An Adaptive Logging Framework for Persistent Memories

Pavan Poudel and Gokarna Sharma(✉)

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

Abstract. Persistent memory is receiving a tremendous amount of attention recently from both academia and industry. Atomic and durable transactions have been studied to ensure *crash consistency* in persistent memory. However, whether to use undo or redo logging to execute those transactions is still a hotly debated topic. *Redo* logging seems appropriate for write-dominated workloads and transactions in high contention scenarios whereas *undo* logging 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 logging method between redo or undo to achieve better performance. In this paper, we argue that we can obtain the best of both worlds without the need of such a priori knowledge. Particularly, we present an *adaptive* logging framework that dynamically switches between redo and undo logging at runtime so that the performance is always better than that is obtained from a priori selection of either undo or redo logging. We formally model our framework, prove its correctness, and provide an extensive evaluation of it through a persistent memory emulation of TinySTM using 5 micro-benchmarks and 8 complex benchmarks from STAMP and STAMPEDE suites that are well-known and widely used in the literature. The results show significant benefits of our logging framework.

1 Introduction

Recent advancements in memory technology (such as phase change memory, STT-RAM, and memristors) suggest the possibility of non-volatile memory (NVM) devices that are fast and byte-addressable as dynamic random access memory (DRAM). Moreover, they are predicted to be more power-efficient than DRAM, yet non-volatile and cheap as hard disk drives (HDDs) [3]. Persistent memory can allow applications to access the data structures through a fast load/store interface, without first performing block I/O and then transferring data into memory based structures [6,20,21]. This feature is quite instrumental to avoid many overheads and drawbacks of block-oriented storage such as HDDs. Therefore, one of the most central issues in persistent memory is programming models that directly leverage persistence of the memory.

The challenge for any programming model designed for persistent memory is how to ensure consistency of the application data in the event of sudden power

T. Izumi and P. Kuznetsov (Eds.): SSS 2018, LNCS 11201, pp. 32–49, 2018.
https://doi.org/10.1007/978-3-030-03232-6_3

failure or system crash. This issue is commonly known as *crash consistency* and the existing research has quite focused on this issue [6,13,16,20,23]. A simply way to achieve crash consistency is to serialize multiple write operations when manipulating data structures. However, this hampers application performance due to inherent serialization. One common technique used in modern processors to avoid this problem is *reordering*, i.e., exploit parallelism through shuffling the execution of multiple write operations. However, if a failure occurs between two reordered writes, it is again difficult to guarantee consistency and the data structure could end up in an inconsistent state.

Further difficulty arises when persistency meets the growing number of cores. On the one hand, as data is already in persistent memory, it seems unnecessary and redundant to allocate another (duplicate) persistent storage for it. On the other hand, when an address is written, the new value must be exposed *atomically* with a new consistent and persistent state to ensure consistency of data. One way of guaranteeing this atomicity is by means of *locks*. However, locking has several drawbacks and bottlenecks when dealing with particularly the ever growing number of cores [10,18]. A method to achieve atomicity (without the use of locking) is through transactions studied heavily recently in the context of hardware/software transactional memory [10,18]. A *transaction* (in the context of persistent memory) is a sequence of operations on persistent memory that either all occur, or nothing occurs with respect to failures. If the execution of a transaction is interrupted, it is guaranteed, after system restart, to restore the consistent state from the moment when the transaction was started. The ideal goal is to maintain consistent persistent states without the use of locking and without duplicating data.

The prior persistent memory designs, e.g., [6,13,16,20,23], provide *atomic* and *durable* transactions to move the data from a consistent state to another consistent state supporting the ideal goal discussed above (i.e., do not allocate another duplicate persistent storage but duplicate *only* the data needed to maintain consistent states, when necessary). This guarantee is provided by requiring the transactions to write data to a log area (usually called *transaction log*) before updating the data in the original persistent memory locations. Notice that this logging only duplicates the data that a transaction is going to update in persistent memory (reducing significantly the overhead of allocating another duplicate persistent storage for whole data). Transaction logs are of two kinds:

- *Undo logs.* In this logging method, a transaction works by first copying the data in persistent memory locations to a log area (called *undo log*) in persistent memory, makes them durable, and then performs updates in-place in the original data locations. In the event the transaction fails, any modifications to original persistent memory locations are *rolled back* using the old data stored in the (undo) log area.
- *Redo logs.* In this logging method, a transaction copies data in each persistent memory location that it is going to read/write to a log area (called *redo log*), appends all its data updates to that log area, and makes them durable in persistent memory (different than original locations) before writing the

Table 1. A comparison of undo and redo logging in persistent memory [12,13,20,21]

Constraint	Undo logging	Redo logging
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 [12,21]	Reads are intercepted and redirected to the redo log area to read recent uncommitted data [12,13,21]
Persist ordering	Requires to ensure persist ordering for each memory write in a transaction [20]	Requires only one persist ordering for each transaction [13,20]
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 persistent memory using redo log

data back to original persistent memory locations. If the transaction fails, the updates in log area are simply discarded. Therefore, the writing of data to redo log in persistent memory and back to original persistent memory locations happens only when transaction commits.

Table 1 summarizes the advantages and disadvantages of undo and redo logging methods. Although both undo and redo logging for consistency in persistent memory are studied heavily in the literature [6,13,16,20,23], which logging method is better is still not clear and the previous studies provide contradictory conclusions. For example, consider two prominent previous work NV-Heaps [6] and Mnemosyne [20]. The authors of Mnemosyne [20] suggested using redo logging whereas the authors of NV-Heaps [6] and others [7,16] suggested using undo logging. There is no study that elaborates the performance gap between undo and redo logging with comprehensive practical evaluations, besides [21] which answers this partially. Looking at [21], redo logging seems appropriate for write-dominated workloads and high contention scenarios whereas undo logging seems appropriate for read-dominated workloads and low contention scenarios. However, this necessitates a priori knowledge on the workload and contention scenario to select a logging method to obtain better performance.

Contributions. We argue that we can obtain the best of both worlds without any a priori knowledge on workload and contention scenario. Particularly, we present an adaptive logging framework, which we call Adaptive, that dynamically switches the execution using either undo logging or redo logging at the runtime so that the performance on any workload (and contention scenario) is always better than that is obtained by executing the transactions using either

undo logging or redo logging selected a priori. For the experimental evaluation, we incorporate ADAPTIVE in TinySTM [8,9] through appropriate changes and modifications in the TinySTM execution model to emulate persistent memory. TinySTM is a well known software transactional memory (STM) implementation [18] that has been used for experimentation in both persistent and volatile memory settings. TinySTM has already implemented individually both undo and redo logging methods but only for DRAM settings. We extend (open source) TinySTM distribution 1.0.5 [2] to incorporate our ADAPTIVE framework as well as to emulate persistent memory support (as real persistent memory is not yet available [13]). We then run experiments using ADAPTIVE against a diverse set of benchmarks (5 micro-benchmarks and 8 complex benchmarks from STAMP and STAMPEDE benchmark suites [15,17]) widely used in transactional memory (TM) research in the literature [8–10].

We measure the performance of ADAPTIVE in terms of *total number of movement of data* by a transaction to and from persistent memory. The motivation behind this performance metric is as follows. It has been heavily advocated that persistent memories significantly outperform traditional DRAM due to low standby power and fast access speed [22,24]. However, persistent memories suffer from the *write endurance* problem, i.e., every persistent memory unit can sustain a very limited number of writes before it wears-out. The total number of writes to the persistent memory address can also be defined as the total number of movement of data to and from the memory address. To mitigate the endurance problem, the movements of data should be minimized.

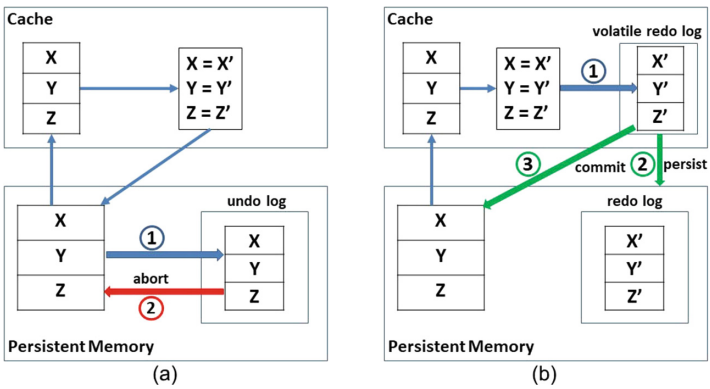

Fig. 1. An illustration of (a) undo and (b) redo logging methods in persistent memory.

Therefore, ADAPTIVE focuses on minimizing the total number of movements of data to and from the persistent memory by incorporating the best of both redo and undo logging frameworks switching dynamically. Specifically, for undo

logging, we measure the total number of movement of data between the original persistent memory locations and the undo log area, whereas for redo logging, we measure the total number of movement of data between the original persistent memory, volatile redo log area, and the persistent redo log area. Figure 1 illustrates these moves through ◯ steps for both undo and redo logging methods. The results suggest that, when using an *eager* version of redo logging, ADAPTIVE achieves up to 6× better performance than redo logging and up to 4.6× better performance than undo logging. When a *lazy* version of redo logging is used, ADAPTIVE again achieves up to 6× better performance than redo logging and up to 35× better performance than undo logging. The implication of our results is that switching between undo and redo logging dynamically at runtime provides a way to exploit positive aspects of both the logging methods, minimizing the total number of movements of data using undo or redo logging methods individually. This all is achieved with a minimal increase in total execution time, i.e., the execution time increase in ADAPTIVE is only at most 17% more than the total execution time using either undo or redo logging.

Related Work. The literature on redo and undo logging methods for crash consistency in persistent memory is vast. We discuss here only very closely related works. The most closely related work is due to Wan *et al.* [21], where they empirically evaluated redo and undo logging methods on the open source NVM library (NVML) [1] for some constrained workloads, and suggested that "*one logging method does not fit all workloads*". Particularly, they reported that (i) redo logging significantly outperforms undo logging for workloads in which a transaction updates large number of different objects, while it underperforms undo logging for read-dominated workloads, and (ii) undo logging is more sensitive to *read-to-write* ratios whereas redo logging is less sensitive to those ratios [21]. However, they did not consider the adaptive framework where logging method is dynamically switched at runtime. *Our framework provides the best of the both worlds without requiring a priori knowledge on the workload and contention scenario.*

The other works mostly proposed methods to provide crash consistency either through undo logging or through redo logging, and there is no work that elaborates the performance gap between undo and redo logging methods. Coburn *et al.* [6] suggested NV-HEAPS, a STM implementation for persistent memory using undo logging. The basic idea follows DSTM [10], in which transactional objects are stored in persistent memory. Each transaction T maintains a volatile read log and a non-volatile undo log. If a system failure occurs, T is aborted and the undo log, which is persistent, is used to reverse the changes of T. Volos *et al.* [20] suggested MNEMOSYNE for persistent memory using redo logging and derived from TinySTM [8,9]. We observed that NV-HEAPS [6] and MNEMOSYNE [20] drew absolutely opposite conclusions on whether undo or redo logging is better for persistent memory. The former prefers to use undo logging, and the latter opts to use redo logging. Our results suggest that a combination of both of them is better than using these methods individually. A salient feature of our method is it does not require any priori knowledge on workload and contention scenarios.

Recently, Avni *et al.* [3] studied hardware transactional memory (HTM) based transactions for consistency in persistent memory through redo logging. DudeTM [13] provided a technique to answer whether to use undo or redo logging through a framework 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 than ours and needs a shared shadow memory, besides persistent memory where that data is. The recent several papers, e.g., [4,5,11,12,14,16,19,23], provided techniques to improve the time to log the data (e.g., through coalescing, through persistent cache, through hardware support, through undo+redo logging methods, etc.) for both undo and redo logs. However, our focus is on taking a different approach of dynamically switching between undo and redo logging at runtime to exploit advantages of both the methods and our extensive experimental evaluation (Sect. 4) confirms this exploitation.

Paper Organization. We discuss the memory model in Sect. 2. We outline our adaptive logging framework in Sect. 3 and evaluate it in Sect. 4. Finally, we conclude in Sect. 5. Some experimental results are omitted due to space constraints.

2 Model

We consider a computer system with unlimited persistent memory, many processing cores, and no HDD. All persistent memory is cacheable 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. 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.

For redo logging, we make sure that all writes that are in volatile cache reach persistent log before a transaction commit, while all transactional writes stay in the cache. Moreover, to make sure that the last committed value is used in the restart process, we attach a version to each logged variable x. Note that the technique of verifying that x is logged only once in the system can also be used for this purpose. For undo logging, the data in persistent undo log is used in the restart process (no versioning required).

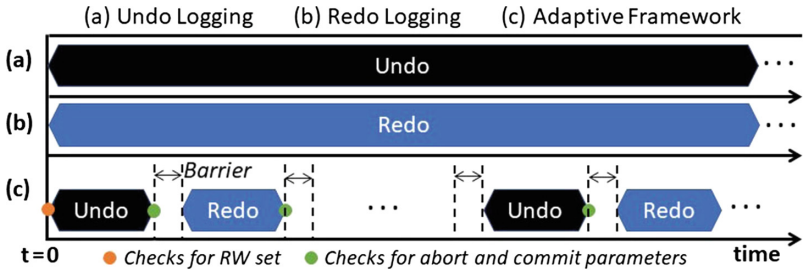

Fig. 2. An illustration of undo logging, redo logging, and adaptive logging methods. The barrier in ADAPTIVE is to let finish executing in-flight transactions before switching.

3 Adaptive Logging Framework

We now describe our adaptive logging framework, ADAPTIVE, that runs transactions using either undo or redo logging, switching between these two logging methods dynamically at runtime. In the existing persistent memory designs, e.g., [6,13,16,20,23], transactions execute using either redo logging or undo logging (without switching) selected a priori. Figure 2 compares ADAPTIVE with undo and redo logging. The pseudocode of ADAPTIVE is given in Algorithm 1.

Let T be a transaction that comes to the system at time $t \geq 0$. We assume that the execution starts at time $t_0 = 0$. In the following, we describe how ADAPTIVE schedules T using either undo logging or redo logging dynamically switching at runtime.

We need the following definitions. Let $N_{ucommit}, N_{rcommit}$ be the number of transaction commits in ADAPTIVE from time $t_0 = 0$ until the current time $t > t_0$ for transactions executed using undo logging and redo logging, respectively. Particularly, $N_{ucommit}$ ($N_{rcommit}$) counts the number of transactions that are committed in ADAPTIVE while running using undo (redo) logging method. Similarly, let N_{uabort}, N_{rabort} be the number of transaction aborts in ADAPTIVE from time $t_0 = 0$ until time $t > t_0$ for transactions executed using undo logging and redo logging, respectively. Furthermore, let N_{commit} and N_{abort} be the total number of commits and aborts in ADAPTIVE, respectively. We have that $N_{commit} = N_{ucommit} + N_{rcommit}$ and $N_{abort} = N_{uabort} + N_{rabort}$, respectively.

The idea in ADAPTIVE is to decide on which logging method to use for executing T based on the parameters $N_{ucommit}, N_{rcommit}, N_{uabort}$, and N_{rabort} learned from the system at runtime. However, if T comes to the system at time $t_0 = 0$, we have all $N_{ucommit}, N_{rcommit}, N_{uabort}$, and N_{rabort} zero. We treat this as a special case and rely on the size of the read and write sets of T to decide on which logging method to use. Let $Wset(T)$ be the *write set* of T which is essentially the persistent memory locations that T would modify while in execution. Similarly, let $Rset(T)$ be the *read set* of T which is essentially the persistent memory locations that T would read (but not modify) while in execution. We have that $RW(T) = Rset(T) + Wset(T)$, where $RW(T)$ denotes

Algorithm 1. Adaptive logging framework for a transaction T at any time $t \geq 0$.

```
1  N_ucommit ← number of commits until t for transactions executed using undo
   logging;
2  N_rcommit ← number of commits until t for transactions executed using redo
   logging;
3  N_uabort ← number of aborts until t for transactions executed using undo
   logging;
4  N_rabort ← number of aborts until t for transactions executed using undo logging;
5  N_commit ← N_ucommit + N_rcommit, N_abort ← N_uabort + N_rabort;
6  if N_commit + N_abort == 0 then
7      Wset(T) ← write set of transaction T;
8      Rset(T) ← read set of transaction T;
9      if Wset(T) is greater than Rset(T) then execute T using redo logging;
10     else execute T using undo logging;
11 if N_commit + N_abort > 0 then
12     AAR ← N_abort / (N_commit+N_abort), AAR_undo ← N_uabort / (N_ucommit+N_uabort),
       ACR_undo ← N_uabort / N_ucommit, ACR_redo ← N_rabort / N_rcommit;
13     if (AAR ≥ 2/3) ∨ ((ACR_undo > ACR_redo) ∧ (AAR_undo ≥ 2/3)) then
14         execute T using redo logging;
15     else execute T using undo logging;
```

the total number of persistent memory locations that T reads and modifies while in execution. Therefore, at $t_0 = 0$, if $Wset(T)$ is greater than $Rset(T)$, then T is executed using redo logging, otherwise using undo logging.

If T comes to the system 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. $AAR = \frac{N_{abort}}{N_{commit}+N_{abort}}$ denotes the *average abort ratio* of transactions in ADAPTIVE from time $t = 0$ until time t (using both redo and undo logging). $AAR_{undo} = \frac{N_{uabort}}{N_{ucommit}+N_{uabort}}$ denotes the average abort ratio of transactions in ADAPTIVE from time $t = 0$ until time t executed using undo logging. Furthermore, $ACR_{undo} = \frac{N_{uabort}}{N_{ucommit}}$ and $ACR_{redo} = \frac{N_{rabort}}{N_{rcommit}}$ denote the *abort to commit ratio* of transactions in ADAPTIVE from time $t = 0$ until time t using undo logging and redo logging, respectively. At any time $t \geq 0$, $0 \leq AAR \leq 1$ and $0 \leq AAR_{undo} \leq 1$.

At any time $t > t_0$ in ADAPTIVE, T is executed using redo logging if (i) $AAR \geq \frac{2}{3}$ or (ii) $ACR_{undo} > ACR_{redo}$ and $AAR_{undo} \geq \frac{2}{3}$. Otherwise, T is executed using undo logging. 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 is that it works on all the benchmarks we experimented our framework against. We now discuss how the switching threshold is computed.

Computing the Switching Threshold $\frac{2}{3}$. The idea we employ is to compute the number of data movements for redo and undo logging, separately, and switch

between these methods when the data movement increases. Ideally, we would like to use the logging method in ADAPTIVE 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. Let W_{undo} be the total number of operations of moving data (i) from the original persistent memory locations to the undo log area (again in persistent memory) and (ii) from the undo log area back to the original persistent memory locations. The first kind of moves are shown as ① in Fig. 1(a) and the second kind of moves are shown as ② in Fig. 1(a). The first kind of moves are always done in undo logging and the second kind of moves are done only when the transaction aborts. Therefore,

$$W_{undo} = (N_{commit} + 2N_{abort}) \cdot S. \tag{1}$$

Let W_{redo} be the total number of operations of moving data (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 in the volatile cache, 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 Fig. 1(b), and the second and third kind of moves are shown as ② and ③ in Fig. 1(b), respectively. The first kind of moves are always done in redo logging and the second and third kind of moves are done only when the transaction commits. Therefore,

$$W_{redo} = (3N_{commit} + N_{abort}) \cdot S, \tag{2}$$

Notice that a transaction can run using either undo or redo logging when $W_{undo} = W_{redo}$ as the selection of a logging method does not have impact on the total number of movements. Therefore, from Eqs. 1 and 2, we have that

$$(N_{commit} + 2N_{abort}) \cdot S = (3N_{commit} + N_{abort}) \cdot S \tag{3}$$

$$N_{commit} + 2N_{abort} = 3N_{commit} + N_{abort} \tag{4}$$

$$N_{abort} = 2N_{commit} \tag{5}$$

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

$$\frac{N_{abort}}{N} + \frac{N_{commit}}{N} \geq 1 \tag{6}$$

$$\frac{2N_{commit}}{N} + \frac{N_{commit}}{N} \geq 1 \tag{7}$$

$$\frac{N_{commit}}{N} \geq \frac{1}{3} \tag{8}$$

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 $\frac{2}{3}$, then $W_{undo} > W_{redo}$. Thus, ADAPTIVE switches execution to

redo logging when $\frac{N_{abort}}{N} \geq \frac{2}{3}$ (Line 13 of Algorithm 1) and stay with undo logging, otherwise.

Time Barrier Requirement and Design. The ideal scenario in ADAPTIVE is to let each transaction T run Algorithm 1 and decide which logging method (redo or undo) to use for it to execute individually based on the parameters it infers at runtime. For several benchmarks we experimented with, this works perfectly fine. However, for some benchmarks, this creates a problem as some transactions are still in progress using one logging method and when T executes using other logging method, the conflict detection and resolution mechanisms interfere, hampering consistency. Therefore, to handle this situation, we introduce a *time barrier* (as shown in Fig. 2) that helps to synchronize the transactions while switching from one logging method to another. Suppose currently transactions are running using undo logging. Let a new transaction T arrives and it decides to run using redo logging. Since there are transactions still running using undo logging, T waits until those transactions finish executing. We show later in the experimental results that the possible increase in total execution time is due to time barriers and this increase is minimal compared to the substantial reductions in the total number of data movements achieved in ADAPTIVE.

Correctness of ADAPTIVE: We provide the correctness proof showing that the algorithm discussed above behaves correctly even under faults, achieving crash consistency.

Theorem 1. *Algorithm 1 provides crash consistency.*

Proof. Consider a transaction T that arrives at time $t \geq 0$. Suppose T runs with undo logging (Line 10 or 15 from Algorithm 1). T maintains the undo log with unique transaction ID in the undo log area in persistent memory where the current records of memory locations accessed by T are stored. T then directly updates on those persistent memory locations. Now, consider any new transaction $T_1 \neq T$ that arrives at time $t_1 > t$. Suppose T_1 also runs with undo logging and at some time $t_2 > t_1$, T and T_1 both conflict. Now, the transaction T aborts and to rollback to the previous consistent state, the records stored in the persistent undo log are written back to the original memory locations accessed by T.

Suppose now that T_1 tries to execute using redo logging. Since T has arrived before T_1, T is already running with undo logging. Since T_1 satisfied for redo logging, T_1 has to wait until T finishes executing (either commit or abort). Since $t_1 > t$ and T_1 runs after T finishes its execution, there is no conflict between T and T_1 and barrier helps to synchronize the execution of T and T_1.

Finally, consider the power failure scenario. Let a transaction T is running using either undo logging or redo logging and suddenly, the power failure occurs. When the system is restarted, the persistent log area is scanned and replayed. With the persistent undo log records, the inconsistent memory locations are rolled back to the previous consistent states. With the persistent redo log records, the memory locations are updated with the latest committed values. Both the

log records are discarded after they are replayed. The incomplete log records are also discarded. □

4 Experimental Evaluation

We now evaluate the performance of ADAPTIVE using 5 micro-benchmarks and 8 complex benchmarks. The evaluation is performed in a STM-based implementation using TinySTM [8,9] modified appropriately to emulate persistent memory model described in Sect. 2. 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. We first describe in detail how the experimental platform is set up. We then describe how a persistent memory framework is emulated. We finally describe benchmarks and the results achieved. All the results presented in this section are the average of 10 experimental runs. Moreover, the results are for varying number of threads ranging from 1 to 16.

Experimental Setup. We developed a STM-based implementation using TinySTM [8,9]. TinySTM is a word-based STM that uses locks to protect shared memory locations. TinySTM has implemented separately both redo logging and undo logging methods (called *Redo* and *Undo*, respectively) 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 original memory locations of data until commit [8,9]. Furthermore, *Write_Back* design has two different implementations: *Write_Back_ETL* (also called *eager or encounter-time locking*) and *Write_Back_CTL* (also called *lazy or commit-time locking*). Encounter-time locking (ETL) detects transaction conflicts early at the time of memory 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 *Redo* in TinySTM: one based on *ETL* is called *Redo_ETL* and another based on *CTL* is called *Redo_CTL*.

We use *Redo_ETL* and *Undo* implementations to obtain an adaptive design, which we call *Adaptive_ETL*. Specifically, *Adaptive_ETL* uses *Redo_ETL* design of TinySTM as a redo logging method and *Undo* design of TinySTM as a undo logging method while executing Algorithm 1. Similarly, we use *Redo_CTL* and *Undo* implementations to obtain an adaptive design, which we call *Adaptive_CTL*. Therefore, we run experiments with five different designs *Redo_ETL*, *Redo_CTL*, *Undo*, *Adaptive_ETL*, and *Adaptive_CTL*, and compare, particularly, the results using *Adaptive_ETL* with *Redo_ETL* and *Undo* implementations, and the results using *Adaptive_CTL* with *Redo_CTL* and *Undo* implementations.

Persistent Memory Emulation. Persistent memory is not available yet (even for experimentation purposes) [13]. Therefore, we emulate it using DRAM in our experiments following previous works, e.g., [3]. We separate 500 MB region of DRAM for the persistent memory emulation. We use this region for keeping the persistent undo log when a transaction runs using undo logging and to persist the redo log when transaction runs using redo logging. 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 undo logging) and update (in case of commit but not yet written to memory in redo logging) operations will be handled by those persistent log records.

Benchmarks. We use both micro and complex benchmarks in the experiments. *Micro – Benchmarks:* We use 5 well-known and widely-used different micro-benchmarks, namely *bank, red black tree, hash set, linked list,* and *skip list* that are available in the TinySTM distribution [8,9] and used for experimentation in several papers, e.g., [10,13,21]. These micro-benchmarks simulate the basic concurrent access scenario for transactions with (relatively) small read/write sets.
STAMP: STAMP is also a well-known and widely-used benchmark suite. 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 [15].
STAMPEDE: Recently, Nguyen *et al.* [17] argued that the programming model and data structures used in STAMP benchmarks introduce performance bottlenecks. They modified them in a way the bottlenecks can be removed. They provided a set of rewritten STAMP benchmarks called STAMPEDE benchmarks. These are the same 8 STAMP benchmarks with the only difference on programming model and data structures.

Results on Micro-benchmarks. Figure 3 provides the experimental results for all 5 different micro-benchmarks. All the transactions in these benchmarks were run with *update rate* of 20%. When transactions were executed with small number of threads, we found that the transaction commit rate is higher than the transaction abort rate and the cost in redo logging is higher than the cost in undo logging. With the increase in number of threads, the abort rate is also increased. We noticed that *Redo_CTL* has consistently better performance than *Redo_ETL* on all the five micro-benchmarks. This is because the early detection of conflict and locking the memory address has increased the abort rate than the detecting conflict and locking the memory address at the commit time. *Adaptive_ETL* achieved up to 3.4× performance improvement compared to *Redo_ETL*. Similarly, *Adaptive_CTL* achieved up to 3× performance improvement compared to the *Redo_CTL*. Compared to *Undo*, *Adaptive_ETL* achieved up to 1.1× performance improvement and *Adaptive_CTL* achieved up to 1.3× performance improvement. Furthermore, *Adaptive_CTL* performed up to 2.5× better than *Adaptive_ETL*. The results show that ADAPTIVE always performs

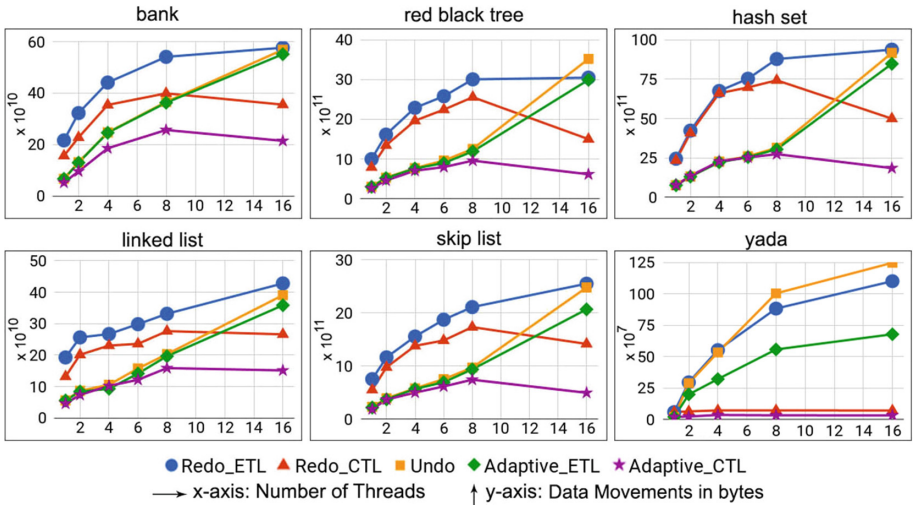

Fig. 3. An illustration of data movements in micro-benchmarks and *yada* from STAMP

better than *Redo* or *Undo*. We also noticed that *Adaptive_CTL* performs better than *Adaptive_ETL* in each micro-benchmark, since *Redo_CTL* performing better than *Redo_ETL*.

Results on STAMP Benchmarks. Figure 4 provides results for STAMP benchmarks. We found that when the transactions are executed with low number of threads, the transaction commit rate is higher and undo performs better than redo. This is due to low contention for memory access with small number of threads. With increasing number of threads, transaction abort rate also increases and undo starts to perform worse due to the frequent requirement of rollback. The results obtained for *genome* and *kmeans-low* show that undo starts to perform worse than redo beyond 8 threads. The same scenario starts beyond 4 threads in *Intruder* and *yada*. Moreover, we noticed that, irrespective of the abort rate change in redo and undo logging, ADAPTIVE always has better performance. Specifically, *Adaptive_ETL* achieved up to 6× performance improvement compared to *Redo_ETL* and up to 2× performance improvement compared to *Undo*. *Adaptive_CTL* achieved up to 3× performance improvement compared to *Redo_CTL* and up to 35× performance improvement (in *yada*) compared to *Undo*.

Results on STAMPEDE Benchmarks. Figure 5 provides the experimental results for STAMPEDE benchmarks. Similar to micro-benchmarks and STAMP benchmarks, ADAPTIVE has better performance compared to *Redo* or *Undo* in STAMPEDE benchmarks. *Adaptive_ETL* performed up to 3.6× better than the *Redo_ETL*. *Adaptive_CTL* performed up to 6× better than the *Redo_CTL*. Compared to *Undo*, *Adaptive_ETL* achieved up to 4.6× better performance and *Adaptive_CTL* achieved up to 3.1× better performance.

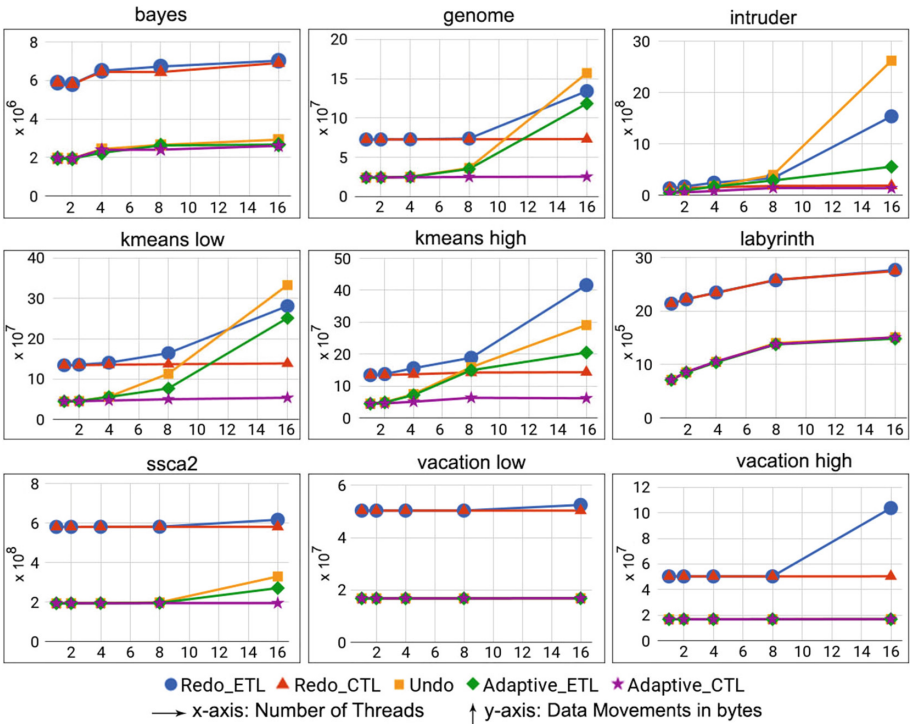

Fig. 4. An illustration of data movements in STAMP benchmarks

Execution Time and Throughput Results. The execution time is impacted in ADAPTIVE due to the switching between undo and redo logging at runtime. In most of the benchmarks, the increase in time is compensated as ADAPTIVE lowers the number of aborts. We were interested in what is the maximum increase on time in any benchmark we used in our experimentation. For micro-benchmarks, we measured *throughput* (instead of execution time) in terms of total number of transactions executed per second. This is because all 5 micro-benchmarks were executed for a fixed time interval of 10,000 ms and throughput is a natural performance parameter to examine the execution characteristic in this interval. All the 5 micro-benchmarks were executed with 5 different logging designs and the total number of transactions for each design were counted. The results obtained are omitted due to space constraints. We noticed that, in some applications, throughput of *Redo_ETL* is at most 16% more than the throughput of *Adaptive_ETL*. Throughput of *Redo_CTL* is at most 13% more than that of *Adaptive_CTL*. Throughput of *Undo* is at most 11% more than the throughput of *Adaptive_ETL* and at most 16% more than the throughput of *Adaptive_CTL*. These results imply that the throughput of ADAPTIVE is slightly decreased (less than 16%) compared to *Undo* or *Redo*. That means, the execution time for

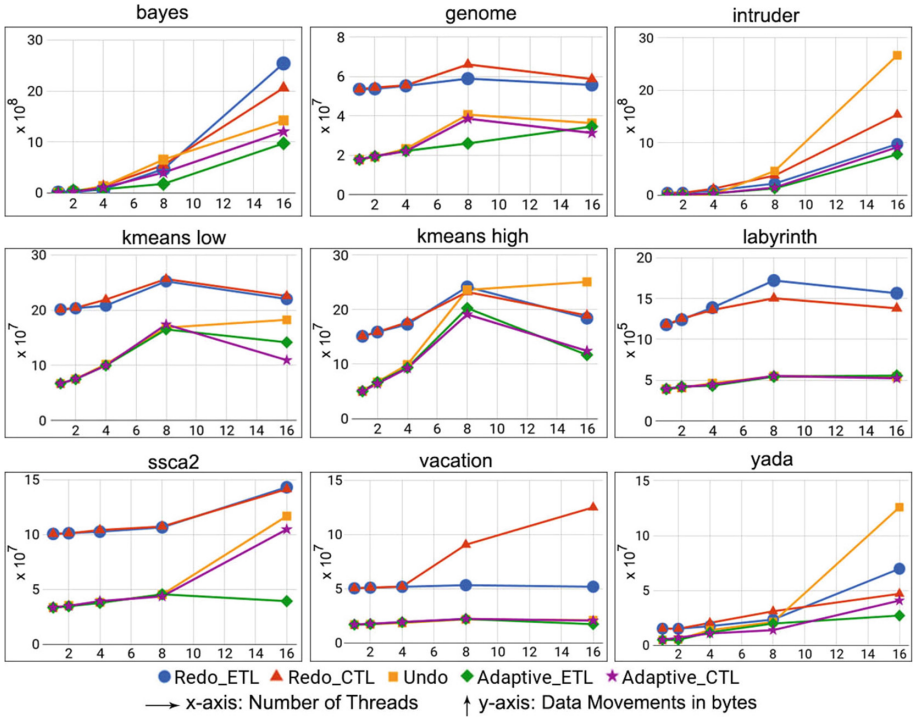

Fig. 5. An illustration of data movements in STAMPEDE benchmarks

the micro-benchmarks may increase by at most 16% in ADAPTIVE compared to *Undo* or *Redo*.

For the STAMP and STAMPEDE benchmarks, we measured the execution time for each of the applications. Figure 6 compares the execution time for STAMP benchmarks (the results for STAMPEDE are omitted due to space constraints). We noticed that the execution time in ADAPTIVE is at most 17% more compared to the execution time of *Undo* or *Redo*. As ADAPTIVE lowers the number of aborts, some applications (e.g. bayes, kmeans high, ssca2 in Fig. 6) have decreased execution time in ADAPTIVE than in *Undo* or *Redo* designs. We claim that the increase in execution time (decrease in throughput accordingly) for some applications is largely dominated by the performance improvement in terms of *total number of data movements.*

To summarize, in all of the cases, ADAPTIVE performs better for number of data movements compared to individual *Undo* and *Redo* designs, without increasing the execution time running using *Undo* and *Redo* designs. In some cases, the execution time increases but that is minimal compared to that of using *Undo* and *Redo* designs.

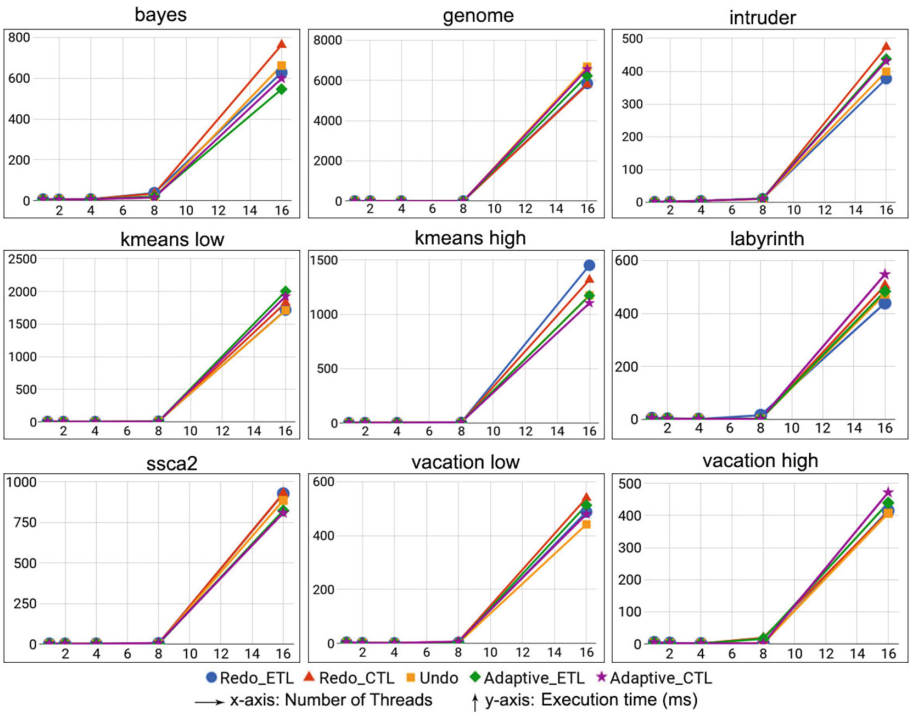

Fig. 6. An illustration of execution time for STAMP benchmarks

5 Concluding Remarks

Persistent memory is gaining much attention recently from both academia and industry. One of the most challenging issues in persistent memory is how to ensure consistency of the application data in the event of sudden power failure or system crash (commonly known as crash consistency). Redo and undo logging methods are the widely used techniques for maintaining crash consistency in persistent memory. However, they were studied separately and whether to use redo or undo logging (and which is in fact better) is still in hot debate. In this paper, we have presented an adaptive logging framework that dynamically switches between undo and redo logging methods at runtime to obtain the best of the both worlds. Our framework is quite simple and achieves significantly better performance (in terms of number of data movements addressing the write endurance problem) compared to undo and redo logging in 5 micro-benchmarks and 8 applications in STAMP and STAMPEDE benchmarks (with a minimal overhead in execution time). We believe our results and techniques will be helpful in choosing proper logging method for future consistency designs for persistent memories.

References

1. The Persistent Memory Development Kit (PMDK). https://github.com/pmem/pmdk/. Accessed 23 Feb 2018
2. TinySTM 1.0.5. http://tmware.org/sites/tmware.org/files/tinySTM/tinySTM-1.0.5.tgz. Accessed 23 Feb 2018
3. Avni, H., Levy, E., Mendelson, A.: Hardware transactions in nonvolatile memory. In: Moses, Y. (ed.) DISC 2015. LNCS, vol. 9363, pp. 617–630. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48653-5_41
4. Chatzistergiou, A., Cintra, M., Viglas, S.: Rewind: recovery write-ahead system for in-memory non-volatile data-structures. PVLDB **8**, 497–508 (2015)
5. Coburn, J., Bunker, T., Schwarz, M., Gupta, R., Swanson, S.: From ARIES to MARS: transaction support for next-generation, solid-state drives. In: SOSP, pp. 197–212 (2013)
6. Coburn, J., et al.: NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: ASPLOS, pp. 105–118 (2011)
7. Dulloor, S.R., et al.: System software for persistent memory. In: EuroSys, pp. 15:1–15:15 (2014)
8. Felber, P., Fetzer, C., Marlier, P., Riegel, T.: Time-based software transactional memory. IEEE Trans. Parallel Distrib. Syst. **21**(12), 1793–1807 (2010)
9. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPOPP, pp. 237–246 (2008)
10. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101 (2003)
11. Izraelevitz, J., Kelly, T., Kolli, A.: Failure-atomic persistent memory updates via JUSTDO logging. ASPLOS **44**, 427–442 (2016)
12. Kolli, A., Pelley, S., Saidi, A., Chen, P.M., Wenisch, T.F.: High-performance transactions for persistent memories. In: ASPLOS, pp. 399–411 (2016)
13. Liu, M., et al.: DudeTM: building durable transactions with decoupling for persistent memory. In: ASPLOS, pp. 329–343 (2017)
14. Lu, Y., Shu, J., Sun, L.: Blurred persistence: efficient transactions in persistent memory. Trans. Storage **12**(1), 3:1–3:29 (2016)
15. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: stanford transactional applications for multi-processing. In: IISWC, pp. 35–46 (2008)
16. Narayanan, D., Hodson, O.: Whole-system persistence. In: ASPLOS, pp. 401–410 (2012)
17. Nguyen, D., Pingali, K.: What scalable programs need from transactional memory. In: ASPLOS, pp. 105–118 (2017)
18. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
19. Shin, S., Tirukkovalluri, S.K., Tuck, J., Solihin, Y.: Proteus: a flexible and fast software supported hardware logging approach for NVM. In: MICRO, pp. 178–190 (2017)
20. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: ASPLOS, pp. 91–104 (2011)
21. Wan, H., Lu, Y., Xu, Y., Shu, J.: Empirical study of redo and undo logging in persistent memory. In: NVMSA, pp. 1–6 (2016)
22. Zhang, Y., Swanson, S.: A study of application performance with non-volatile main memory. In: 2015 31st Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10 (2015)

23. 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: MICRO, pp. 421–432 (2013)
24. Zhou, P., Zhao, B., Yang, J., Zhang, Y.: A durable and energy efficient main memory using phase change memory technology. In: SIGARCH Computer Architecture News, vol. 37, no. 3, pp. 14–23 (2009)