive_CTL* has upto 1.3× less number of aborts than *Lazy_CTL* and upto 3.8× less number of aborts than *Eager*. Figure 7 shows that the total data movements to and from the persistent memory (i.e., loads and stores to the PM) has been decreased in ADAPTIVE versioning. *Adaptive_ETL* has up to 3.4× less data movements than *Lazy_ETL* and up to 1.1× less data movements than *Eager*. *Adaptive_CTL* has up to 3× and 1.3× less number of data movements compared to that in *Lazy_CTL* and *Eager*, respectively. Figure 8 shows the total number of stores to the PM. We can see that lazy versioning has less number of stores to the persistent memory. This is because the aborts in lazy versioning do not participate in stores to the persistent memory. On the other hand, in eager versioning, an abort requires the memory addresses to be rolled back to the previous consistent states, thus increasing the total number of stores to the PM. We observed that the total number of stores in ADAPTIVE is always less than the *Eager* and is greater than *Lazy* in most of the cases. Compared to *Eager*, *Adaptive_ETL* has up to 1.4× less number of PM stores and *Adaptive_CTL* has up to 2.1× less number of PM stores. Compared to *Lazy*, *Adaptive_ETL* has up to 1.8× more number of PM stores and *Adaptive_CTL* has up to 1.6× more number of PM stores. We also noticed that *Adaptive_CTL* performs better than *Adaptive_ETL* in each micro-benchmark. This is because *Lazy_CTL* performs better

than *Lazy_ETL* and *Adaptive_ETL* was designed using *Eager* and *Lazy_ETL* whereas *Adaptive_CTL* was designed using *Eager* and *Lazy_CTL*, respectively.

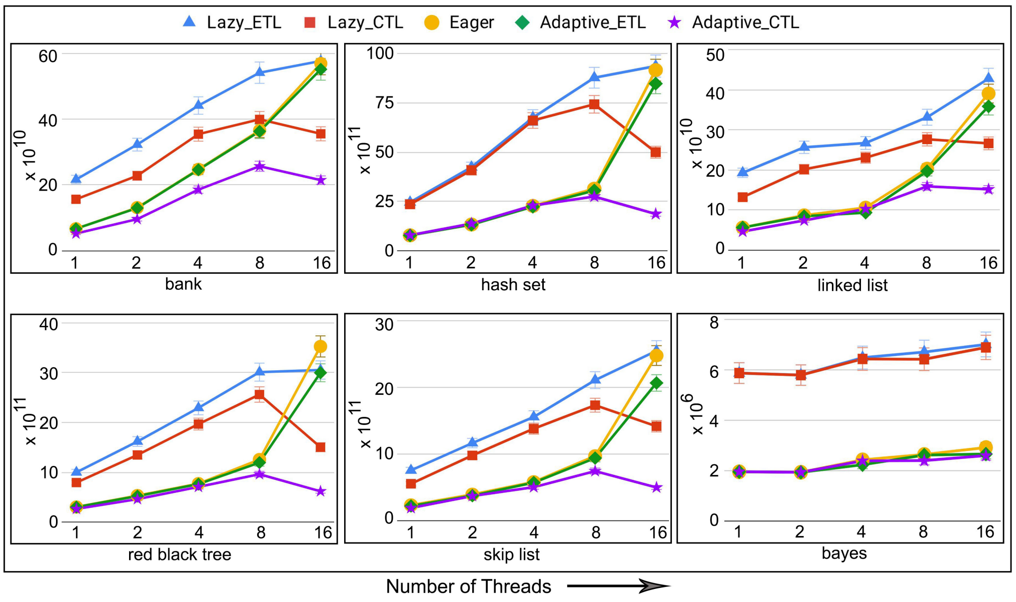

Figure 7. Data movements in micro-benchmarks and *bayes* from STAMP executing in persistent TM.

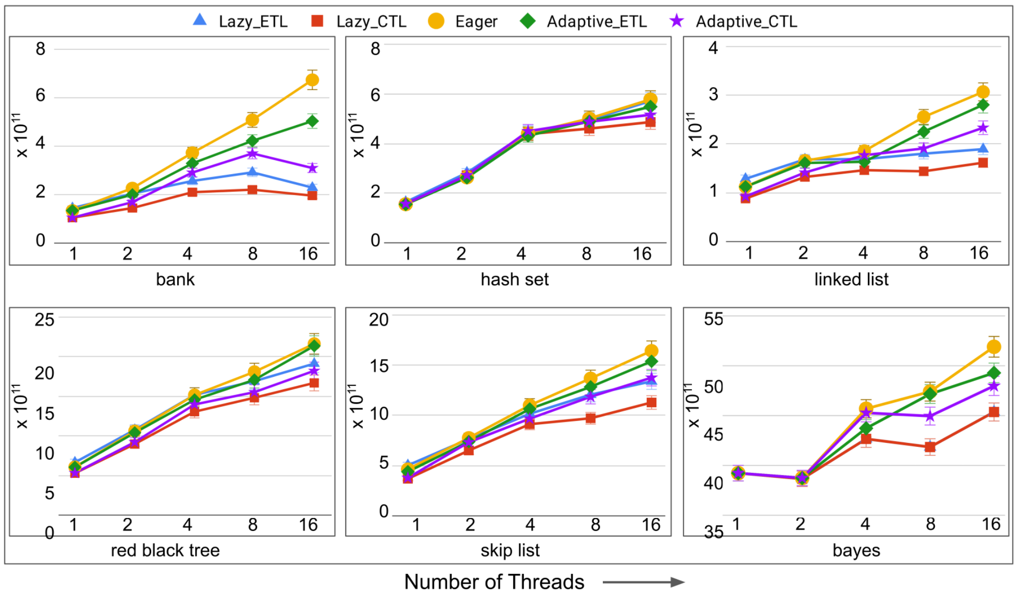

Figure 8. Total number of stores to the PM in micro-benchmarks and *bayes* from STAMP.

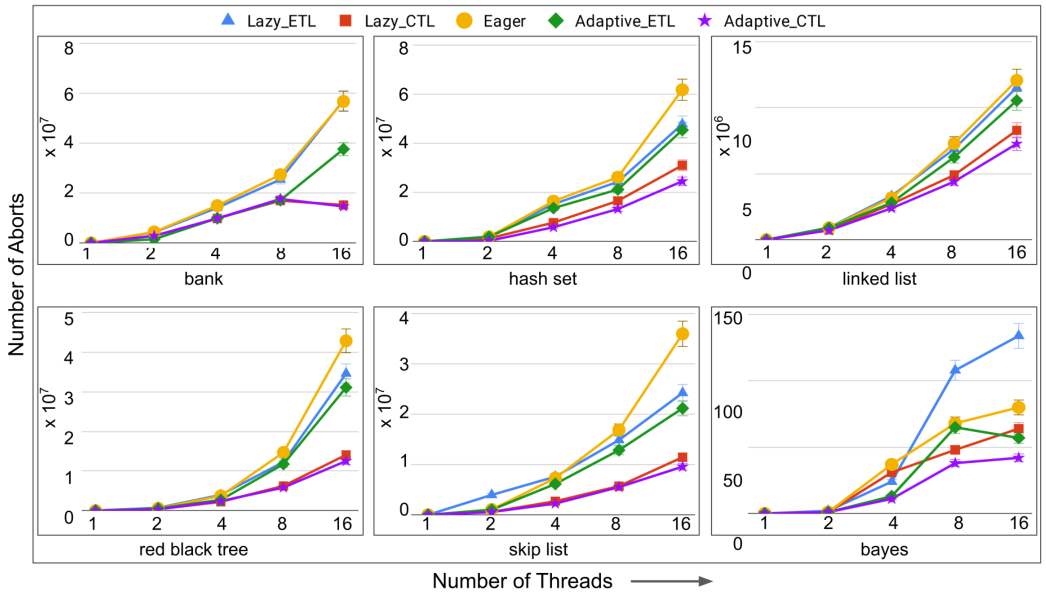

Figure 9. Number of aborts in micro-benchmarks and *bayes* from STAMP in persistent TM.

Results on STAMP Benchmarks. Figure 10 provides total data movement results. It is obvious that when transactions are executed with less number of threads, transaction commit rate is higher and there is less number of total data movements *Eager* than *Lazy*. With the increase in number of threads, transaction abort rate also increases and total number of data movements in *Eager* also starts to increase due to the frequent requirement of rollbacks. The results obtained for *genome* and *kmeans-low* show that *Eager* starts to encounter more data movements than *Lazy* beyond 8 threads. The same scenario starts beyond 4 threads in *Intruder* and *yada*. Irrespective of the abort rate change, ADAPTIVE always has less number of total data movements compared to the respective *eager* and *lazy* versioning. Specifically, *Adaptive_ETL* has up to 6× less data movements than *Lazy_ETL* and up to 2× less data movements than *Eager*. *Adaptive_CTL* achieved up to 3× less data movements compared to *Lazy_CTL* and up to 35× less data movements (in *yada*) compared to *Eager*.

Figure 11 shows the results for total number of stores to the persistent memory. Similar to micro-benchmarks, *Lazy* versioning has less number of PM stores than the *Eager* versioning in STAMP benchmarks as well. In ADAPTIVE versioning, total number of PM stores decreases compared to *Eager* (up to 28×) and increases compared to *Lazy* (up to 2×).

Figure 12 shows the results for total number of aborts. Similar to micro-benchmarks, the total number of aborts in STAMP benchmarks also decreases when executing the transactions using ADAPTIVE versioning. This is because the ADAPTIVE versioning always tries to minimize the total data movements by adapting a suitable versioning method between *Eager* and *Lazy*. In *Adaptive_ETL*, the total number of aborts are up to 3× and 17× less than *Eager* and *Lazy_ETL*, respectively. In *Adaptive_CTL*, the total number of aborts are up to 240× and 2.8× less than *Eager* and *Lazy_CTL*, respectively.

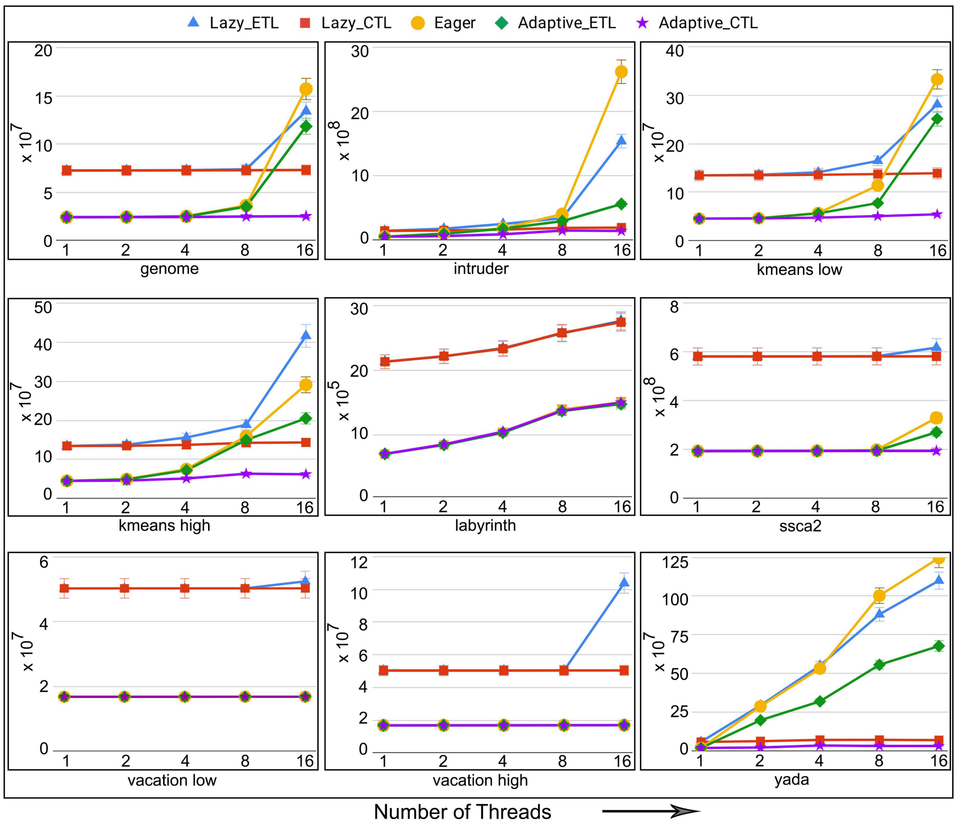

Figure 10. Data movements in STAMP benchmarks executing in persistent TM.

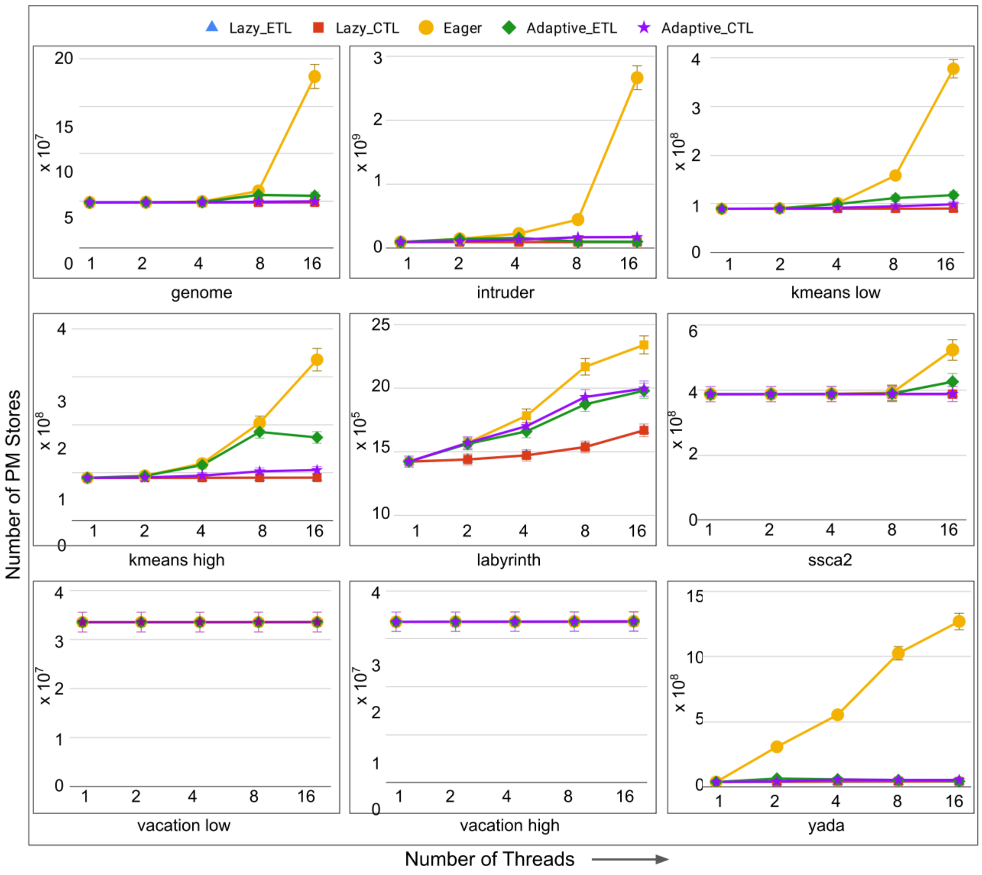

Figure 11. Total number of stores to the PM in STAMP benchmarks.

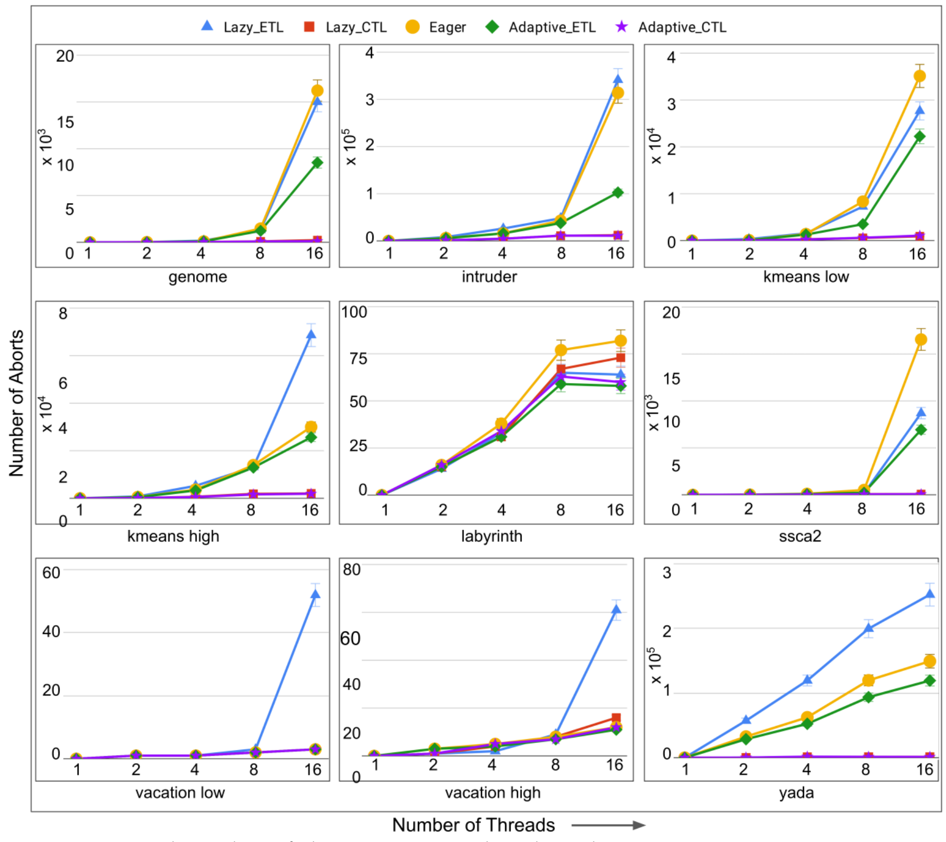

Figure 12. Total number of aborts in STAMP benchmarks in persistent TM.

Results on STAMPEDE Benchmarks. Figure 13 provides the experimental results for total data movements. Similar to micro- and STAMP benchmarks, ADAPTIVE has less number of total data movements compared to *Eager* and *Lazy* in STAMPEDE benchmarks as well. We observed that *Adaptive_ETL* has up to 3.6× less data movements than *Lazy_ETL* and *Adaptive_CTL* phas up to 6× less data movements than *Lazy_CTL*. Compared to *Eager*, *Adaptive_ETL* achieved up to 4.6× less data movements and *Adaptive_CTL* achieved up to 3.1× less data movements.

Figure 14 shows the experimental results for total number of PM stores in STAMPEDE benchmarks. It also follows the results obtained for micro- and STAMP benchmarks where *Lazy* versioning has less number of PM stores than the *Eager* versioning and ADAPTIVE versioning lies between the two values. To be precise, *Adaptive_ETL* has up to 64× less number of PM stores than *Eager* and up to 2.7× more number of PM stores than *Lazy_ETL* whereas *Adaptive_CTL* has up to 9× less number of PM stores than *Eager* and up to 18× more number of PM stores than *Lazy_CTL*

The experimental results for the total number of aborts are shown in Figure 15. Again, similar to micro- and STAMP benchmarks, the total number of aborts in ADAPTIVE versioning has been decreased compared to that in *Eager* and *Lazy* versioning. Precisely, *Adaptive_ETL* has up to 14.3× and 14.7× less number of aborts compared to *Eager* and *Lazy_ETL*, respectively. Similarly, *Adaptive_CTL* has up to 2.7× and 9.2× less number of aborts compared to *Eager* and *Lazy_CTL*, respectively.

To summarize the above results, in all three benchmark suites, we observed that the total movements of data (i.e., loads and stores) and the total number of aborts in ADAPTIVE versioning decrease compared to that in *Eager* and *Lazy* versioning. This helps us to achieve better execution time in ADAPTIVE. Particularly, as the number of aborts decrease in ADAPTIVE design compared to the non-adaptive baselines, there will be less number of transaction restarts as well as less number of data movements which ultimately reduces the total execution time of the transactions. We present the execution time results for all three benchmark suites in the next sub-section. We also observed that the total number of stores to persistent memories in ADAPTIVE versioning are decreased compared to *Eager*. However, compared to *Lazy*, they are increased in ADAPTIVE versioning.

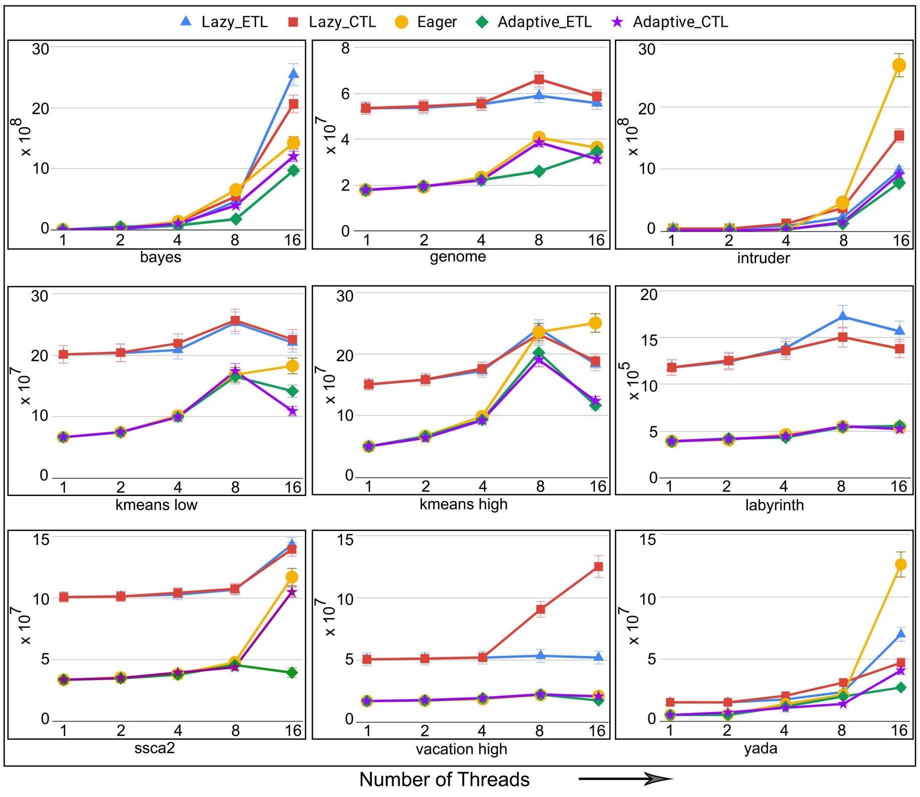

Figure 13. Data movements in STAMPEDE benchmarks executing in persistent TM.

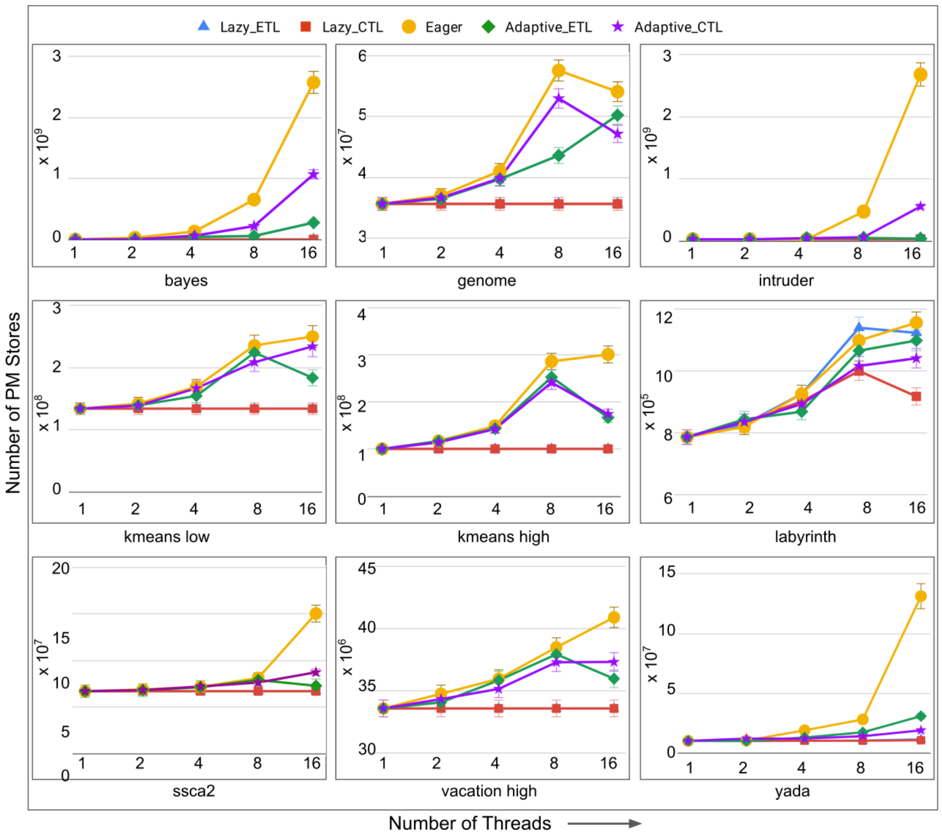

Figure 14. Total number of stores to the PM in STAMPEDE benchmarks.

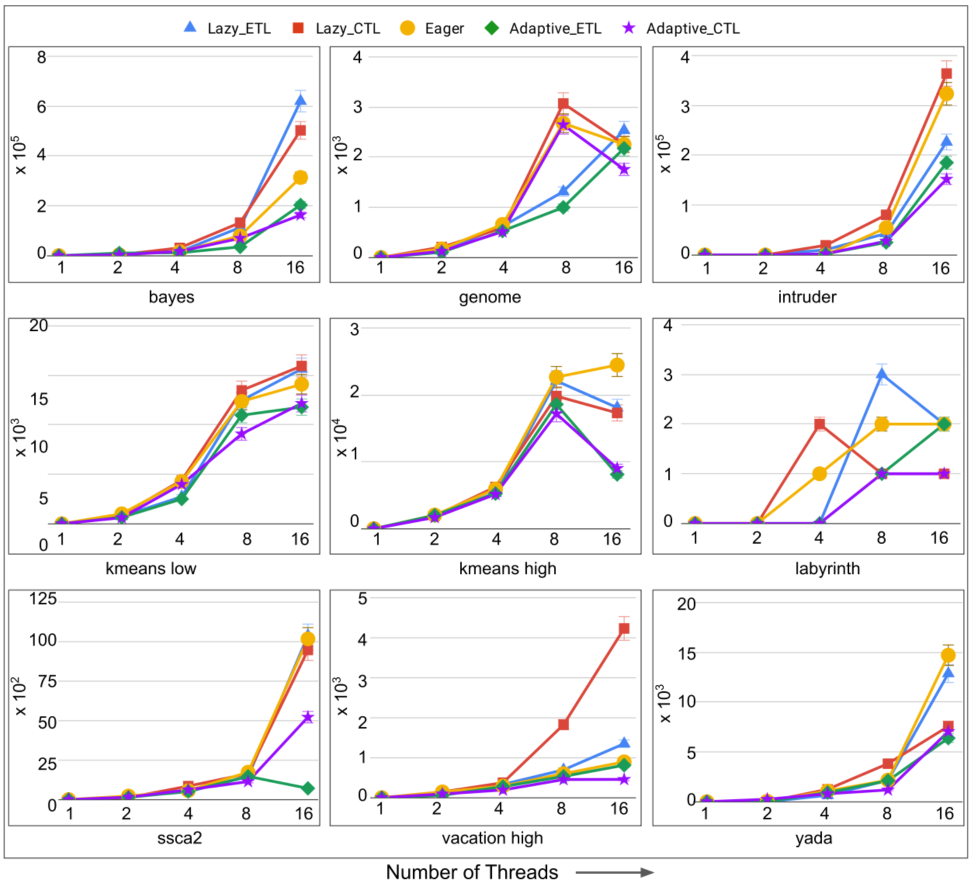

Figure 15. Total number of aborts in STAMPEDE benchmarks in persistent TM.

Execution Time Results for Persistent Memories. Execution time is impacted in ADAPTIVE due to the switching between eager and lazy versioning at runtime. Additionally, the design of time barrier also introduces time delay in some benchmarks. In most of the benchmarks, the delay due to time barrier is compensated as ADAPTIVE lowers the data movements and the number of aborts. We were interested to see the maximum increase on time in any benchmark that we used in our experimentation.

The results obtained for micro-benchmarks are shown in Figure 16. Recall that, for micro-benchmarks, we measured the execution time for 10,000 transactions, each executed with an update rate of 20%. All the 5 micro-benchmarks were executed with the five different versioning designs and the total number of transactions for each design were counted.

We noticed that, in most of the applications, execution time in ADAPTIVE decreases compared to that in both *Eager* and *Lazy* versioning. This is due to the decrease in total number of data movements and total number of aborts in ADAPTIVE versioning. In *Adaptive_ETL*, execution time decreases by up to 21% compared to *Lazy_ETL* and up to 17% compared to *Eager*. In *Adaptive_CTL*, execution time decreases by up to 28% compared to *Lazy_CTL* and up to 33% compared to *Eager*. However, in some applications (for example see the results for *bank* and *linked list*), the execution time in ADAPTIVE increases compared to that in *Eager* or *Lazy* versioning. This is because the decrease in number of aborts is significantly less and is insufficient to compensate the overhead due to barrier and switching between the versioning methods. In *bank* micro-benchmark, we noticed that the execution time in *Adaptive_CTL* increases by up to 9% compared to *Lazy_CTL*. In *linked list*, the execution time in *Adaptive_CTL* increases by up to 12% compared to *Lazy_CTL* and the execution time in *Adaptive_ETL* increases by up to 10% compared to *Lazy_ETL*.

For the STAMP and STAMPEDE benchmarks, we measured the execution time for each of the applications. Figures 17 and 18 illustrate the execution time results for the STAMP and STAMPEDE benchmarks, respectively. As ADAPTIVE lowers the data movements and the number of aborts, most of the applications (e.g., bayes, kmeans high, labyrinth, ssca2 and vacation high in Figures 17 and 18) have decreased execution time in ADAPTIVE than in *Eager* or *Lazy* designs. However, in some applications (e.g., genome, intruder and yada), we noticed that the execution time in ADAPTIVE increases by at most 16% more compared to the execution time of *Eager* or *Lazy*.

We observed that the experimental results presented above for the execution time in persistent TM barely scale in throughput beyond 8 threads. This occurred due to the experimental environment used for conducting the tests. For persistent TM, the tests were executed on an Intel Corei7-7700K 4.20 GHz, 64-bit Haswell processor with 4 cores, each with 2 hyper threads. That means, it has 4 physical cores and 8 logical cores. Now, up to 8 threads, each process core handles an individual thread. But when executing with 16 threads, threads spend more time waiting for their turn to be handled, thus increasing execution time and decreasing throughput.

To summarize, in all of the benchmark suites, ADAPTIVE performs better for total number of data movements and total number of aborts compared to individual *Eager* and *Lazy* designs. Also, in most of the applications in each benchmark suite, the total execution time in ADAPTIVE decreases compared to that in *Eager* and *Lazy*. However, in some cases, the decrease in data movements and the number of aborts is insufficient to lower the overhead due to barrier and switching between the versioning methods and that increases the execution time in ADAPTIVE by at most 16% compared to that of using *Eager* and *Lazy* designs.

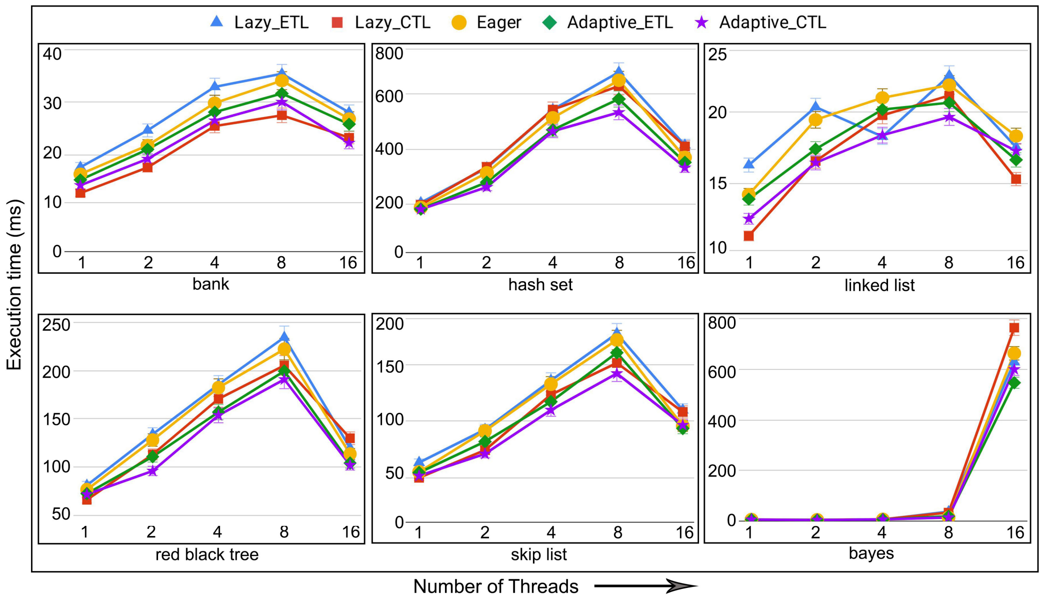

Figure 16. Execution time in micro-benchmarks and *bayes* from STAMP executing in persistent TM.

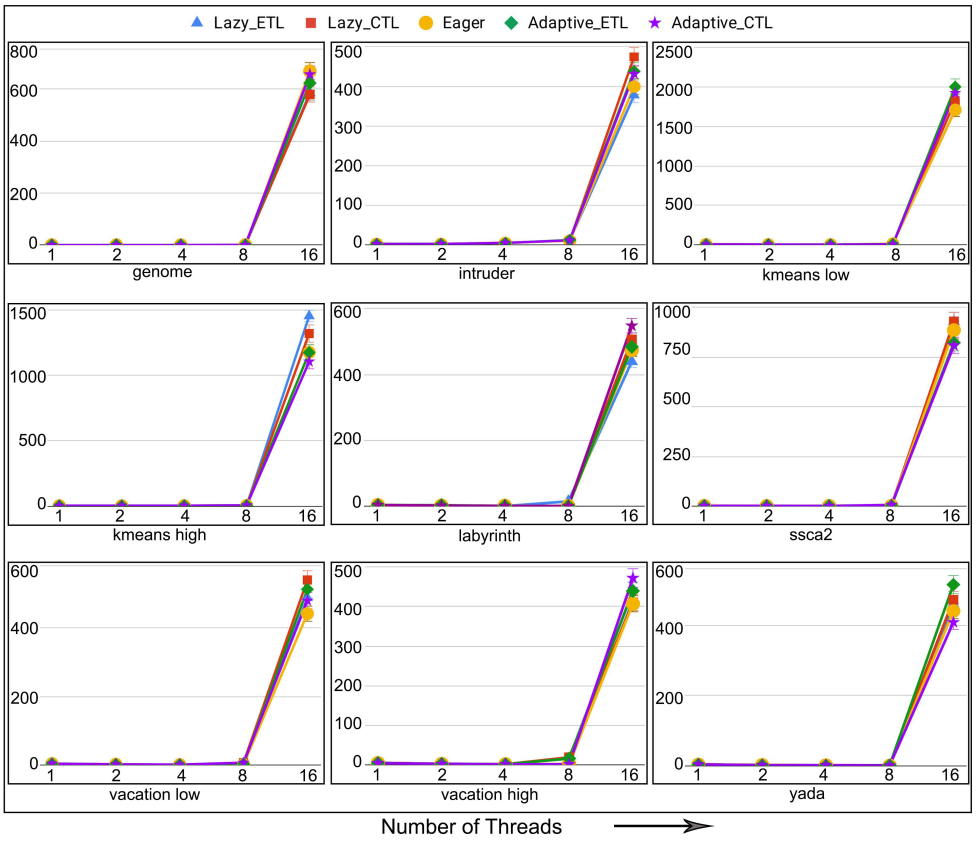

Figure 17. Execution time in STAMP benchmarks executing in persistent TM.

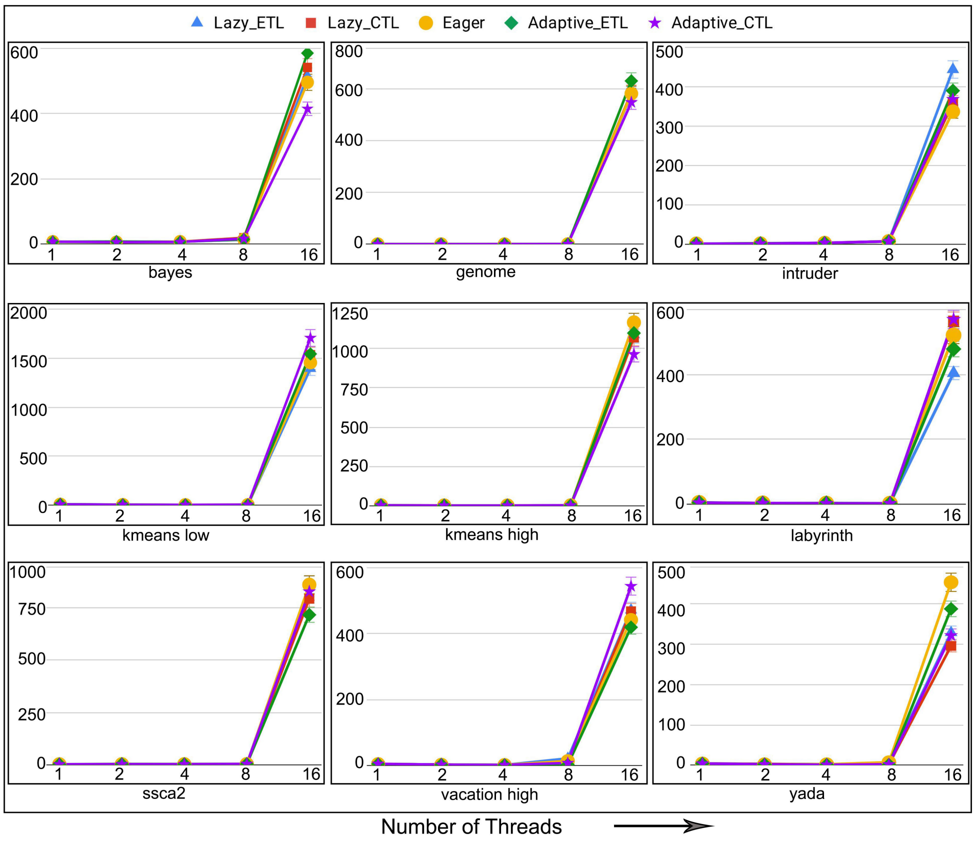

Figure 18. Execution time in STAMPEDE benchmarks executing in persistent TM.

6.5. Evaluation of Adaptive in Non-Persistent Memories

We report the results from experiments performed using up to 32 threads. The evaluation is on performance metrics metrics execution time and number of aborts.

Results on Micro-benchmarks. The execution time results in five different micro-benchmarks are provided in Figure 19. Figure 20 provides the result for the number of aborts. The results are for 10,000 transactions, each executed with *update rate* of 20%. Figure 19 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 20 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*.

Results on STAMP Benchmarks. Figures 21 and 22, respectively, provide the execution time and number of aborts results. 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* 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 Figure 22, 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*.

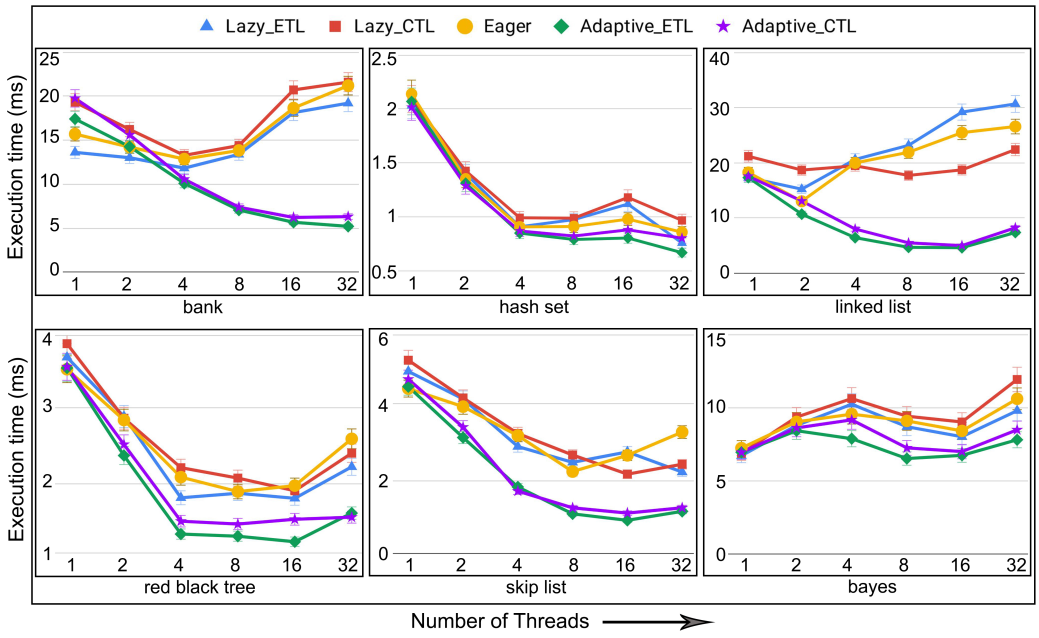

Figure 19. Execution time in micro-benchmarks and bayes from STAMP using better time barrier in non-persistent TM.

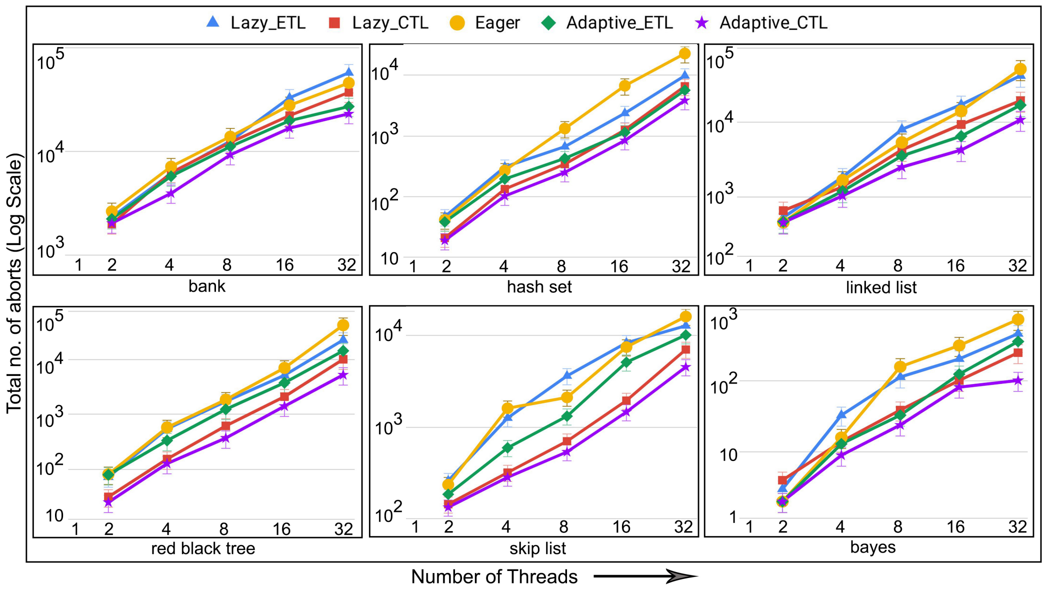

Figure 20. Number of aborts in micro-benchmarks and bayes from STAMP using better time barrier in non-persistent TM.

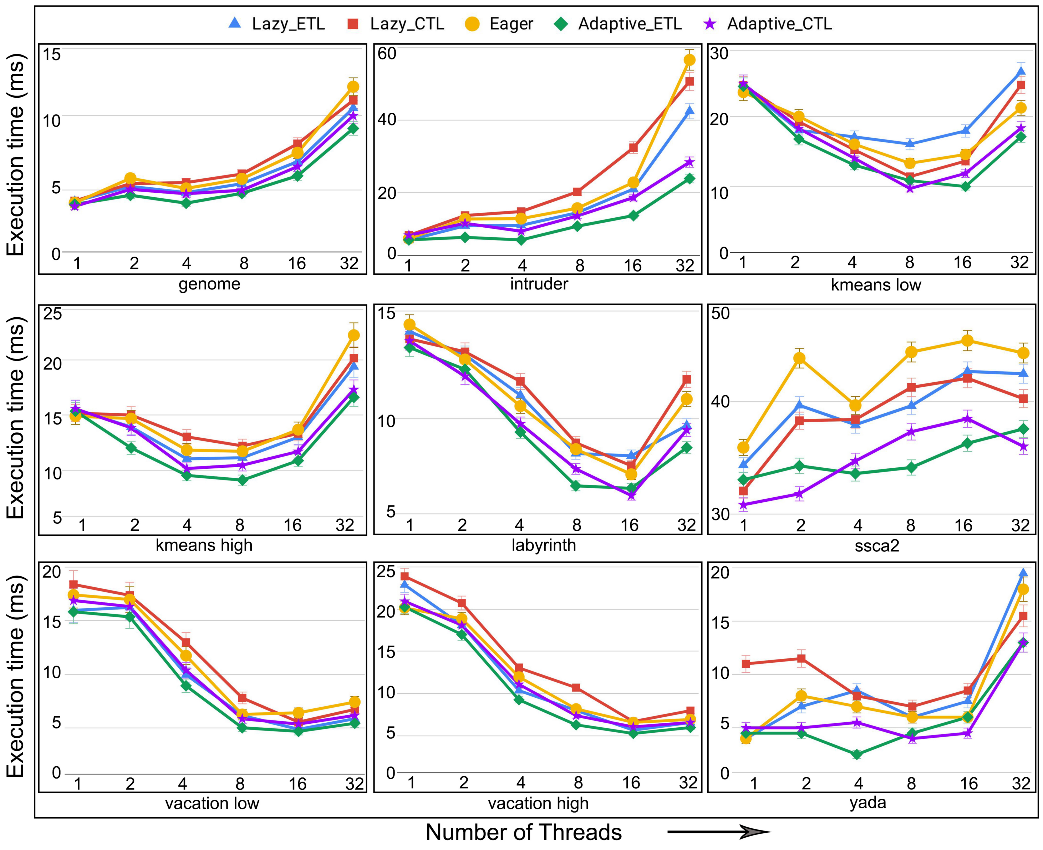

Figure 21. Execution time in STAMP benchmarks using better time barrier in non-persistent TM.

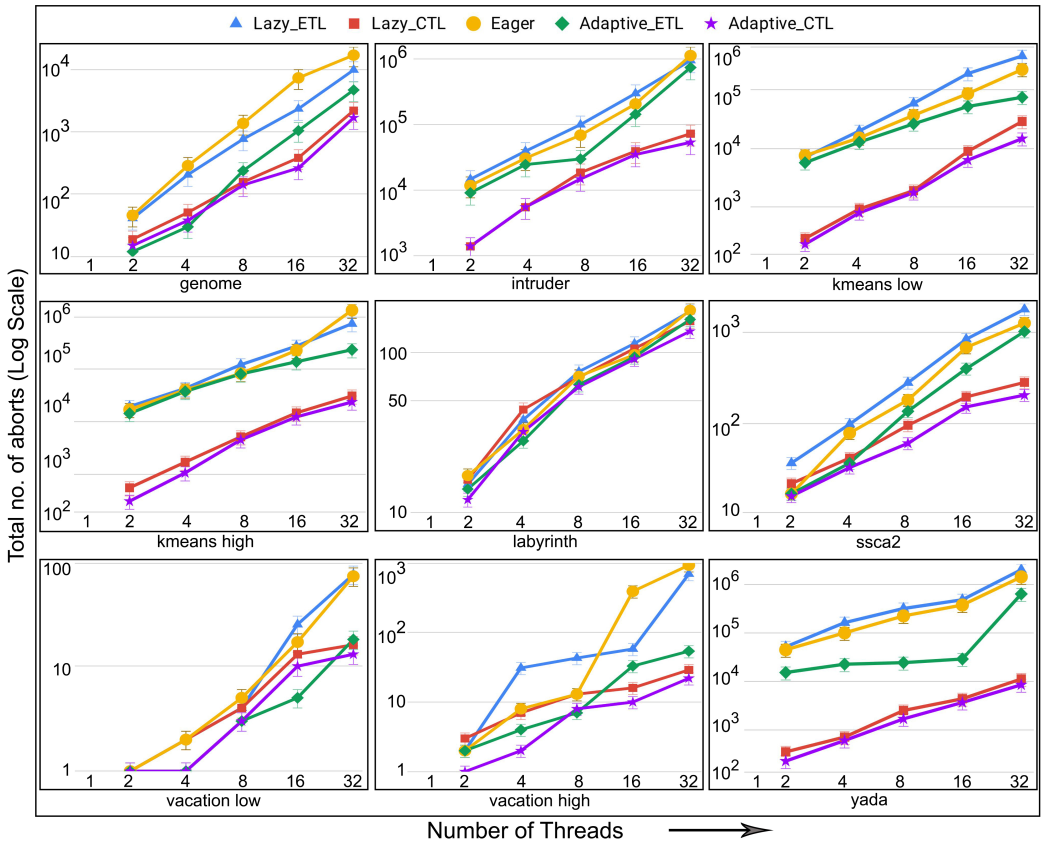

Figure 22. Number of aborts in STAMP benchmarks using better time barrier in non-persistent TM.

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 (Figures 23 and 24). 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 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 time gain for ADAPTIVE ranges between 1 and 1.16 when running with threads up to 4 numbers. It is interesting to mention here that the ADAPTIVE versioning technique outperforms both *eager* and *lazy* versioning for most of the applications in all the benchmark suites. This is mainly due to the decrease in number of aborts and the better time barrier design where non-conflicting transactions can execute and commit in parallel. Furthermore, the delay due to writing data in persistent log is not the concern in non-persistent TMs which also helps in reducing the total execution time.

Further Results. The results in Figures 19–24 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 $SW\_INT = 10$. Figures 25 and 26 show the execution time and total number of aborts, respectively 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 the improvement on execution time. Figure 27 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.

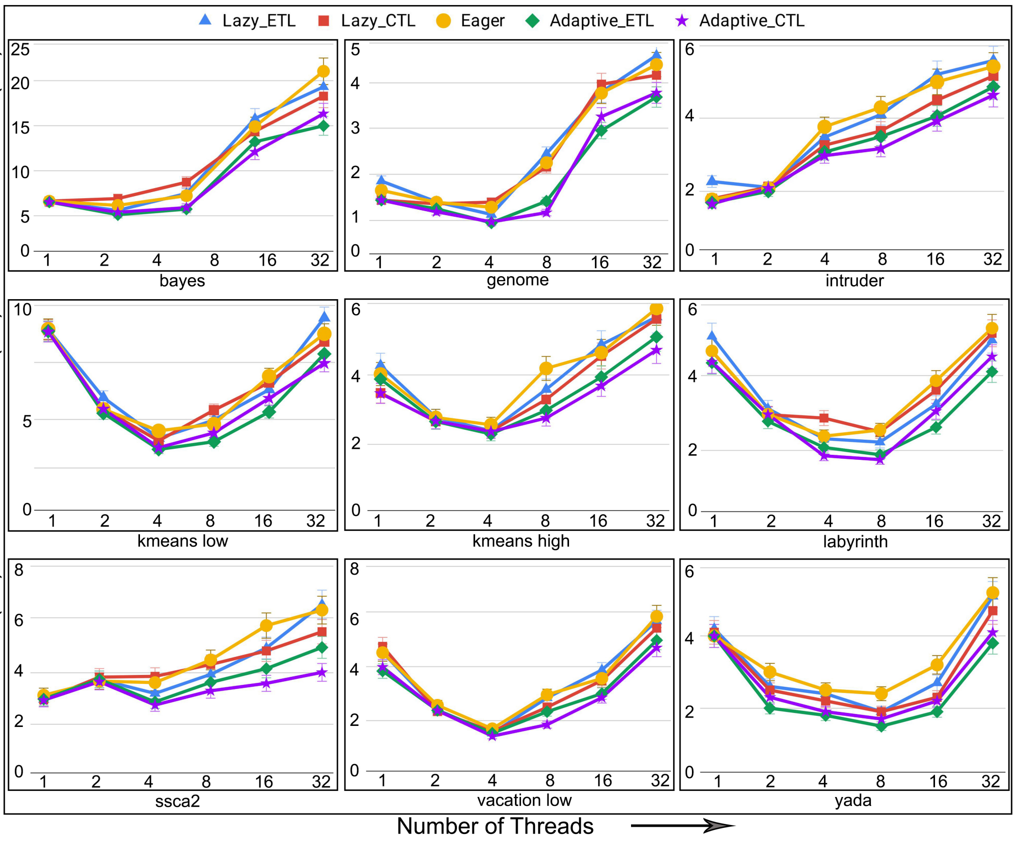

Figure 23. Execution time in STAMPEDE benchmarks using better time barrier in non-persistent TM.

The experiments so far use $Threshold_{Eager} = \frac{1}{2}$ and $Threshold_{Lazy} = 2$ as computed in Section 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 Section 4 are appropriate.

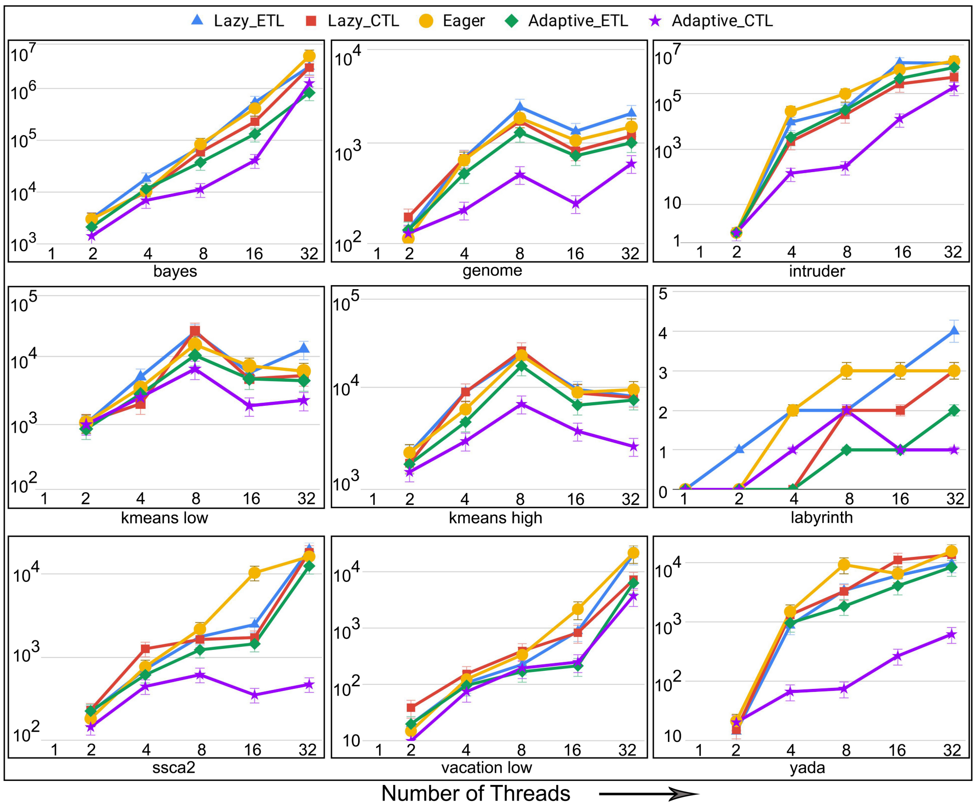

Figure 24. Number of aborts in STAMPEDE benchmarks using better time barrier in non-persistent TM.

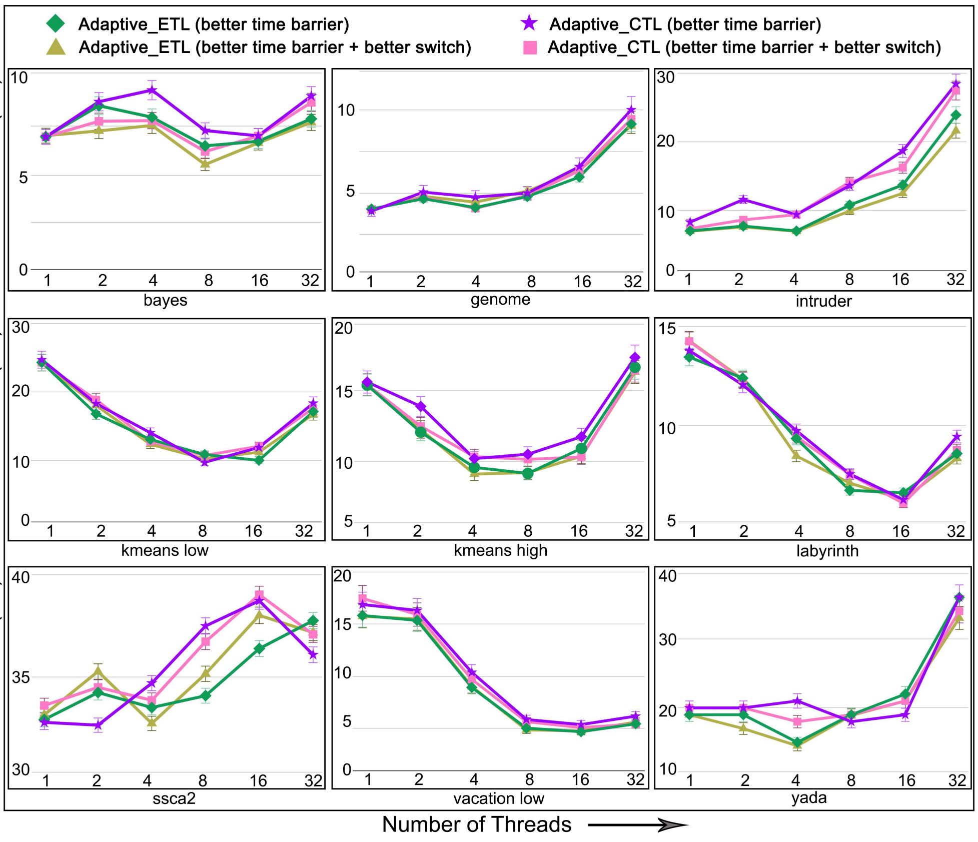

Figure 25. Execution time in STAMP benchmarks using better barrier and better switch in non-persistent TM.

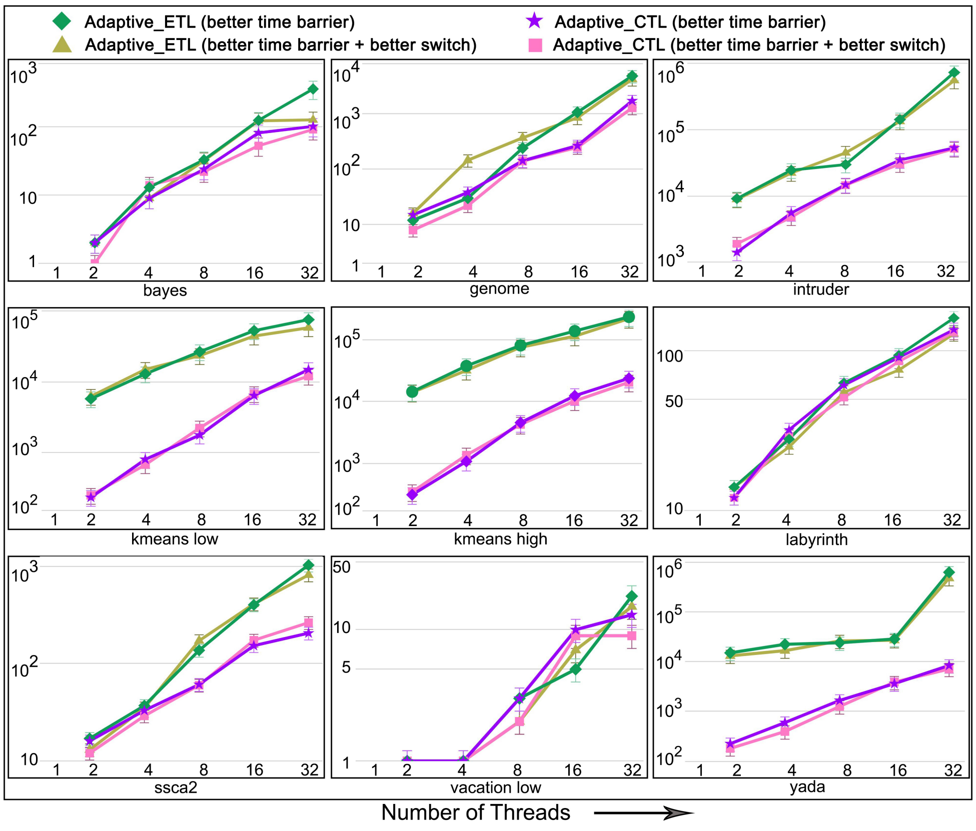

Figure 26. Aborts in STAMP benchmarks using better barrier and better switch in non-persistent TM.

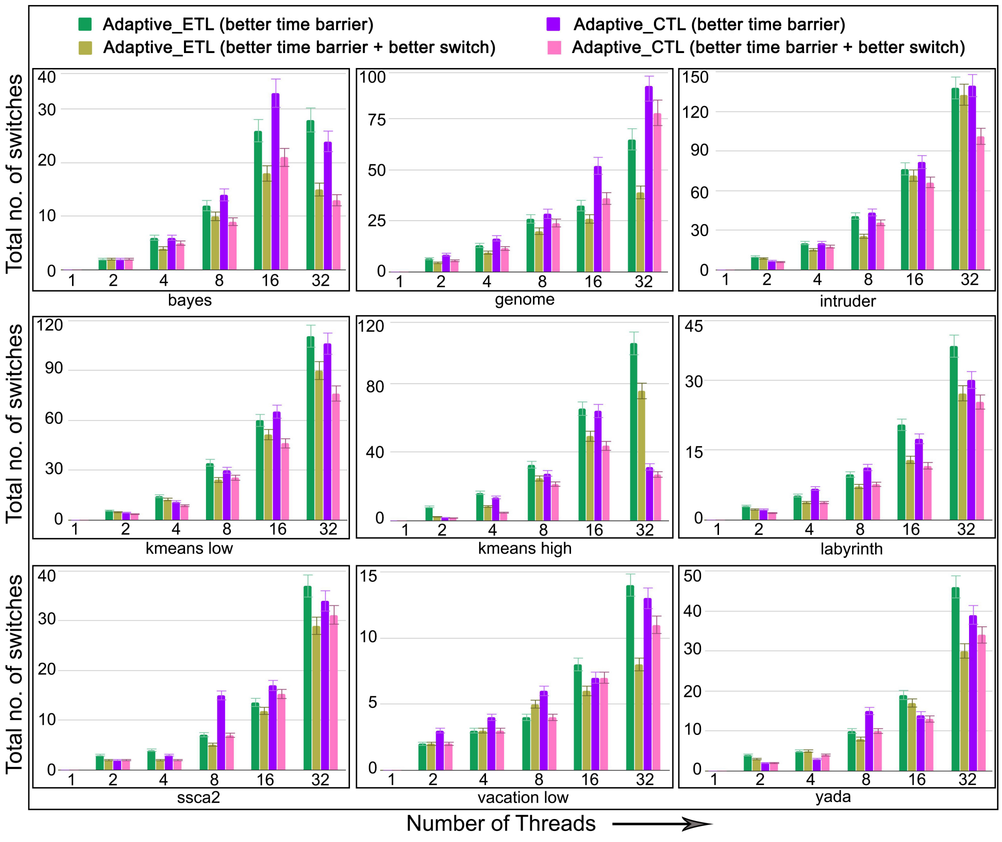

Figure 27. Decrease in total number of switches between versioning methods using better switch mechanism in non-persistent TM.

The results reported in Figures 19–27 use *suicide* as a contention management technique. We were interested to see whether other strategies perform better than *suicide*. Therefore, we performed experiments using 4 different contention management techniques *suicide, delay, back-off*, and *kill* for the comparison. The execution time is shown in Figure 28 and the number of aborts is shown in Figure 29 for *Adaptive_ETL* in STAMP benchmarks. The results showed not significant change on performance in some of the benchmarks, while in the rest, the selection of contention management technique affected the performance. For example, *genome* and *intruder* performed better with *suicide* whereas, *kmeans* 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 upto 4 threads. This is mainly because transactions have almost constant abort rate and versioning method change is not necessary.

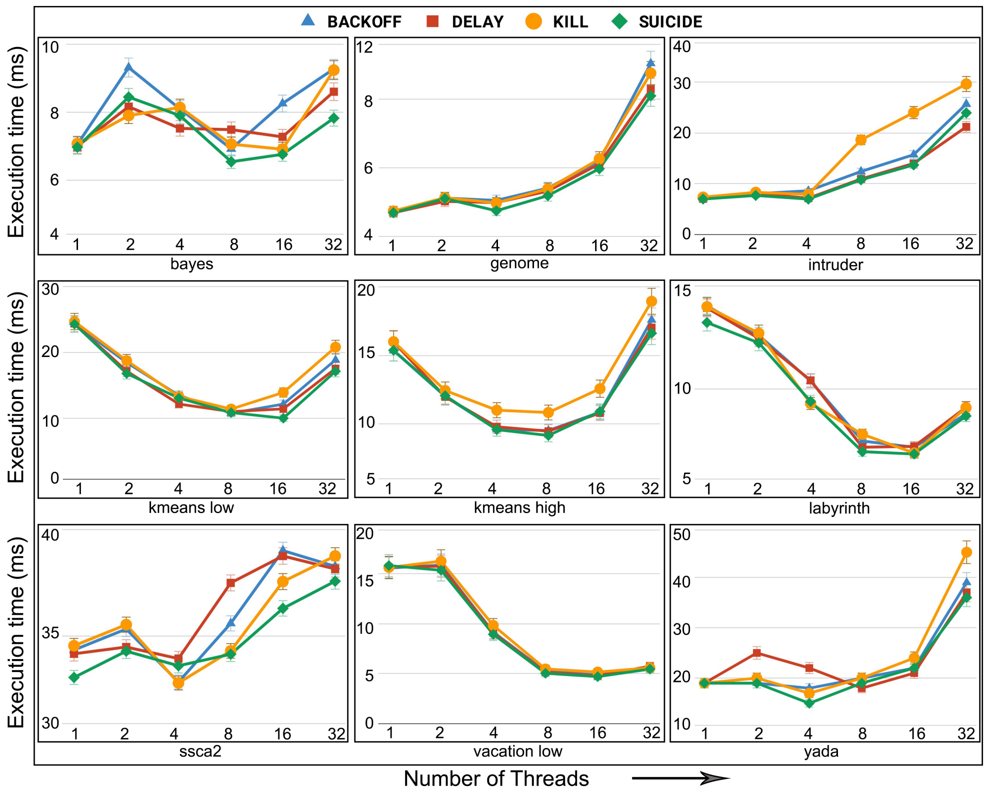

Figure 28. Execution time in STAMP benchmarks for *Adaptive_ETL* for four different contention management techniques in non-persistent TM.

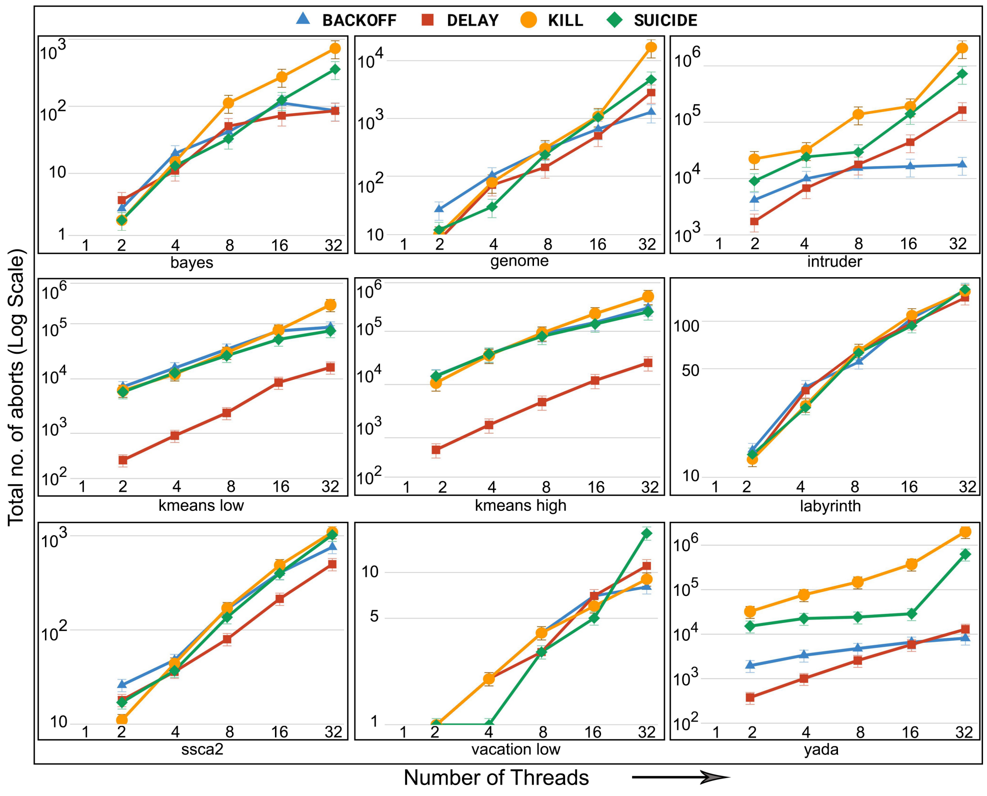

Figure 29. Aborts in STAMP benchmarks for *Adaptive_ETL* for four different contention management techniques in non-persistent TM.

7. Concluding Remarks

Transactional memory has been receiving much attention from both academia and industry. One of the most challenging issues 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 article, we have presented an adaptive framework that dynamically switches between eager and lazy versioning at runtime through appropriate transaction abort/commit data collected at runtime, obviating the need of a priori knowledge on the workload and contention scenario to pick the appropriate versioning method for better performance. Our framework is quite simple and applicable in both persistent and non-persistent TM systems. The framework achieves significantly better performance in terms of execution time and number of aborts for both persistent and non-persistent memories compared to eager and lazy versioning running individually in 5 micro-benchmarks and 8 applications from STAMP and STAMPEDE suites. In persistent TM systems, the adaptive framework achieved performance improvements as much as 1.5× for execution time and as much as 240× for number of aborts, whereas in non-persistent TM systems, it achieved performance improvements as much as 6.3× for execution time and as much as 170× for number of aborts. We believe that our results and techniques will be helpful in choosing proper versioning for TM systems.

For the future work, it will be interesting to see whether there is a better technique on making decision on when to switch between eager and lazy versioning and how to minimize the time gap of switching from one versioning method to another. It will also be interesting to run experiments on the real persistent memory such as Optane DC persistent memory [47] and provide the comparison against more state-of-the-art STM, Durable STM, and HTM implementations.

Author Contributions: Conceptualization, G.S.; methodology, P.P. and G.S.; software, P.P.; validation, G.S.; formal analysis, P.P. and G.S.; investigation, P.P. and G.S.; resources, G.S.; writing—original draft preparation, P.P.; writing—review and editing, G.S.; visualization, P.P.; supervision, G.S.; project administration, G.S.; funding acquisition, G.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Science Foundation grant number CCF-1936450.

Data Availability Statement: Not applicable.

Acknowledgments: The authors thank the anonymous reviewers for their valuable comments.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Herlihy, M.; Moss, J.E.B. Transactional Memory: Architectural Support for Lock-free Data Structures. In Proceedings of the ISCA, San Diego, CA, USA, 16–19 May 1993; pp. 289–300.
2. Shavit, N.; Touitou, D. Software Transactional Memory. *Distrib. Comput.* **1997**, *10*, 99–116. [CrossRef]
3. Intel. 2012. Available online: http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell (accessed on 4 February 2021).
4. Haring, R.; Ohmacht, M.; Fox, T.; Gschwind, M.; Satterfield, D.; Sugavanam, K.; Coteus, P.; Heidelberger, P.; Blumrich, M.; Wisniewski, R.; et al. The IBM Blue Gene/Q Compute Chip. *IEEE Micro* **2012**, *32*, 48–60. [CrossRef]
5. 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 Proceedings of the ISCA, Portland, Oregon, 13–17 June 2015; pp. 144–157. [CrossRef]
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 Proceedings of the ISCA, Tel Aviv, Israel, 23–27 June 2013; pp. 225–236. [CrossRef]
7. Bocchino, R.L.; Adve, V.S.; Chamberlain, B.L. Software transactional memory for large scale clusters. In Proceedings of the PPoPP, Salt Lake City, UT, USA, 20–23 February 2008; pp. 247–258.
8. Fung, W.W.L.; Singh, I.; Brownsword, A.; Aamodt, T.M. Hardware transactional memory for GPU architectures. In Proceedings of the MICRO, Porto Alegre, Brazil, 3–7 December 2011; pp. 296–307.
9. Manassiev, K.; Mihailescu, M.; Amza, C. Exploiting Distributed Version Concurrency in a Transactional Memory Cluster. In Proceedings of the PPoPP, Manhattan, NY, USA, 29–31 March 2006; pp. 198–208.
10. Ananian, C.S.; Asanovic, K.; Kuszmaul, B.C.; Leiserson, C.E.; Lie, S. Unbounded Transactional Memory. In Proceedings of the HPCA, San Francisco, CA, USA, 12–16 February 2005; pp. 316–327.
11. Hammond, L.; Wong, V.; Chen, M.; Carlstrom, B.D.; Davis, J.D.; Hertzberg, B.; Prabhu, M.K.; Wijaya, H.; Kozyrakis, C.; Olukotun, K. Transactional Memory Coherence and Consistency. *SIGARCH Comput. Archit. News* **2004**, *32*, 102. [CrossRef]
12. Moore, K.E. LogTM: Log-Based Transactional Memory. In Proceedings of the HPCA, Austin, TA, USA, 11–15 February 2006; pp. 258–269.
13. Rajwar, R.; Herlihy, M.; Lai, K. Virtualizing Transactional Memory. In *Proceedings of the ISCA*; IEEE Computer Society: Washington, DC, USA, 2005; pp. 494–505. [CrossRef]
14. RSTM. Available online: http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml (accessed 14 February 2019).
15. Dalessandro, L.; Spear, M.F.; Scott, M.L. NOrec: Streamlining STM by Abolishing Ownership Records. In Proceedings of the PPOPP, Bangalore, India, 9–14 January 2010; pp. 67–78. [CrossRef]
16. Dragojevic, A.; Guerraoui, R.; Kapalka, M. Stretching Transactional Memory. In Proceedings of the PLDI, Dublin, Ireland, 15–20 June 2009; pp. 155–165. [CrossRef]
17. Zhao, J.; Li, S.; Yoon, D.H.; Xie, Y.; Jouppi, N.P. Kiln: Closing the Performance Gap Between Systems with and without Persistence Support. In Proceedings of the MICRO, Shanghai, China, 22–26 April 2013; pp. 421–432.
18. Coburn, J.; Caulfield, A.M.; Akel, A.; Grupp, L.M.; Gupta, R.K.; Jhala, R.; Swanson, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In Proceedings of the ASPLOS, Newport Beach, CA, USA, 5–11 May 2011; pp. 105–118.
19. Volos, H.; Tack, A.J.; Swift, M.M. Mnemosyne: Lightweight Persistent Memory. In Proceedings of the ASPLOS, Newport Beach, CA, USA, 5–11 May 2011; pp. 91–104.
20. Narayanan, D.; Hodson, O. Whole-system Persistence. In Proceedings of the ASPLOS, London, UK, 3–7 March 2012; pp. 401–410.
21. Liu, M.; Zhang, M.; Chen, K.; Qian, X.; Wu, Y.; Zheng, W.; Ren, J. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In Proceedings of the ASPLOS, Xi'an, China, 8–12 April 2017; pp. 329–343.
22. Wan, H.; Lu, Y.; Xu, Y.; Shu, J. Empirical study of redo and undo logging in persistent memory. In Proceedings of the NVMSA, Deagu, Korea, 17–19 August 2016; pp. 1–6.

23. Minh, C.C.; Chung, J.; Kozyrakis, C.; Olukotun, K. STAMP: Stanford Transactional Applications for Multi-Processing. In Proceedings of the IISWC, Seattle, WA, USA, 14–16 September 2008; pp. 35–46.
24. Poudel, P.; Sharma, G. An Adaptive Logging Framework for Persistent Memories. In Proceedings of the SSS, Tokyo, Japan, 4–7 November, 2018; pp. 32–49. [CrossRef]
25. Poudel, P.; Sharma, G. Adaptive Versioning in Transactional Memories. In Proceedings of the SSS, Pisa, Italy, 22–25 October 2019; pp. 277–295. [CrossRef]
26. Felber, P.; Fetzer, C.; Marlier, P.; Riegel, T. Time-Based Software Transactional Memory. *IEEE Trans. Parallel Distrib. Syst.* **2010**, *21*, 1793–1807. [CrossRef]
27. Felber, P.; Fetzer, C.; Riegel, T. Dynamic performance tuning of word-based software transactional memory. In Proceedings of the PPOPP, Salt Lake City, UT, USA, 20–23 February 2008; pp. 237–246.
28. Herlihy, M.; Luchangco, V.; Moir, M.; Scherer, , W.N., III. Software Transactional Memory for Dynamic-sized Data Structures. In Proceedings of the PODC, Boston, MA, USA, 13–16 July 2003; pp. 92–101.
29. Nguyen, D.; Pingali, K. What Scalable Programs Need from Transactional Memory. In Proceedings of the ASPLOS, Xi'an, China, 8–12 April 2017; pp. 105–118.
30. Liu, H.; Chen, D.; Jin, H.; Liao, X.; He, B.; Hu, K.; Zhang, Y. A Survey of Non-Volatile Main Memory Technologies: State-of-the-Arts, Practices, and Future Directions. *J. Comput. Sci. Technol.* **2021**, *36*, 4–32. [CrossRef]
31. Asadinia, M.; Sarbazi-Azad, H. Chapter One—Introduction to non-volatile memory technologies. In *Durable Phase-Change Memory Architectures*; Asadinia, M., Sarbazi-Azad, H., Eds.; Advances in Computers; Elsevier: Amsterdam, The Netherlands, 2020; Volume 118, pp. 1–13.
32. Damron, P.; Fedorova, A.; Lev, Y.; Luchangco, V.; Moir, M.; Nussbaum, D. Hybrid transactional memory. In Proceedings of the ASPLOS, San Jose, CA, USA, 21–25 October 2006; pp. 336–346. [CrossRef]
33. Kumar, S.; Chu, M.; Hughes, C.J.; Kundu, P.; Nguyen, A.D. Hybrid transactional memory. In Proceedings of the PPOPP, Manhattan, NY, USA, 29–31 March 2006; pp. 209–220.
34. Dalessandro, L.; Carouge, F.; White, S.; Lev, Y.; Moir, M.; Scott, M.L.; Spear, M.F. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In Proceedings of the ASPLOS, Newport Beach, CA, USA, 5–11 May 2011; pp. 39–52.
35. Alistarh, D.; Kopinsky, J.; Kuznetsov, P.; Ravi, S.; Shavit, N. Inherent limitations of hybrid transactional memory. *Distrib. Comput.* **2018**, *31*, 167–185. [CrossRef]
36. Ruan, W.; Spear, M.F. Hybrid Transactional Memory Revisited. In Proceedings of the DISC, Tokyo, Japan, 5–9 October 2015; pp. 215–231.
37. Riegel, T.; Marlier, P.; Nowack, M.; Felber, P.; Fetzer, C. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In Proceedings of the SPAA, San Jose, CA, USA, 4–6 June 2011; pp. 53–64.
38. Lev, Y.; Moir, M.; Nussbaum, D. PhTM: Phased transactional memory. In Proceedings of the Workshop on Transactional Computing (Transact), Portland, Oregon, 16 August 2007.
39. De Carvalho, J.P.L.; Araujo, G.; Baldassin, A. Revisiting phased transactional memory. In Proceedings of the ICS, Florence, Italy, 12–15 September 2017; pp. 25:1–25:10. [CrossRef]
40. De Carvalho, J.P.L.; Araujo, G.; Baldassin, A. The Case for Phase-Based Transactional Memory. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 459–472. [CrossRef]
41. The Persistent Memory Development Kit (PMDK). Available online: https://github.com/pmem/pmdk/ (accessed on 14 February 2019).
42. Dulloor, S.R.; Kumar, S.; Keshavamurthy, A.; Lantz, P.; Reddy, D.; Sankaran, R.; Jackson, J. System Software for Persistent Memory. In Proceedings of the EuroSys, Amsterdam, The Netherlands, 14–16 April 2014; pp. 15:1–15:15.
43. Avni, H.; Levy, E.; Mendelson, A. Hardware Transactions in Nonvolatile Memory. In Proceedings of the DISC, Tokyo, Japan, 5–9 October 2015; pp. 617–630.
44. Genç, K.; Bond, M.D.; Xu, G.H. Crafty: Efficient, HTM-Compatible Persistent Transactions. In *Proceedings of the PLDI*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 59–74. [CrossRef]
45. Joshi, A.; Nagarajan, V.; Cintra, M.; Viglas, S. DHTM: Durable Hardware Transactional Memory. In *Proceedings of the ISCA*; IEEE Press: New York City, NY, USA, 2018; pp. 452–465. [CrossRef]
46. Castro, D.; Baldassin, A.; Barreto, J.; Romano, P. SPHT: Scalable Persistent Hardware Transactions. In Proceedings of the FAST, Santa Clara, CA, USA, 22–25 February 2021.
47. Optane DC Persistent Memory. Available online: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html (accessed on 25 February 2021).
48. Baldassin, A.; Murari, R.; de Carvalho, J.P.L.; Araujo, G.; Castro, D.; Barreto, J.; Romano, P. NV-PhTM: An Efficient Phase-Based Transactional System for Non-volatile Memory. In *Proceedings of the Euro-Par*; Malawski, M., Rzadca, K., Eds.; Springer: New York City, NY, USA, 2020; Volume 12247, pp. 477–492.
49. Krishnan, R.M.; Kim, J.; Mathew, A.; Fu, X.; Demeri, A.; Min, C.; Kannan, S. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the ASPLOS*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 335–349. [CrossRef]

50. Gu, J.; Yu, Q.; Wang, X.; Wang, Z.; Zang, B.; Guan, H.; Chen, H. Pisces: A Scalable and Efficient Persistent Transactional Memory. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference, Renton, WA, USA, 10–12 July 2019; pp. 913–928.
51. Kolli, A.; Pelley, S.; Saidi, A.G.; Chen, P.M.; Wenisch, T.F. High-Performance Transactions for Persistent Memories. In Proceedings of the ASPLOS, Atlanta, GA, USA, 2–3 April 2016; pp. 399–411. [CrossRef]
52. Lu, Y.; Shu, J.; Sun, L.; Mutlu, O. Loose-Ordering Consistency for persistent memory. In Proceedings of the ICCD, Seoul, Korea, 19–22 Ocotober 2014; pp. 216–223. [CrossRef]
53. Memaripour, A.; Badam, A.; Phanishayee, A.; Zhou, Y.; Alagappan, R.; Strauss, K.; Swanson, S. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*; Association for Computing Machinery: New York, NY, USA, 2017; pp. 499–512. [CrossRef]
54. Izraelevitz, J.; Kelly, T.; Kolli, A. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In Proceedings of the ASPLOS, Atlanta, GA, USA, 2–3 April 2016; pp. 427–442.
55. Coburn, J.; Bunker, T.; Schwarz, M.; Gupta, R.; Swanson, S. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In Proceedings of the SOSP, Farmington, PA, USA, 3–6 November 2013; pp. 197–212.
56. Shin, S.; Tirukkovalluri, S.K.; Tuck, J.; Solihin, Y. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In Proceedings of the MICRO, Boston, MA, USA, 14–18 Ocotober 2017; pp. 178–190.
57. Chatzistergiou, A.; Cintra, M.; Viglas, S. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *PVLDB* **2015**, *8*, 497–508.
58. Lu, Y.; Shu, J.; Sun, L. Blurred Persistence: Efficient Transactions in Persistent Memory. *Trans. Storage* **2016**, *12*, 3:1–3:29. [CrossRef]
59. Keidar, I.; Perelman, D. Multi-versioning in Transactional Memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001*; Guerraoui, R., Romano, P., Eds.; Springer International Publishing: Cham, Switzerland, 2015; pp. 150–165. [CrossRef]
60. Sharma, G.; Busch, C. A Competitive Analysis for Balanced Transactional Memory Workloads. *Algorithmica* **2012**, *63*, 296–322. [CrossRef]
61. Sharma, G.; Busch, C. Window-Based Greedy Contention Management for Transactional Memory: Theory and Practice. *Distrib. Comput.* **2012**, *25*, 225–248. [CrossRef]
62. Attiya, H.; Gramoli, V.; Milani, A. A Provably Starvation-Free Distributed Directory Protocol. In Proceedings of the SSS, New York City, NY, USA, 20–22 September 2010; pp. 405–419.
63. Guerraoui, R.; Herlihy, M.; Pochon, B. Toward a Theory of Transactional Contention Managers. In Proceedings of the PODC, Las Vegas, NA, USA, 17–20 July 2005; pp. 258–264.
64. Ansari, M.; Luján, M.; Kotselidis, C.; Jarvis, K.; Kirkham, C.C.; Watson, I. Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering. In Proceedings of the HiPEAC, Munich Germany, 2–4 June 2009; pp. 4–18.
65. III, W.N.S.; Scott, M.L. Advanced contention management for dynamic software transactional memory. In Proceedings of the PODC, Las Vegas, NA, USA, 17–20 July 2005; pp. 240–248.
66. Schneider, J.; Wattenhofer, R. Bounds on contention management algorithms. *Theor. Comput. Sci.* **2011**, *412*, 4151–4160. [CrossRef]
67. Attiya, H.; Epstein, L.; Shachnai, H.; Tamir, T. Transactional Contention Management as a Non-Clairvoyant Scheduling Problem. *Algorithmica* **2010**, *57*, 44–61. [CrossRef]
68. Attiya, H.; Milani, A. Transactional scheduling for read-dominated workloads. *J. Parallel Distrib. Comput.* **2012**, *72*, 1386–1396. [CrossRef]
69. Keidar, I.; Perelman, D. On Avoiding Spare Aborts in Transactional Memory. *Theory Comput. Syst.* **2015**, *57*, 261–285. [CrossRef]