Mitigating Sync Overhead in Single-Level Store Systems

Yuanchao Xu*†, Hu Wan*, Zeyi Hou*, Keni Qiu*
*College of Information Engineering, Capital Normal University, Beijing, China
†State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{xuyuanchao, wanhu, houzeyi, qiukn}@cnu.edu.cn

ABSTRACT

Emerging non-volatile memory technologies offer the durability of disk and the byte-addressability of DRAM, which makes it feasible to build up single-level store systems. However, due to extremely low latency of persistent writes to non-volatile memory, software stack accounts for the majority of the overall performance overhead, one of which comes from crash consistency guarantees. In order to let persistent data structures survive power failures or system crashes, some measures, such as write-ahead logging or copy-on-write, along with frequent cacheline flushes, must be taken to ensure the consistency of durable data, thereby incurring non-trivial sync overhead. In this paper, we propose two techniques to mitigate the sync overhead. First, we leverage write-optimized non-volatile memory to store log entries on chip instead of off chip, thereby eliminating sync overhead. Second, we present an adaptive caching mode policy in terms of data access patterns to eliminate unnecessary sync overhead. Evaluation results indicate that the two techniques help improve the overall performance from 5.88× to 6.77× compared to conventional transactional persistent memory.

CCS Concepts

•Software and its engineering → Memory management; •Computer systems organization → Reliability;

Keywords

Sync overhead, non-volatile memory, crash consistency, single-level store

1. INTRODUCTION

From a user's perspective, it is expected to complete a particular operation (e.g., withdraw) atomically, namely "all or nothing". However, almost all the programmer-defined operations cannot be done through a single disk I/O or a single memory write operation, since their granularity are usually less than that of a programmer-defined operation. Hence, if a system crash or power failure occurs during updating a persistent data structure, part of the data may be lost, thereby making persistent data structures left in an inconsistent state due to the coexistence of new data and old data. This is an issue known as crash-consistency problem. For example, in order to complete a user operation, we have to update two persistent data structures in order, A and B. If a failure occurs after A is persisted but before B is persisted, data inconsistency will happen. Unfortunately, all kinds of failures, such as software failures, hardware failures, and power failures, are nearly impossible to be avoided [1, 2, 3].

In recent years, byte-addressable non-volatile memory technologies, also called *persistent memory* (PM), including phase change memory (PCM) [4] and spin-transfer torque random access memory (STT-RAM) [5], promise the fusion of volatile memory and persistent storage [17]. This allows system architects to build a single-level store system through using persistent memory entirely to replace both working memory (i.e., DRAM) and persistent storage (i.e., hard disk drives or solid state drives) in traditional two-level storage systems. By contrast, single-level store systems avoid data movement between volatile primary memory and non-volatile secondary storage, thereby yielding better I/O performance. However, to achieve this goal, it still requires disruptive software changes, including programming model, compiler, and operating system [20, 24], since traditional block-oriented file system cannot exploit the potential of byte-addressability of persistent memory.

Instead of issuing reads/writes on files via POSIX-based file system interface, applications can access persistent memory directly via load/store instructions. Due to byte addressability of persistent memory, *transactional memory* (TM) appears to be a viable alternative in ensuring the consistency of data shared among transactions [13]. TM has many different implementations for *volatile* memory and *persistent* memory [25, 27]. In this work, we only focus on **transactional persistent memory**, which can ensure correct recovery after a failure. Each transaction requires two writes to persistent memory for each update: one for storing the copy of the data (i.e, log) for recovery, and another for writing the data itself. To ensure the consistency of durable data in persistent memory upon a failure, it still requires flushing the dirty cachelines out of the cache hierarchy timely after normal store instructions by issuing sync operations (i.e., *clflush* and *sfence*). This will result in a significant sync overhead [29, 30], and thus has become the performance bottleneck for single-level store systems. Unfortunately, sync is

CF'16, May 16 - 19, 2016, Como, Italy

ISBN 978-1-4503-4128-8/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2903150.2903161

an imperative operation in enable crash consistency, including transactions committing, ordering constraints, and data synchronization.

A lifetime of a transaction can be divided into two main phases, logging phase and checkpoint phase. We observe that sync overhead associated with these two phases may be further mitigated. In this work, we propose two techniques, *on-chip logging* and *adaptive caching mode*. The key idea of on-chip logging is to store log entries on chip by leveraging software-managed yet non-volatile memory. The key idea of adaptive caching mode is to choose the suitable caching mode in terms of data access patterns.

Our contributions are summarized as follows:

- We use write-optimized non-volatile memory to store log entries on chip, thereby eliminating sync overhead and reducing persist latency compared with traditional off-chip logging.
- We present an adaptive synchronization policy based on caching mode in terms of data access patterns to eliminate unnecessary cachelines flushing to persistent memory.

We implement and evaluate two techniques. The results indicate that the two techniques help improve the overall performance up to 6.77× on average compared to conventional transactional persistent memory implementation.

2. BACKGROUND

In this section, we target single-level store systems, and discuss the problems related to crash consistency, including TM, and sync overhead.

2.1 Software Stack for Single-Level Store

Compared with two-level storage model, single-level store has obvious advantage in low power and high performance. The execution time and power assumption can be reduced by about two orders of magnitude [20]. However, there is a long way to go for single-level store systems due to the key challenge from software stack. Because existing storage software stack is designed on the assumption of low-speed and block-based storage media like hard disk drives and solid state drives, it is inefficient to cope with high-speed storage media like persistent memory. Some subsystems in operating system, including file system and virtual memory, should be optimized and even redesigned for single-level store systems [24] for at least the following three reasons. First, traditional virtual memory subsystem assumes that DRAM is volatile, yet unlimited endurance. In contrast, some emerging memory technologies (e.g., PCM) offer data persistence but limited endurance. Second, as for single-level store systems, data movement between primary memory and secondary storage is avoided, since paging fault does not happen any more. Third, user applications can access persistent memory like DRAM directly via load/store instructions. Thus, file system may be simplified drastically compared to block-based device file systems as shown in Fig. 1 [10].

2.2 Consistency and Transactional Memory

Crash consistency needs to ensure atomicity, consistency, and durability of data. In traditional file systems, there are

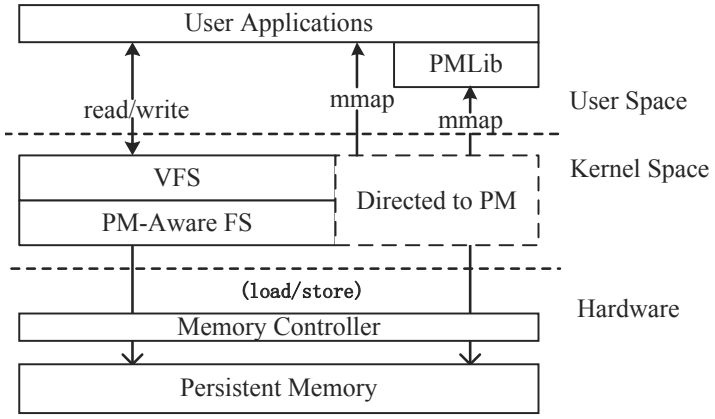

Figure 1: Single-level store architecture using persistent memory.

two typical techniques, including write-ahead logging (WAL) [12] and copy-on-write (COW) [22], to provide the support of data consistency. They are widely used in almost all the existing file systems. For example, ext4 file system adopts WAL, while Btrfs uses COW. The newly proposed file system, PMFS [10], uses WAL for metadata and COW for user data, respectively. Although file systems for persistent memory still use these two consistency techniques, they should be improved since existing implementations are not optimal, particularly for those high performance transaction processing systems.

Transactional persistent memory [27] is considered as an ideal way in persistent memory to enable data consistency upon a failure. Traditional TM [13] only ensures atomicity, isolation, and consistency of a group of memory operations. To ensure that the data within a transaction can be persisted to memory after this transaction is committed, it requires adding durability to TM. This is called transactional persistent memory, which is identical with database transactions. The durability has two implications. First, the memory must be non-volatile so that the data can retain after system reboots. Second, the data must be written back to memory in correct persist order for recovery. TM can be implemented at the software or hardware level. Hardware TM may comprise modifications in processors, cache, and bus protocol to support transactions. Software TM [11], such as Mnemosyne [25] and NV-Heaps [8], can provide TM semantics in a software runtime library or programming language, thus requiring minimal hardware support at the cost of performance penalty.

In short, transactional persistent memory mechanism can not only ensure crash consistency at the granularity of transaction, but also achieve high concurrent transaction processing. However, hardware complexity is a great challenge.

2.3 Synchronization Overhead

We assume that there are only volatile cache and persistent memory in our target system. In this scenario, in order to make data durable, we must flush the dirty cachelines back into persistent memory by manually issuing sync instructions. Existing legacy applications access memory under memory consistency model, which only ensures that the data are globally visible to all cores. In other words, memory consistency model only ensure that the data reach last level cache instead of off-chip memory. Since cache is

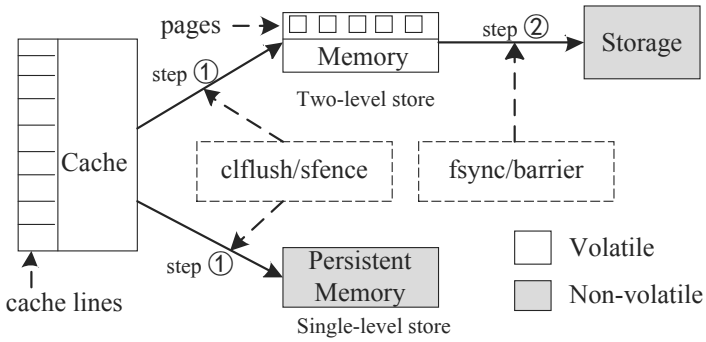

Figure 2: Sync paths comparison between two-level storage model and single-level store model.

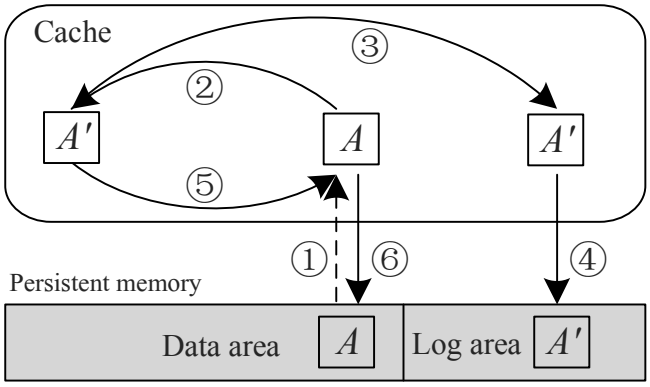

Figure 3: Transaction phases in traditional transactional persistent memory.

volatile, dirty cachelines must be flushed back to persistent memory in case of a failure. In addition, sync can enforce persist ordering constraints for recovery after a failure.

In a transaction system, sync operations occur frequently for at least two reasons. First, when a transaction is committed, it requires to issue a sync operation to ensure ordering constraints. Second, during the phase of checkpoint, sync is used to make sure that all the data in the data area is in the newest version. As a result, frequent sync operations incur significant performance costs. We evaluate the performance difference of FS-Mark benchmark between with sync enabled and sync disabled. Astonishingly, if sync is enabled, the performance degrades dramatically about 9× on average.

3. MOTIVATION

In our target system, we observe that sync overhead in logging phase and checkpoint phase can be partially eliminated or reduced.

3.1 Overview of Sync Overhead

In traditional two-level storage model, the sync overhead derived from two steps as shown in Fig. 2. The first step is to flush dirty cachelines back into volatile working memory (i.e., DRAM). The second step is to write modified dirty pages back to persistent storage. Due to low-speed of hard disk drives or solid state drives compared to DRAM, the second step becomes the main source of sync overhead for two-level storage systems. However, under single-level store systems, since it does not exist to write dirty pages to secondary storage media, the overhead of data movement between memory and storage is eliminated. Hence, flushing cachelines out of the cache hierarchy turns out to be the main source of sync overhead in single-level store systems.

TM usually uses write-ahead logging mechanism to enable recovery. As Fig. 3 shows, the data associated with a transaction are first read from the *data area* in persistent memory to the CPU cache (Step ❶), and then are copied to the *execution area* [18] (Step ❷). Otherwise, the generated new-version data (A')may be evicted to the persistent memory and overwrite the old-version data (A) in the *data area*, thereby violating the atomicity property of transactions. When a transaction is committing, the executed data are copied to the *log area* (Step ❸) for logging persistence (Step ❹). After the log has been persisted, the committed data are checkpointed to the *data area* (Step ❺) for checkpoint persistence (Step ❻). Therefore, each data update has been written into persistent memory twice, thus worsening the sync overhead.

3.2 Sync Overhead in Logging Phase

For each transaction, logging phase must be done completely before checkpoint phase starts in order to ensure the atomicity of all the data within a transaction. Therefore, logging phase should be desired as short as possible. As Fig. 3 shows, each log entry needs to be flushed from CPU cache to off-chip persistent memory in order to make log entries durable (Step ❹). Unfortunately, it incurs non-trivial sync overhead. In practice, log entries can be stored in any non-volatile area only if the data can be recovered correctly from log area after a failure. Once all the data within a transaction are checkpointed to their home locations, the log entries related to this transaction will become useless and may be discarded immediately. Thus, too large log space is not essential, thereby making it feasible to store all the log entries on chip. We will examine it in Section 5.1.2. In addition, prior work has already demonstrated that separating log area from data area helps improve performance. This approach is called external journaling [14].

We thus propose on-chip logging technique for transactional persistent memory to eliminate the sync overhead in logging phase. The details will be discussed in Section 4.1.

3.3 Sync Overhead in Checkpoint Phase

There are three primary approaches to reduce sync overhead. The first approach is to reduce the number of sync operations by identifying and then eliminating unnecessary synchronization operations [18, 29]. The second approach is to reduce the overhead of single sync operation by skipping those unmodified cachelines [16]. The third approach is to reduce persist ordering constraints by relaxing unnecessary ordering constraints from the perspective of memory persistency model [9, 19, 21].

To date, there are two primary methods to relax persist ordering constraints. The first method is to utilize novel synchronization instructions, including *clflushopt*, *clwb*, and *pcommit*. *clflushopt* provides unordered version of *clflush*. *clwb* does not invalidate the cacheline when writing back modified data of a cacheline. *pcommit* commits data writes queued in the memory subsystem to persistent memory serially. The second method is to leverage relaxed memory

persistency model such as *epoch persistency* [9, 15] and *s-trand persistency* [21].

We also aim to mitigate sync overhead by reducing the number of sync operations. We observe that different data have various memory access patterns, e.g., append-only data or overwrite data. Likewise, each caching mode, e.g., *write-back* (WB), *write-through* (WT), *uncacheable* (UC), or *write-combining* (WC), has different sync overhead [7]. For instance, from the perspective of data locality, WB mode can yield better performance than UC mode. However, WB is not always the best choice any more for all the applications, particularly for applications with low data locality, since WB mode still requires extra sync operations to make data durable, whereas UC mode does not.

Therefore, we propose an adaptive caching mode policy in checkpoint phase to reduce the number of sync operations by eliminating unnecessary cachelines flush.

4. DESIGN OVERVIEW

In this section, we discuss how to eliminate sync overhead by leveraging on-chip logging in logging phase, and also how to reduce sync overhead by adopting adaptive caching mode in checkpoint phase. Fig. 4 depicts the architectural overview of logging phase and checkpoint phase for a transaction in single-level store systems. Our two proposed techniques are denoted by dotted lines in this figure.

4.1 On-chip Logging

On-chip logging is used to eliminate sync overhead in logging phase through storing the copy of data (i.e., log) in the software-managed on-chip non-volatile memory. As we know, log data are only used as recovery information of real data after a failure. If all the data within a transaction are updated to their home locations, all the log entries related to this transaction will not be used for ever and thus can be discarded at any time. Therefore, theoretically speaking, log area may be placed on chip only if the required memory is not too large. Since on-chip logging mechanism directly persists log entries to on-chip non-volatile memory instead of off-chip persistent memory, it not only significantly reduces memory access latency compared to off-chip logging but also eliminates sync overhead incurred by flushing cachelines to off-chip memory. Although previous work [14] also mentioned to leverage external device for logging, this device is still off chip rather than on chip. In addition, since log is read-only (when recovery is needed) and append-only, there is no point occupying CPU cache space, which is only beneficial to applications with good data locality. More importantly, log will pollute cache. Therefore, we employ special non-volatile memory instead of cache (non-volatile) to store all the log entries of both user data and metadata in logging phase. This is so-called on-chip logging mechanism, which helps eliminate sync overhead and reduce write latency. The detail will be discussed in Section 5.1.

4.2 Adaptive Caching Mode

Adaptive caching mode is used to reduce sync overhead in checkpoint phase through choosing the most suitable caching mode for different data with various access patterns. We are motivated by the fact that not all the data have identical locality features and reliability requirements. Each caching mode has different impact on the data locality, while it incurs different sync overhead since each caching mode has different degree of ordering constraints. For example, WT mode is stronger in persist ordering constraints than WB mode. In modern processors, the caching mode can be specified at the page-level granularity of system memory. Unfortunately, almost all the existing operating systems adopt a single caching mode for all the data of arbitrary applications. As a consequence, it is difficult to meet various locality and reliability demands for those workloads with various data access patterns. Thus, we propose an adaptive caching mode mechanism, which dynamically determines a relative suitable caching mode by trading off between data locality and reliability requirement. We will discuss it in detail in Section 5.2.

4.3 Comparison of Sync overhead

Fig. 5(a) shows the sequence of conventional transactional persistent memory, which persists log entries to off-chip persistent memory. Since memory store instructions only ensure that the log entries reach last level cache according to memory consistency model. The log entries cannot reach persistent memory unless the programmer enforces issuing *clflush* and *sfence* instructions. In a transaction, all the logs must be flushed into persistent memory first before this transaction is committed. The system can then update the persistent data structures located in their home locations in place. Like logging, it also requires to use flush and fence operations to ensure the updated real data durable and consistent. As a result, it needs two sync operations for each update to ensure crash consistency. Fig. 5(b) shows the timeline of our proposed design using on-chip logging and adaptive sync. Since log area is on-chip, log can directly persist to non-volatile memory, it does not need issuing flush and fence operations, thereby eliminating the sync overhead in logging phase. In the checkpoint phase, we use adaptive sync based on caching mode instead of a single caching mode (WB mode is the default). Since some caching modes do not need sync operations, e.g., WT or UC, it reduces unnecessary sync operations, thereby reducing the sync overhead in checkpoint phase. With our proposed two techniques, the sync overhead in transactional persistent memory system is significantly mitigated.

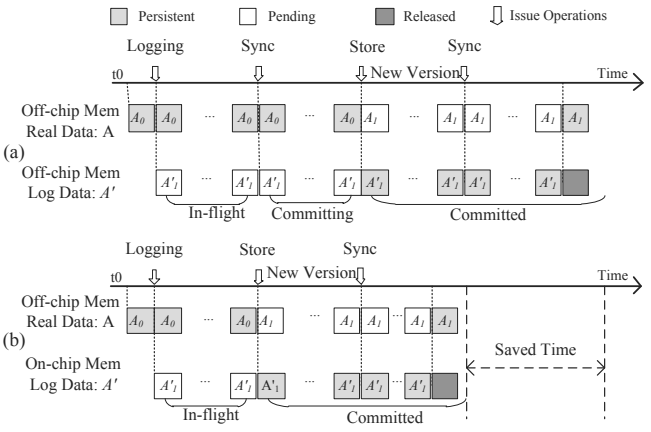

Figure 5: Comparison of the timeline. (a) Conventional design. (b) Our proposed design with on-chip logging and adaptive caching mode. Block *A* represents an old-version data block. Block *A'* represents the new-version data block.

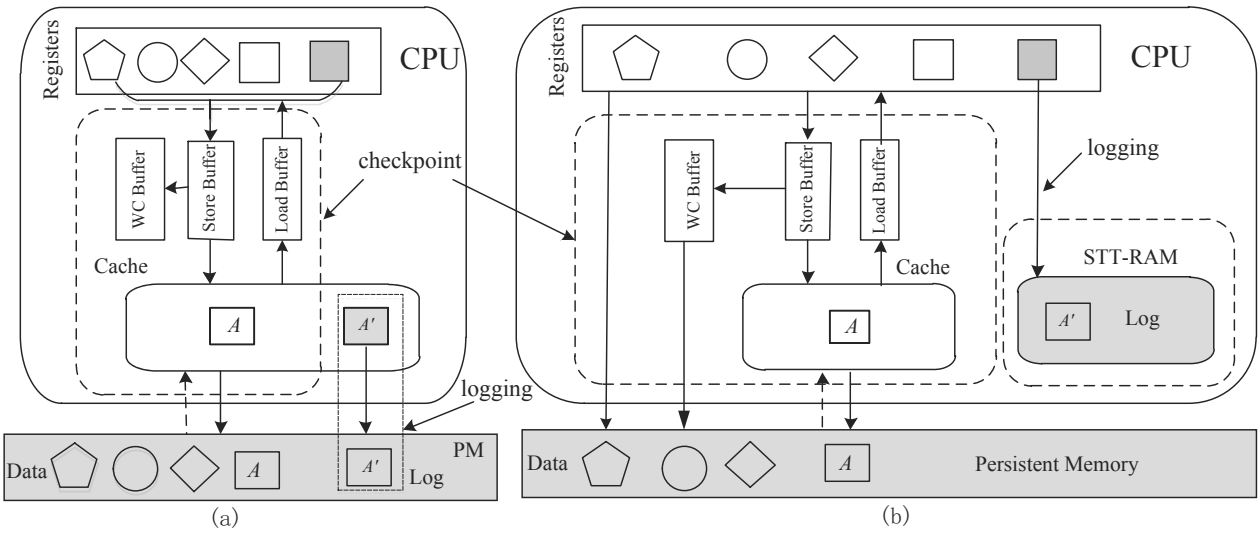

Figure 4: Architectural overview of logging phase and checkpoint phase for a transaction in single-level store systems. (a) Traditional transactional persistent memory. (b) Our proposed scheme with on-chip logging and adaptive caching mode.

5. IMPLEMENTATION

In this section, we discuss the implementation details about on-chip logging and adaptive caching mode.

5.1 On-chip Logging

5.1.1 Individual On-chip Memory for Logging

Recent research work [28] advocates that STT-RAM may replace SRAM as CPU cache due to similar performance and endurance to SRAM. Moreover, since STT-RAM is non-volatile, Kiln [30] leverages it as non-volatile cache to enable crash consistency without the requirement of consistent guarantee techniques such as WAL or COW. However, it needs to modify cache architecture, thereby making hardware more complex. Unlike Kiln, We use STT-RAM as individual on-chip memory to store log entries of transactions without involving any modification to CPU cache. Meanwhile, since log data are append-only or read-only once (when recovery is needed), all the log data bypass CPU cache and are written into STT-RAM directly. With individual non-volatile memory, it not only avoids the pollution to the data in CPU cache, but also reduces memory access latency compared to off-chip logging. According to log data's characteristics, we may adopt write-optimized policy to further reduce write latency and energy since retention time of data in STT-RAM is related to write latency and energy [17, 23]. Log is only a temporary copy of real data, hence, we may reduce the number of iterations of writes, thereby making STT-RAM behave like volatile memory.

Individual on-chip memory is similar to traditional scratch-pad memory technology in embedded systems [6], however, we use on-chip memory to store log data instead of further improving data locality. Thus, the memory technology must be non-volatile to make log data durable.

5.1.2 Software Management

Different from CPU cache, STT-RAM in this work is visible to operating system. Its address space is a part of the whole memory address space. In other words, on-chip STT-RAM and off-chip persistent memory are unified addressing. Moreover, STT-RAM does not have tag array and relevant comparison logic like CPU cache, thus it is more energy-efficient and area-efficient than CPU cache. Operating system needs to differentiate logging operations from normal write operations. From the perspective of user applications, the location of log area is transparent, meanwhile, legacy user codes are not required to be modified.

The space size of STT-RAM depends on the degree of lazy checkpoint. Once all the data within a transaction are written back to their home locations successfully (i.e., checkpoint), log data associated with this transaction may be discarded immediately. If the log space is not enough, operating system can enforce doing checkpoint so as to free a part of log space. In addition, STT-RAM is about 4× higher in storage density than SRAM [5]. We evaluate the impact of log space size on performance in Section 6.2. The result shows that 64MB is relatively suitable in terms of performance per area if chip area constraint is also taken into account. 64MB STT-RAM is about equivalent to 16MB SRAM in chip area. Thus, the problem about the log space size on-chip may be mitigated.

STT-RAM is only used as log area to store the log entries. A typical layout of log area is shown in Fig. 6. The structure of each log entry is cacheline (e.g., 64B) aligned, which includes *transaction identifier*, *data address*, *data type*, *data size*, and *data value*. Each log entry can store up to 50 bytes data, while one page can contain up to 64 log entries. When a new transaction is created, a 4 KB free page in log area is allocated to store log entries associated with this transaction. If the number of log entries within a transaction exceeds this limit, a new free page will be allocated to that transaction. Due to transaction concurrency, log entries across different transactions may be written to STT-RAM simultaneously, thus the address space belonging to the same transaction may be discontinuous. Aside from persistent data structures, some volatile data structures are also needed to manage individual on-chip memory. For instance, it is required to record the pointer of the last log entry of each

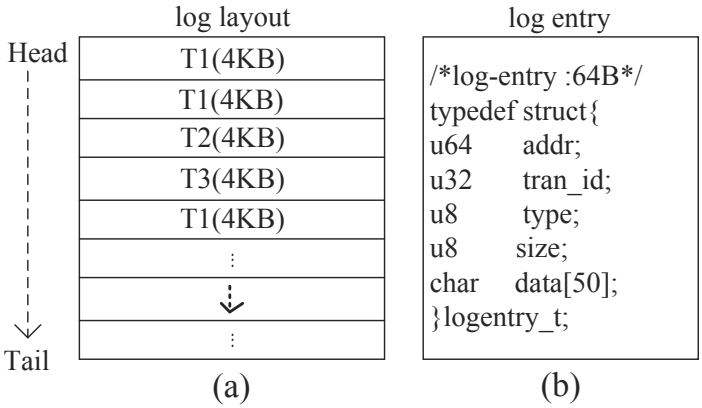

Figure 6: A typical layout of log area and structure of each log entry.

committing transaction. In short, the management of persistent memory needs support from both file system subsystem and virtual memory subsystem.

5.1.3 Transaction Recovery

A transaction includes three states: uncommitted, committed, and checkpointed. If a transaction is committed, the *type* field of the last log entry is marked as **committed_t**, or else marked as **committing_t**. After a transaction is committed successfully, its state will be transferred into checkpointed, which is recorded in a volatile data structure. Generally speaking, we expect to do checkpoint as lazy as possible in order to reduce the number of sync operations via persistent coalescing and data caching. However, due to the limited space of on-chip memory including CPU cache and STT-RAM, checkpoint should be done timely in order to release the occupied log space.

After system restarts, recovery module begins to scan the log area from head to tail. Uncommitted transactions will be skipped. For committed transactions, the address and value pair of each log entry is first read out from log area, and then used to update the corresponding address of persistent memory in place with the value. Finally, when all the committed transactions are processed completely, the whole log area will be free. If some committed transactions have already been checkpointed but their log spaces have not been released due to sudden power loss or system crash, the recovery process for those transactions will be restarted to ensure the completeness of checkpoint.

5.2 Adaptive Caching Mode

5.2.1 Caching Modes

Most modern processors provide several different caching modes. For example, Intel processors support up to six caching modes, including UC, WC, WT, and WB. Of these four typical caching modes, WT mode and WB mode belong to *cacheable mode*, whereas UC mode and WC mode belong to *uncacheable mode*. They have different data locality property and ordering constraints degree. WB mode can be cached and written back whenever it is forced. WT mode allows all the data to be cached and also be written straight out to memory simultaneously. In other words, memory is kept up to date with the CPU cache. UC simply cannot be cached. WC is like UC, except that writes can be coalesced before being sent out to memory. Each caching mode has different sync requirements, thus leading to different consistency guarantee overhead due to adopting different degree of ordering constraints. In existing systems, WB is used as the default caching mode for the whole memory regions.

5.2.2 Data Diversity

Each caching mode implies different degree ordering constraints, and thus provides different consistency guarantee at the different overhead. Likewise, various data have different consistency requirements and locality characteristics. For example, some metadata, such as persistent data structure *inode table* in the file systems, have been updated frequently, whereas user data may be written back to persistent memory just only once. From the perspective of data locality, we can simply believe that WB mode is better than UC mode since WB mode reduces the number of off-chip memory accesses. However, under transactional persistent memory systems, it is not sure which mode is better, since all the data should be written back to off-chip persistent memory in order. With UC mode, it naturally provides strict persist ordering constraints. With WB mode, we must flush dirty cachelines to persistent memory by using sync instructions explicitly. Kumud *et al.* [7] also concluded that WB mode is not always optimal for all the data. Thus, it is desired to choose relative suitable caching mode for different data by trading off between reliability and performance.

5.2.3 Adaptive Policy

Identifying data access pattern online is a great challenge for any adaptive policy. In this work, we adopt a sampling-based approach. First, when a new page is allocated, we use WC mode as the default caching mode. Then, during the sampling phase (e.g., 10s), the transaction system will accumulate the number of writing data to the same page at the page-level granularity as transactions are running. Next, at the end of sampling phase, the transaction system calculates the average value of the total writes (*avg_w*) for all pages related to transactions. If the number of writes for a page is less than *avg_w*, the caching mode for this page will be set to UC mode. After a period of time (e.g., 30s), sampling phase restarts again. If the number of writes for this page is more than *avg_w*, the caching mode for this page will be set back to WC mode. For those memory regions via non-transactional writes, we choose WB as the default caching mode for performance rather than reliability. For the simplicity of implementation, we only consider three caching modes in this work. In addition, software hints appears to be a viable alternative due to its simplicity, though it may require to modify legacy user applications.

5.2.4 Hardware/Software Support

For Intel x86 architecture, we can specify a caching mode at the page-level granularity through modifying PAT (7th bit), PCD (4th bit), and PWT (3rd bit) in the page table entry [7]. We implement multiple versions of sync operations by using the three instructions, including flushing cachelines (*clflush*), waiting until completion (*sfence*) and draining write combining buffer (*pcommit*). For UC mode, they are not required to issue sync operations. For WC mode, only *pcommit* instruction is required for most platforms except for Intel CPU with ADR feature. Although Intel provides novel sync instructions, such *clflushopt* and *clwb*,

the CPUs supporting new instructions are not yet available. Hence, we do not consider the optimization of critical sync path in transactional persistent memory using these new instructions in this work.

6. EVALUATION

In this section, we first evaluate the effects of on-chip logging and adaptive caching mode, respectively. Then, we evaluate the overall performance. The baseline system is a general TM implementation based on persistent memory with a single WB mode.

6.1 Experimental Setup

Platform. All of our experiments are conducted on a Linux server (kernel version 3.11.0) with an Intel Xeon E5-2680 2.5GHz CPU and 16GB DRAM. The single-level store system in our evaluation is based on the open-source Linux *non-volatile memory library* (NVML) and PMFS developed by Intel lab. NVML provides a transactional object store, supporting transactions, memory management and general facilities for low-level persistent memory operations.

We emulate both STT-RAM (on-chip) and PCM (off-chip) with DRAM. We reserve 1GB of DRAM to emulate persistent memory by using *memmap* kernel parameter to manually create an e820 type 12 memory region. To account for off-chip persistent memory's slower writes relative to on-chip non-volatile memory, we add delays on operations that go to DRAM for the additional latency of persistent memory. Similar to Mnemosyne [25], we implement the delay with a loop that reads the processor's timestamp counter. In addition, we use non-temporal store instructions to bypass the CPU caches when accessing STT-RAM. In order to evaluate the effects of adaptive caching mode on performance, we establish a mapping of the architecture specific memory type via ioremap functions (e.g., *ioremap_wc()*) when mapping physical persistent memory to kernel address space.

Benchmark. We use micro-benchmarks listed in Table 1 to evaluate the performance. These micro-benchmarks implement data structures that provide transactional object store for persistent memory with respect to any power-fail interruptions. By using atomic transactions, these data structures can get away without any recovery process since every memory transaction is either done in all or nothing.

Table 1: Micro-benchmarks used in experiments.

Workloads	Description
CTree	insert/delete nodes in a crit-bit tree
BTree	insert/delete nodes in a b-tree
RBTree	insert/delete nodes in a red-black tree
Hashmap	insert/delete entries in a hashmap

6.2 On-Chip Logging

First, we measure the performance difference between on-chip logging and off-chip logging. The only difference between them lies in log area. One employs write-optimized STT-RAM, while the other uses PCM. As shown in Fig. 7, compared with off-chip logging, on-chip logging can yield 2.48×, 2.09×, 2.03×, and 2.47× improvement in throughput (operations/second) for CTree, BTree, RBTree, and Hashmap, respectively.

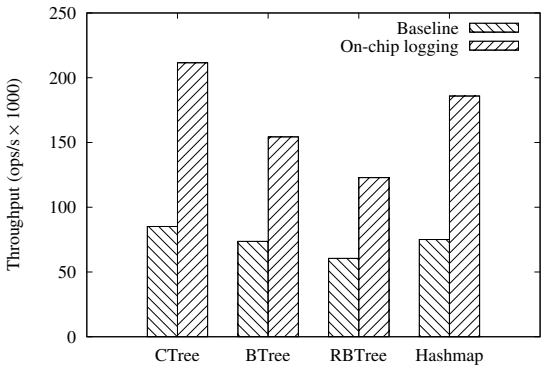

Figure 7: Performance comparison between on-chip logging and off-chip logging.

To evaluate the impact of log space size on the performance, we change the size of STT-RAM ranging from 16MB to 128MB. Fig. 8 shows that, as the size of STT-RAM increases from 16MB to 128MB, the throughput also gradually improves. After the size exceeds 64MB, the throughput improvement becomes insignificant. Theoretically speaking, the throughput should gradually be improved as the increasing of STT-RAM size. However, due to the limited number of transactions and the constraint of chip area, on-chip memory space is impossible to be very large. In addition, in order to enforce persist ordering constraints, the system needs to do checkpoint in time by flushing their cachelines to persistent memory. After checkpoint, the log space occupied by committed transactions may be released, thus much more log space has no contribution to throughput improvement. By trading off between performance and chip area, 64MB is a relatively suitable size for on-chip STT-RAM.

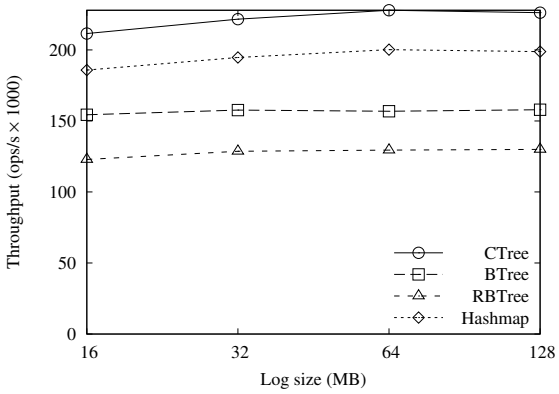

Figure 8: Performance comparison under different log space size.

6.3 Adaptive Caching Mode

We first evaluate the effects of different caching modes on the throughput. Because PMFS does not support WT mode at present, we only evaluate the performance of four micro-benchmarks under other three caching modes including UC, WC, and WB. Of these three caching modes, WB mode still requires extra sync operations for TM systems. The evaluation results are shown in Fig. 9. We can conclude that WB with sync is the worst in performance for all the four benchmarks. It indicates that sync operation has

significant negative impact on performance. It is well-known that WB without sync provides better performance in non-transactional systems. However, for transactional systems, it is compulsory to use sync operations for ensuring ordering constraints and data durability. The results also show that CTree and BTree perform the best in UC mode, while RB-Tree and Hashmap perform a little better in WC mode than in UC mode. This indicates that the performance depends on many factors including the ratio of reads to writes, the number of cacheline flushes, and so on.

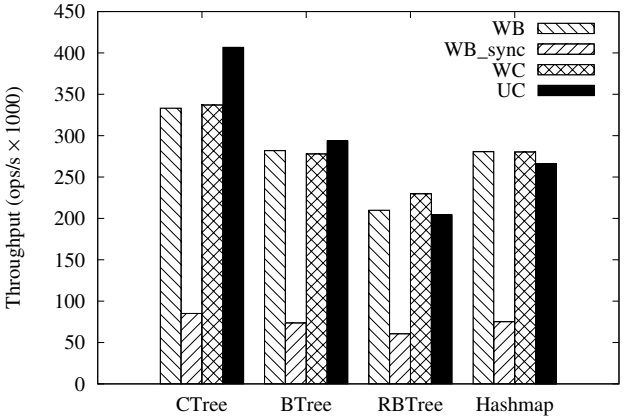

Figure 9: Performance comparison under different caching modes.

According to experimental results, we form four mixed workloads by combining four micro-benchmarks mentioned above. We evaluate the performance under WB with sync, WC, UC, and adaptive caching mode (called adaptive sync in Fig. 10). As shown in Fig. 10, it indicates that adaptive sync can provide better performance than other single caching modes. Compared with WB mode with sync, adaptive sync can yield 4.10× on average performance improvement. Adaptive sync can achieve 3.5% on average performance improvement compared to WC and UC mode. This indicates that WB mode is not the suitable choice in transactional systems. Although the advantage of adaptive sync is not obvious compared with UC or WC, it only indicates that the four benchmarks do not have enough write-asymmetry [7]. In practice, even if all the operations are symmetric, adaptive caching mode mechanism would not worsen the performance.

6.4 Overall Performance

Combined with the two proposed techniques, we evaluate the overall performance. As shown in Fig. 11, the overall performance of our design is 6.77×, 6.46×, 5.88×, 6.37× than that of conventional transactional persistent memory for four mixed workloads, respectively.

7. RELATED WORK

There have been many proposals for reducing consistency overhead at the hardware or software level [9, 18, 19, 21, 23, 26, 29]. Condit *et al.* [9] adopted an epoch barrier mechanism to order memory writes to minimize the flush traffic. Wang *et al.* [26] advocated to leverage existing hardware support for write combining to protect committed work upon a failure. Lu *et al.* [19] introduced LOC, which satisfies

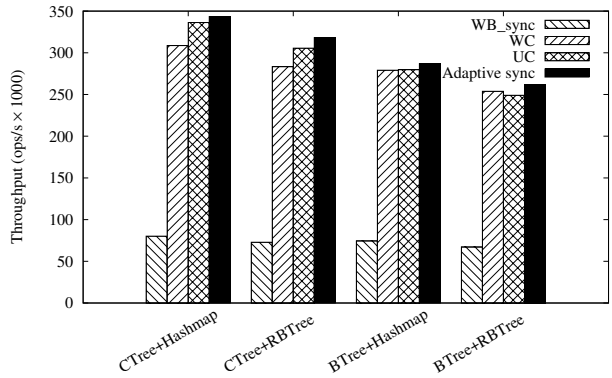

Figure 10: Performance comparison between single caching mode and adaptive caching mode.

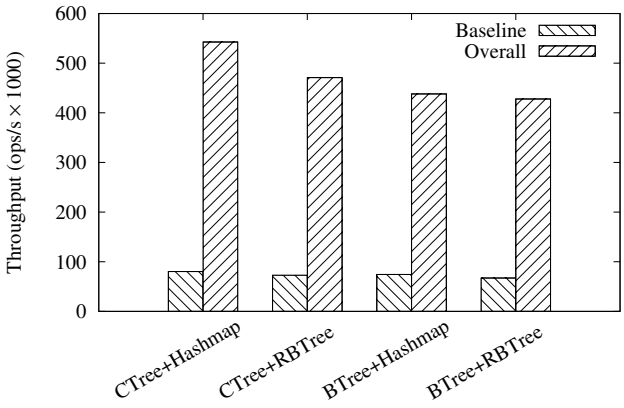

Figure 11: Overall performance for four mixed workloads.

the ordering constraints of persistent memory writes at lower performance degradation. Pelley *et al.* [21] introduced relaxed memory persistency model to improve persist concurrency. Yang *et al.* [29] presented to reduce consistency cost by only enforcing consistency on critical data. Lu *et al.* [18] proposed to blur the volatility-persistence boundary to achieve fast transaction processing. Sun *et al.* [23] studied how persistent memory write latency affects system throughput and then proposed differential persistency and dual persistency.

Previous research employed a single caching mode for all the data. PMFS [10] employs WB mode for the whole data. In our work, we adopt adaptive caching mode in terms of memory access patterns. Besides, recent research leveraged non-volatile cache to ensure crash consistency [27, 30]. Zhao *et al.* [30] proposed to store the updated data in non-volatile cache and then write back to persistent memory once relevant transaction is committed, thereby ensuring consistency without using copy-on-write or write-ahead logging. Wang *et al.* [27] implemented transactional persistent memory by modifying cache architecture. Our work is closely related to Kumud [7], however, the latter focused on implications of caching mode on persistent memory programming.

8. CONCLUSIONS

Sync is imperative for ensuring ordering constraints and making data durable in persistent memory. However, sync overhead induced by cachelines flushing has become the per-

formance bottleneck of single-level store systems. To mitigate this issue, we propose two techniques. First, we use individual on-chip non-volatile memory to store log data, thereby eliminating sync overhead incurred by logging. Second, we present an adaptive caching mode policy in terms of data access patterns to eliminate unnecessary sync overhead. The evaluation results indicate that the two techniques help improve the overall performance significantly.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments. This work is supported by National Natural Science Foundation of China under Grant No.61502321 and State Key Laboratory of Computer Architecture under Grant No.CARCH201503.

10. REFERENCES

[1] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown. Recon: Verifying file system consistency at runtime. *TOS*, 8(4):15, 2012.

[2] T. S. Pillai et al. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *OSDI*, pages 433–448, 2014.

[3] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of linux file system evolution. *TOS*, 10(1):3, 2014.

[4] S. Raoux, G. W. Burr, M. J. Breitwisch, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, pages 465–479, 2008.

[5] T. Kawahara. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Design and Test of Computers*, 28(1):52–63, 2010.

[6] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Hardware/software codesign*, pages 73–78, 2002.

[7] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Implications of CPU caching on byte-addressable non-volatile memory programming,Technical Report HPL-2012-236, HP Lab, 2012.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, pages 105–118, 2011.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, pages 133–146, 2009.

[10] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, pages 1–15, 2014.

[11] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *ATC*, pages 319–331, 2012.

[12] R. Hagmann. *Reimplementing the Cedar file system using logging and group commit*, In *SOSP*, pages 155–162, 1987.

[13] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures*, In *ISCA*, pages 289–300, 1993.

[14] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O stack optimization for smartphones. In *ATC*, pages 309–320, 2013.

[15] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. In *MICRO*, pages 660–671. ACM, 2015.

[16] W.-H. Kim, B. Nam, D. Park, and Y. Won. Resolving journaling of journal anomaly in Android I/O: multi-version B-Tree with lazy split. In *FAST*, pages 273–285, 2014.

[17] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang. NVM Duet: Unified Working Memory and Persistent Store Architecture. In *ASPLOS*, pages 455–470, 2014.

[18] Y. Lu, J. Shu, and L. Sun. Blurred persistence in transactional persistent memory. In *MSST*, pages 1–13, 2015.

[19] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *ICCD*, pages 216–223, 2014.

[20] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu. A case for efficient hardware/software cooperative management of storage and memory. In *WEED*, pages 1–7, 2013.

[21] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA*, pages 265–276, 2014.

[22] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *TOCS*, 10(1):26–52, 1992.

[23] L. Sun, Y. Lu, and J. Shu. DP2: reducing transaction overhead with differential and dual persistency in persistent memory. In *CF*, pages 1–8, 2015.

[24] S. Swanson and A. M. Caulfield. Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage. *Computer*, pages 52–59, 2013.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, pages 91–104, 2011.

[26] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *VLDB*, 7(10):865–876, 2014.

[27] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen. Persistent transactional memory. *CAL*, 14(1):58–61, 2015.

[28] W. Xu, H. Sun, X. Wang, Y. Chen, and T. Zhang. Design of last-level on-chip cache using spin-torque transfer ram (STT RAM). *, IEEE Transactions on VLSI Systems*, 19(3):483–493, 2011.

[29] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: reducing consistency cost for NVM-based single level systems. In *FAST*, pages 167–181, 2015.

[30] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, pages 421–432, 2013.