Xu YC, Wan H, Qiu KN *et al.* Reducing synchronization cost for single-level store in mobile systems. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(4): 836–848 July 2016. DOI 10.1007/s11390-016-1666-y

Reducing Synchronization Cost for Single-Level Store in Mobile Systems

Yuan-Chao Xu [1,2], *Member, CCF, ACM*, Hu Wan [1], Ke-Ni Qiu [1], *Member, CCF, ACM*
Tao Li [3], *Member, ACM, IEEE*, and Wei-Gong Zhang [1], *Senior Member, CCF*

[1] *College of Information Engineering, Capital Normal University, Beijing 100048, China*

[2] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[3] *Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, U.S.A.*

E-mail: {xuyuanchao, wanhu, qiukn}@cnu.edu.cn; taoli@ece.ufl.edu; zwg771@cnu.edu.cn

Received June 1, 2015; revised April 7, 2016.

Abstract Emerging byte-addressable non-volatile memory technologies, such as phase change memory (PCM) and spin-transfer torque RAM (STT-RAM), offer both the byte-addressability of memory and the durability of storage, thus making it feasible to build single-level store systems. To ensure the consistency of persistent data structures in the presence of power failures or system crashes, it requires flushing cache lines to persistent memory frequently, thus incurring non-trivial synchronization overhead. To mitigate this issue, we propose two techniques. First, we use non-volatile STT-RAM as scratchpad memory on chip to store recovery information, thereby eliminating synchronization cost in the logging phase due to the avoidance of off-chip logging operations. Second, we present an adaptive synchronization policy based on caching modes in terms of data access patterns, thereby eliminating unnecessary synchronization cost in the checkpoint phase. Evaluation results indicate that the two techniques improve the overall performance from 2.15x to 2.39x compared with conventional transactional persistent memory.

Keywords crash consistency, synchronization cost, persistent memory, power failure, mobile system

1 Introduction

In modern mobile devices, SQLite is a widely used server-less database to store the structured data, as well as metadata related to unstructured data (e.g., photos). Of all writes in smartphones, about 90% of the write requests are to the SQLite database and journal. Besides, 70% are synchronous[1]. These synchronous operations are mainly used to enforce ordering constraints of data persist operations, thereby ensuring persistent data consistency upon system crashes[2-4] or power failures[5-6]. This is an issue known as crash-consistency problem[7]. From users' perspective, it is expected to complete a particular operation (e.g., withdraw) atomically. However, almost all the user operations cannot be done through a single I/O operation, since the granularity of a single I/O operation is usually less than that of a single user operation. If a failure occurs during the period of two I/O operations, the user data will be left in an inconsistent state. Therefore, some measures should be taken to avoid this situation.

In recent years, byte-addressable non-volatile memory technologies (also called persistent memory, PM), including phase change memory (PCM)[8], spin-transfer torque RAM (STT-RAM)[9], and memristor[10], promise the fusion of volatile memory and persistent storage[11]. This allows system architects to use entire PM to replace both working memory (e.g., DRAM) and persistent storage (e.g., eMMC) in the traditional two-level storage model, in order to build a single-level store

Regular Paper

This work was supported by the National Natural Science Foundation of China under Grant Nos. 61502321, 61472260, and 61402302, the Beijing Natural Science Foundation under Grant No. 4143060, the Overseas Visiting Scholar Program of Beijing under Grant No. 067135300100, the State Key Laboratory of Computer Architecture of China under Grant No. CARCH201503, and the Beijing Innovative Teams and Teacher Career Development Program under Grant No. IDHT20150507.

mobile system. Single-level store systems avoid the data movement between fast memory and slow storage, thus reducing both I/O latency and energy significantly compared with two-level storage systems[12]. However, it imposes a great challenge on system software, including programming model and operating system[12-14] because traditional block-oriented software stack cannot efficiently exploit the byte-addressability of PM. Instead of issuing reads and writes on files, applications can access PM directly via load/store memory interface. Thus, transactional memory[15] becomes a good programming model in ensuring data consistency via transaction mechanism for volatile memory and persistent memory. For example, Tiny-STM[16], a lightweight software transactional memory, implements write-ahead logging in order to roll back to previous consistent state when a transaction aborts. In this paper, we focus on transactional persistent memory[17-18], which can enable the correct recovery of data after a system failure. To achieve this, in a transaction, it requires two writes to PM for each update[19]: one to store recovery information in log area, and the other to write the data itself in data area. Besides, to ensure data durable, it still requires to issue a sync operation (i.e., *clflush* and *sfence* instructions) following each normal store instruction in order to enforce flushing dirty cache lines out of the cache hierarchy. As a result, large amount of sync results in dramatic performance overhead[20-21].

Our goal is to reduce the sync cost incurred by crash consistency guarantees. In this work, we propose two techniques. The first is on-chip logging based on scratchpad memory (SPM)[22]. The key idea is to eliminate sync overhead incurred by cache line flushes through leveraging non-volatile software-managed on-chip memory to store log information. The second is adaptive sync based on caching modes. The key idea is to reduce the number of sync through choosing a reasonable caching mode in terms of data access pattern.

Our contributions are summarized as follows.

• We observe two performance bottlenecks in the logging phase and the checkpoint phase within a transaction to enable crash consistency.

• We use non-volatile STT-RAM as scratchpad memory to store recovery information on chip, thereby eliminating the cache line flushes to off-chip PM.

• We present an adaptive sync policy based on caching modes in terms of data access pattern to eliminate unnecessary cache line flushes to PM.

• We implement and evaluate the two techniques. The evaluation results indicate that they help improve the overall performance up to 2.2x on average compared with conventional transactional persistent memory.

2 Background

In this section, we target single-level store in mobile systems, and discuss several problems related to crash consistency, including transactional memory and synchronization overheads.

2.1 Single-Level Store System

PM provides byte-addressable, low-latency features close to DRAM, while retaining non-volatile property similar to eMMC. Thus, it is feasible to build a single-level store system with entire PM. However, since PM is more sensitive to software overhead than eMMC, software stack should be optimized or redesigned so as to exploit the byte-addressability of PM via load/store memory interface as shown in Fig.1.

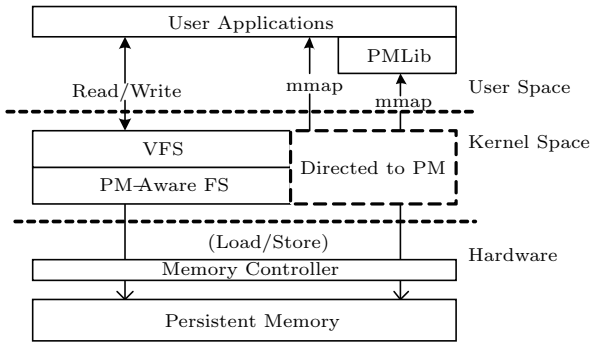

Fig.1. Single-level store system architecture using persistent memory.

Crash consistency needs to ensure the atomicity, consistency, and durability of data updates. There are two typical consistency guarantee techniques including write-ahead logging and copy-on-write. Regardless of what technique is adopted in single-level store systems, the primary consistency overhead comes from frequent cache line flushes to off-chip PM.

2.2 Transactional Memory

Transactional memory is considered as an ideal way to enable data consistency upon a failure for persistent memory. However, traditional transactional memory only ensures the atomicity, isolation and consistency of a series of memory operations (ACI). To ensure that the data within a transaction can persist permanently

once committed, it requires to add durability (D) to transactional memory, called transactional persistent memory[17-18], which is identical with database transactions. To achieve the durability, aside from non-volatile storage media, ordering constraints should also be enforced to enable correct recovery after a failure by issuing sync operations at the end of each transaction. Transactional memory can be implemented at the hardware or software level. Hardware transactional memory may comprise modifications in processors, cache and bus protocol to support transactions. Software transactional memory[23-24] can provide transactional memory semantics in a software runtime library or programming language, thus requiring minimal hardware support with a performance penalty.

In short, transactional persistent memory is a good alternative to ensure crash consistency, and can achieve efficient transaction concurrent control for programmers with ease.

2.3 Synchronization Overheads

We assume that there are only volatile cache and persistent memory in our target system. In this scenario, in order to make the data updates durable, we need to flush the dirty cache lines into persistent memory by issuing sync operations for at least two reasons. First, when a transaction is committed, it requires to issue a sync operation to ensure that all the log entries within a transaction have been persisted before checkpointing. In this case, sync is used to enforce persist ordering constraints for correct recovery. Second, during the checkpoint phase, sync is used to ensure that all the data in the original locations are in a consistent state and reach the newest version. Memory consistency model only ensures that data are visible to all the cores in the multicore systems. In other words, these data may stay at the last level cache; hence they do not reach persistent memory yet, and thus will be lost in case of power failure or system crash. As a result, frequent sync operations lead to significant performance cost. We measure the performance of fs_mark benchmark[25] with sync enabled or disabled. If sync is enabled, the performance degrades about 9x on average as shown in Fig.2. This indicates that sync has a large impact on performance.

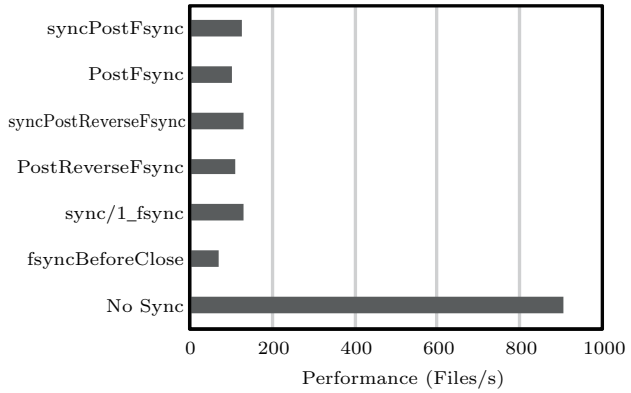

Fig.2. Example of sync overhead (FS-Mark, 1 000 files, 1 MB size, with sync or without sync).

3 Motivation

We observe that sync cost in our target system can be partially eliminated or reduced.

3.1 Sync Cost Within a Transaction

In traditional two-level storage model, the sync cost derives from two phases within a transaction as illustrated in Fig.3. The first phase is to flush dirty cache lines to DRAM. The second phase is to write modified dirty pages back to eMMC. Due to the high-latency of persistent storage, the second phase is the main source of sync cost in the two-level storage model systems. In contrast, for single-level store systems, data movement between fast memory and slow storage is eliminated. Hence, flushing cache lines out of the cache hierarchy turns out to be the performance bottleneck.

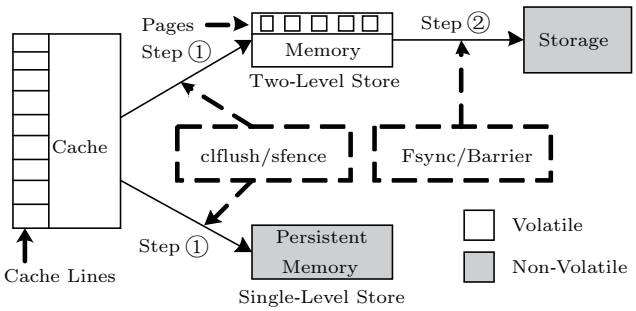

Fig.3. Sync path comparison between two-level store model and single-level store model.

Transactional persistent memory usually uses write-ahead logging to enable recovery. As illustrated in Fig.4[18], logically, data are first read out from the data area to the CPU cache (step ①), and then are copied to the execution area (step ②) in case the generated

new-version data may be evicted, thereby overwriting the old-version data in the data area. This violates the atomicity of transactions. When the transaction is committing, the executed data are first copied to the log area (step ③) for logging persistence (step ④). After the log has been persisted, the committed data are then checkpointed to the data area (step ⑤) for checkpoint persistence (step ⑥). Therefore, each data update has been written into persistent memory twice, thereby aggravating the sync cost.

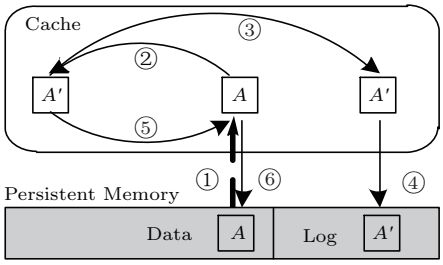

Fig.4. Transaction phases in traditional transactional persistent memory.

3.2 Sync Cost in Logging Phase

For each transaction, the logging phase must be completed before the checkpoint phase in order to ensure the atomicity of the data within a transaction. Each log entry, as the copy of real data, needs to be flushed out of cache to off-chip persistent memory (step ④ as shown in Fig.4) before the real data itself is updated. Therefore, logging operations happen in the critical path.

Theoretically speaking, log data may be stored in any non-volatile storage media only if the log data still retain after a failure. Once the real data are written to their home locations, the corresponding log data will become useless and may be discarded immediately. Thus, too large log space is not essential for performance improvement, thereby making it possible to place all the log data on chip. We thus propose on-chip logging technique to eliminate the sync operations in the logging phase. The details are discussed in Subsection 4.2.

3.3 Sync Cost in Checkpoint Phase

There are three primary approaches to mitigate sync overheads. First, it is to reduce the number of sync operations[18,20] through identifying unnecessary sync operations. Second, it is to reduce the overhead of a single sync operation[26-27] through skipping unmodified cache lines. Third, it is to reduce persistent ordering constraints[28-30] by leveraging novel instructions[31] such as *clflushopt*, *clwb*, and *pcommit*, or adopting relaxed memory persistency models such as epoch persistency[28] and strand persistency[29].

We observe that different data have different memory access patterns, and meanwhile, different caching modes have different sync cost[32]. From the perspective of write latency, WB mode can gain better performance than WT mode. However, from the perspective of data persistency, since WB mode needs extra sync operations to make data durable whereas WT mode does not, WB mode is not the suitable caching mode in some cases, e.g., append-only writes. We find that for SQLite application, WB mode without sync has better performance than WT mode; however, WB mode with sync has worse performance than WT mode, as illustrated in Fig.5. Therefore, we propose an adaptive sync policy based on caching modes to reduce the number of sync operations.

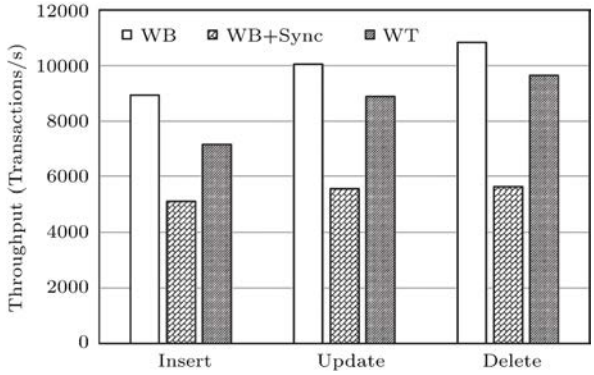

Fig.5. Performance comparison among WB mode without sync, WB mode with sync, and WT mode.

4 Design Overview

In this section, we discuss how to leverage on-chip logging and adaptive sync to reduce sync cost. In a transaction, logging and checkpoint are the two most important phases. Fig.6 shows the comparison of these two phases in a single-level store system between traditional transactional persistent memory and our proposed design scheme.

4.1 Software for Single-Level Store

As traditional storage software stack is designed for slow-speed block devices, it is inefficient to access low-latency byte-addressable non-volatile memory. Existing operating systems should be optimized, or even redesigned for persistent memory[13,33] for at least three

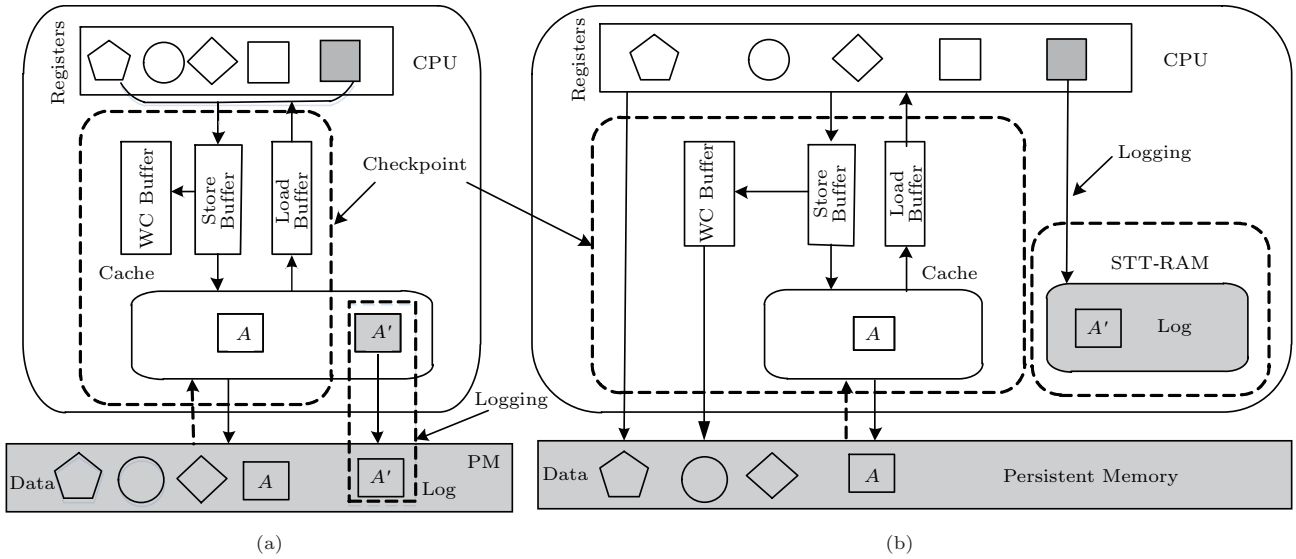

Fig.6. Comparison of the logging phase and the checkpoint phase between (a) traditional transactional persistent memory, and (b) our proposed design with on-chip logging and adaptive sync.

reasons. First, traditional memory management assumes that working memory is volatile and has unlimited endurance. Second, with respect to single-level store systems, paging fault does not occur any more. Third, user applications may access persistent memory directly via load/store memory interface. In our work, we still add two extra modules into the operating system to manage SPM and implement the adaptive sync policy. Besides, we add a small-sized non-volatile STT-RAM as SPM to place log data on chip as shown in Fig.7.

4.2 On-Chip Logging Using SPM

On-chip logging using SPM is used to eliminate sync cost in the logging phase through placing the copy of real data (i.e., log) in the non-volatile, software-managed on-chip memory. Since on-chip logging approach makes the log data durable in non-volatile memory on-chip rather than in off-chip, it significantly reduces access latency and eliminates sync cost incurred by flushing cache lines out to off-chip memory. Although previous work[1] also mentions to use external device for journaling, this device is off chip rather than on chip. In addition, since log data are read-only (when recovery is needed) and append-only, there is no point leveraging CPU cache for data locality. Moreover, it may result in cache pollution.

Therefore, we employ a special non-volatile memory[22] instead of non-volatile cache to store the log entries of both user data and metadata. Fig.8 shows the logging phase within transactional persistent memory using on-chip logging based on scratchpad memory. With this technique, a transaction writes its log data directly to the on-chip log area rather than the off-chip log area. The implementation details will be discussed in Subsection 5.1.

4.3 Adaptive Sync Based on Caching Modes

Adaptive sync based on caching modes is used to reduce sync cost in the checkpoint phase through choosing suitable caching modes for different data. The idea is based on the fact that not all data has the same locality and reliability requirement. Meanwhile, most processors allow any area of system memory to be cached in the L1, L2, and L3 caches (if any). In addition, they also allow the caching mode to be specified in individual pages or regions of system memory. In practice, each caching mode has different sync overheads. Unfortunately, almost all the existing systems adopt a single caching mode for all data, e.g., WB mode by default in PMFS[33]. As a result, it is nearly impossible to meet various locality and reliability demands for different data.

Therefore, we propose an adaptive sync policy, as shown in Fig.9. This technique dynamically determines caching modes via the trade-off between data locality and data reliability. The details will be discussed in Subsection 5.2.

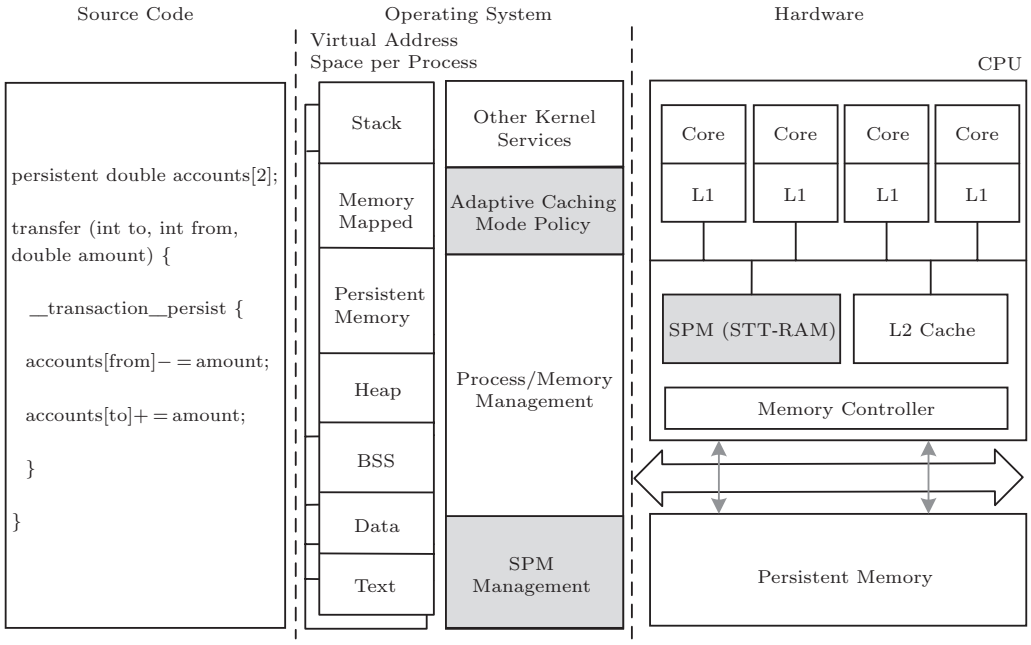

Fig.7. Architecture overview for a single-level store system.

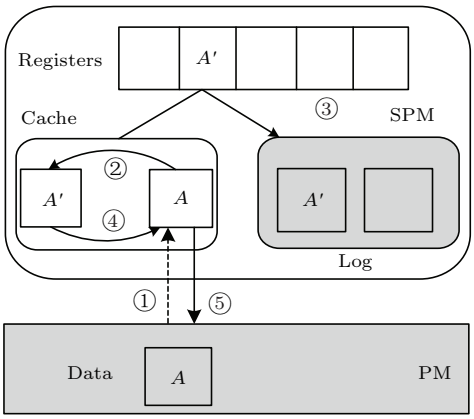

Fig.8. On-chip logging using scratchpad memory.

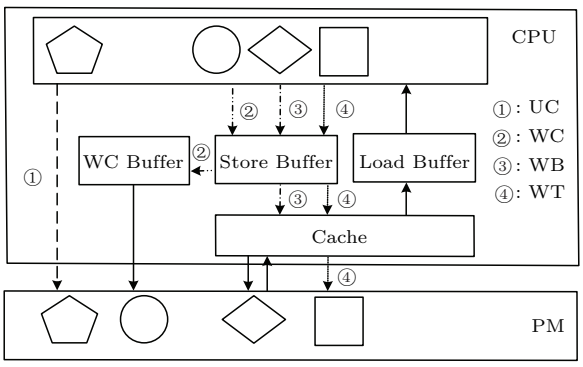

Fig.9. Adaptive sync based on caching modes.

4.4 Comparison of Sync Cost

Fig.10(a) shows the sequence of conventional transactional persistent memory using off-chip logging for a single-level store system. Since store instructions only ensure the data arriving at the last level cache according to memory consistency model, it is required to issue *clflush* and *sfence* instructions to make data durable in non-volatile memory. Moreover, it also requires using sync operations to enforce all the logs persistently before the system can in-place update the real data in their home locations. Consequently, it needs at least two sync operations on average for each data update in order to ensure crash consistency.

Fig.10(b) shows the timeline of our proposed design using on-chip logging and adaptive sync. Since log area is located on chip, it does not require cache line flushes and memory barrier operations, thereby eliminating the sync cost in the logging phase. In the checkpoint phase, we use adaptive sync based on various caching modes instead of a single caching mode. Since some caching modes, e.g., WT mode and UC mode, do not need sync operations, they reduce unnecessary sync operations, thereby mitigating the sync cost in checkpoint phase. Through our proposed two techniques, the overall sync cost in transactional persistent memory system is significantly reduced.

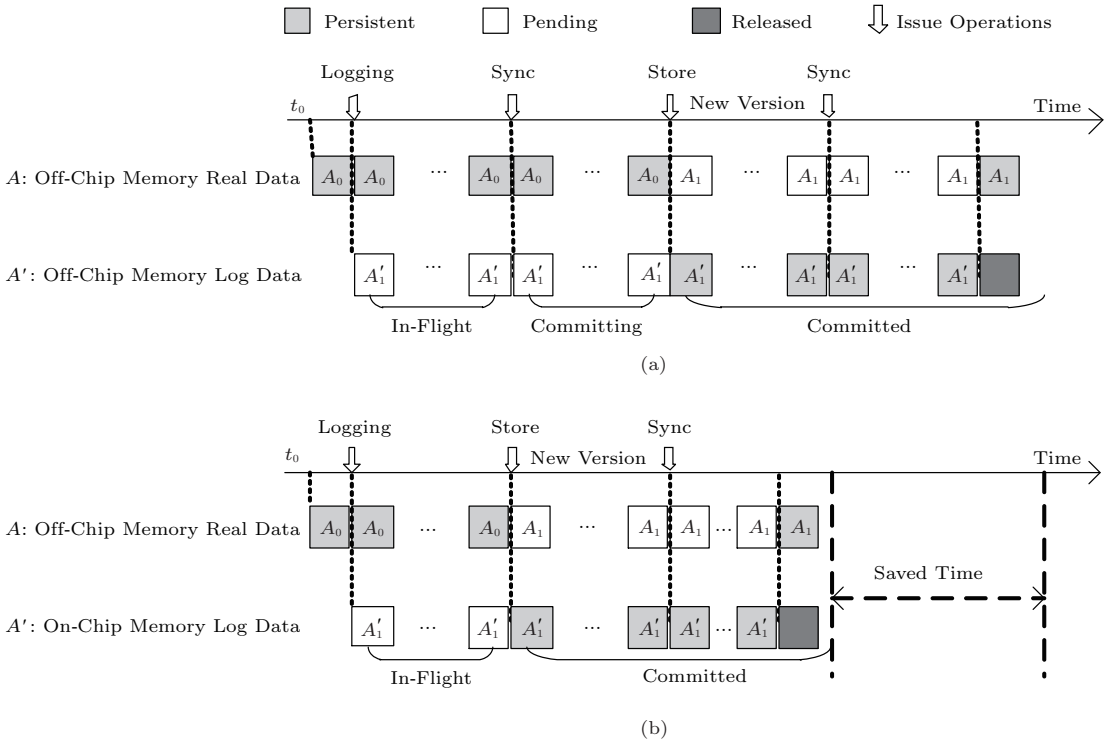

Fig.10. Comparison of the timeline. (a) Traditional transactional persistent memory design. (b) Our proposed design leveraging on-chip logging and adaptive sync. Block A represents the old version of data block. Block A' represents the new version being updated.

5 Implementation

In this section, we discuss the implementation details about the two aforementioned techniques.

5.1 On-Chip Logging

We use non-volatile STT-RAM as SPM to store log entries on chip.

5.1.1 Scratchpad Memory

Prior work[17,21] leverages non-volatile cache to enable crash consistency. The challenge is how to modify cache architecture. In contrast, we explore scratchpad memory instead of non-volatile cache to store log data at the expense of minimal hardware modification. SPM is also located on chip, which may use the same memory technology as cache. However, SPM outperforms cache in access latency and energy efficiency because SPM does not have tag array and relevant comparison logic circuit. Different from traditional SPM technique, we employ non-volatile memory instead of volatile memory to achieve the persistence of data. Because STT-RAM is non-volatile, and has similar endurance but higher density compared with SRAM[27], we employ write-optimized STT-RAM[34] as SPM to place log data. Log data are written directly into SPM without polluting cache space.

5.1.2 SPM Management

Unlike cache, SPM is managed by software. SPM is unified addressing with off-chip memory. As illustrated in Fig.11, the address space of off-chip PM ranges from 0 to $M-1$, while the address space of SPM ranges from M to N. In addition, operating system still needs to differentiate logging write operations from normal write operations. From users' perspective, there is no any difference between on-chip SPM and off-chip PM.

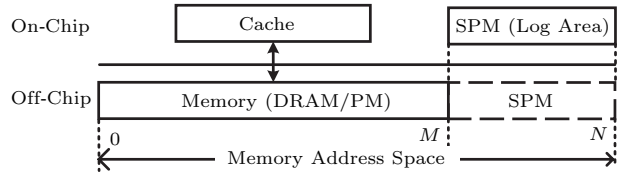

Fig.11. Memory address space partition between off-chip PM and on-chip SPM. M represents the end address of PM. N represents the end address of SPM.

The space size of SPM depends on the degree of lazy checkpoint. Once all the data of a transaction are checkpointed to their home locations successfully,

corresponding log space may be released immediately. Operating system can enforce doing checkpoint to the release part of log space. In addition, STT-RAM is about 4x higher in density than SRAM[34]. We evaluate sync overhead as the size of SPM varies in Subsection 6.2. The result shows that 64 MB is relatively suitable log space size in terms of performance per area. From the perspective of chip area, 64 MB for SPM is about equivalent to 16 MB for SRAM. It indicates that the chip area does not increase significantly.

SPM is only used to store log entries. Each log entry is cache line (e.g., 64 B) aligned, which includes transaction identifier, address, type, size, and data as illustrated in Fig.12. When a transaction is created, a 4 KB new page in SPM is allocated to store log entries for this transaction. One page can contain up to 46 log entries. If the number of log entries associated with this transaction exceeds the limit of 4 KB, another new page will be allocated to this transaction. Log entries from different transactions may be written to SPM concurrently, thus the address space may be discontinuous from a transaction's perspective. Similar to normal memory management in Linux kernel, several volatile data structures are also needed in order to manage SPM. For instance, it needs to record the pointer of the last log entry for each committing transaction.

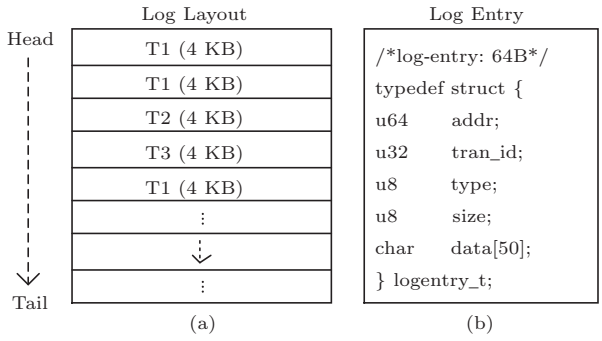

Fig.12. Typical layout of transaction log on scratchpad memory.

5.1.3 Data Recovery

A transaction includes three states: uncommitted, committed, and checkpointed. If a transaction is committed successfully, the type field of the last log entry is marked as *committed_t*, or else marked as *committing_t*. After a transaction is checkpointed, its state will be transferred into checkpointed. Ideally, 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 both cache and SPM, checkpoint should be done timely in order to release the used space.

After system reboot, recovery module in operating system will scan the SPM from head to tail at the page-level granularity. Uncommitted transactions will be skipped. For committed transactions, the data in each log entry are read out first and then used to update the corresponding address of PM in place. When all the committed transactions are processed completely, the whole log space will be released. If checkpointed transactions do not release their occupied log space in time due to sudden power loss or system crash, the recovery process will also be restarted after system boot to ensure the completeness of checkpoint processing.

5.2 Adaptive Sync Based on Caching Modes

We adopt different caching modes at the page-level granularity in terms of various data access patterns.

5.2.1 Caching Modes

Taking Intel processors as an example, they support six caching modes[35], such as strong uncacheable (UC), uncacheable (UC-), write combining (WC), write through (WT), write back (WB), and write protected (WP). These caching modes are categorized into two types, cacheable and uncacheable. For instance, WT mode and WB mode belong to the cacheable type. UC mode and WC mode belong to the uncacheable type. WB memory can be cached and written back whenever it is forced. WT allows memory to be cached, but writes have to be written straightly to memory, i.e., memory is kept up to date with the 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 guarantees overheads.

5.2.2 Data Diversity

Likewise, various data have different consistency requirements and locality characteristics. For example, some metadata, such as *inode* in ex4 file system, has been updated very frequently, whereas user data may be written into persistent memory only once. It is closely related to data reuse distance at the page-level granularity. Bhandari and Chakrabarti[32] also found that WB caching mode is not always optimal for all the applications. Thus, we should choose the relatively suitable caching mode for different data by trading off between reliability and performance.

5.2.3 Adaptive Sync Policy

Identifying data access pattern is a challenge for adaptive sync based on caching modes. Our approach is as follows. When a new page is allocated, its default caching mode is specified as WB. Then, during the sampling phase, transaction system will accumulate the number of writes to each page. At the end of the sampling phase, transaction system computes the average value of the total writes (*avg_writes*). If the number of writes for that page is less than *avg_writes*, the caching mode of that page will be changed to WT caching mode. After the sampling phase, that page keeps WT caching mode regardless of transactions. For simplicity, we consider only two caching modes and assume stable memory access behavior. We plan to explore a more complex adaptive policy in future work.

5.2.4 Hardware/Software Support

For Intel X86 architecture, we can specify caching modes at the page-level granularity through modifying PAT, PCD, and PWT in the page table entry[32]. We implement multiple versions of sync operations by using three instructions, including flushing cache lines (*clflush*), waiting until completion (*sfence*) and draining write combining buffer (*pcommit*). For WT mode, it is not required to issue *clflush* and *sfence* instructions. For WC mode, only *pcommit* instruction is needed for all the platforms except for Intel CPU with Asynchronous Dram Refresh.

6 Evaluation

In this section, we first evaluate the effects of on-chip logging and adaptive sync, separately. Then, we evaluate the overall performance. The baseline system is conventional transactional persistent memory with a single WB caching mode as shown in Fig.6(a).

6.1 Experimental Setup

Platform. We carry out the experiments on Exynos 5420 arndale octa board equipped with 4 GB eMMC and 2 GB DRAM. Because non-volatile memory devices are not yet commercially available, we emulate both SPM and PM with DRAM by using *memmap* kernel parameter to manually create a memory region with e820 type 12. To account for slower write to off-chip PM than to on-chip SPM, we add delays on operations that go to DRAM for the additional latency of off-chip PM. Similar to Mnemosyne[23], we implement the delay with a loop that reads the processor's timestamp counter. In addition, we use non-temporal store instructions to access SPM region in order to bypass the CPU cache. In order to evaluate the effects of adaptive sync based on caching modes, we establish a mapping of the architecture-specific memory type via ioremap functions (e.g., *ioremap_wc*()).

Benchmark. Mobibench[36] is an I/O workload generator designed for mobile systems. It can generate SQLite workload. We perform each of insert, update and delete operations 10 000 times and measure the throughput. We evaluate two SQLite journal modes including write-ahead logging (redo log) and rollback logging (undo log).

6.2 On-Chip Logging

First, we measure the performance difference between on-chip logging using SPM and off-chip logging using PM (baseline). Except for log area, all configuration parameters are the same for the two logging methods. We first evaluate the performance under SQLite journal mode with rollback logging. Fig.13 illustrates that compared with off-chip logging, on-chip logging can achieve up to 92.07%, 93.37%, 100.46% throughput improvement on average for insert, update, and delete, respectively. We then evaluate the performance under SQLite journal mode with write-ahead logging. Fig.14 illustrates that compared with off-chip logging, on-chip logging can achieve up to 96.45%, 99.98%, 109.98% throughput improvement on average for insert, update, and delete, respectively. SQLite write-ahead logging mode is a little faster in throughput than SQLite rollback logging mode. Compared with off-chip logging, the speedup of on-chip logging is very close for the two SQLite journal modes.

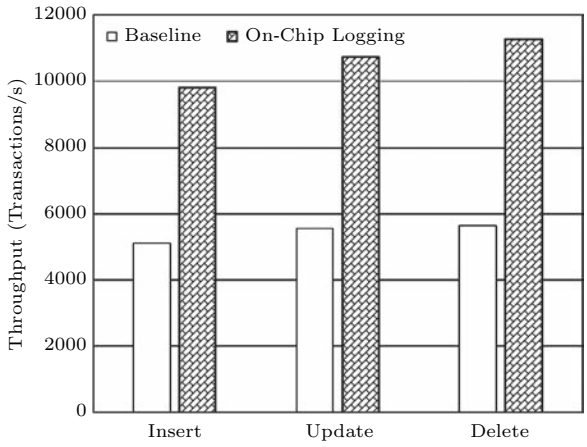

Fig.13. Throughput comparison between on-chip logging and off-chip logging under SQLite rollback logging mode.

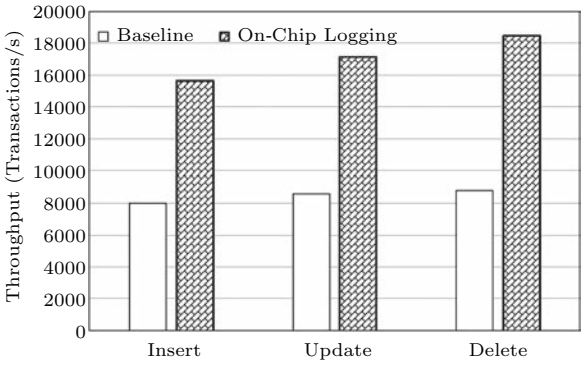

Fig.14. Throughput comparison between on-chip logging and off-chip logging under SQLite write-ahead logging mode.

To evaluate the impact of SPM size on the throughput, we vary the SPM size ranging from 16 MB to 128 MB. Both Fig.15 and Fig.16 show that for both rollback logging and write-ahead logging, as the space size of SPM increases, the throughput (transactions/second) gradually improves as well from the holistic trend. However, after the size exceeds 64 MB, the improvement becomes insignificant. Ideally, throughput should gradually be improved with the increase of the size of SPM. However, on-chip memory is impossible too large due to the area and power constraints. Additionally, due to checkpointing periodically, the log space occupied by committed transactions will be released in time, and thus too large log space has no contribution to throughput. Thus, 64 MB is a relatively reasonable size for SPM in terms of performance per area.

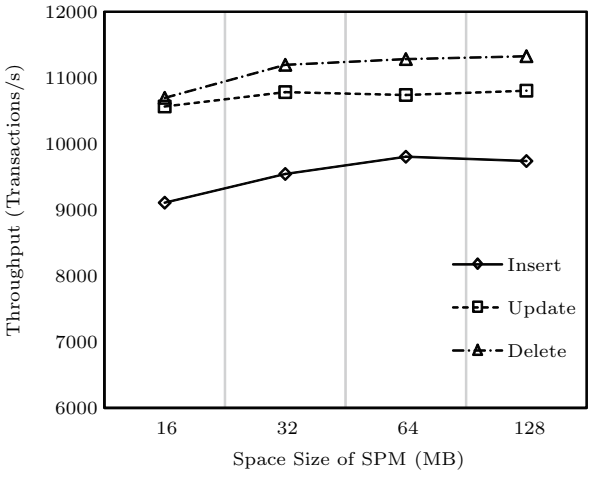

Fig.15. Throughput sensitivity to the space size of SPM under SQLite rollback logging mode.

6.3 Adaptive Sync Based on Caching Modes

We evaluate the effects of adaptive sync based on caching modes on throughput. First, we run SQLite benchmark for five times under three configurations including using WB mode with sync (baseline), using only WT mode, and using our proposed adaptive sync. Then we compute the average value of three operations including insert, update, and delete, respectively. The results are shown in Fig.17 for SQLite rollback logging and in Fig.18 for write-ahead logging, respectively.

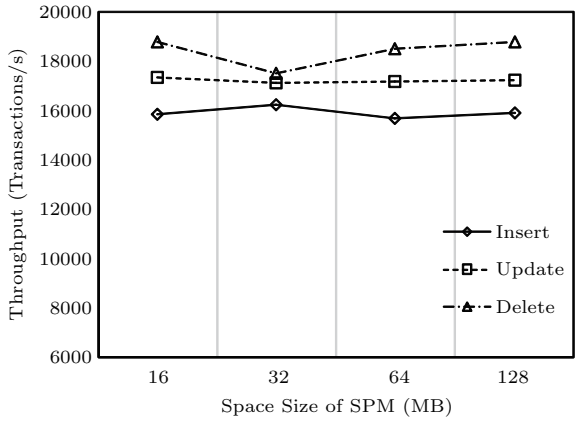

Fig.16. Throughput sensitivity to the space size of SPM under SQLite write-ahead logging mode.

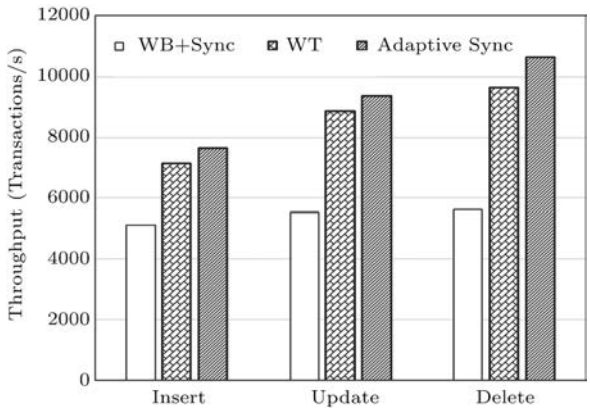

Fig.17. Throughput comparison among WB+sync, only WT caching mode, and adaptive sync under SQLite rollback logging mode.

With SQLite rollback logging, compared with baseline configuration (WB+sync), adaptive sync can achieve up to 49.77%, 68.74%, and 88.96% improvement in throughput on average for insert, update, and delete, respectively. Compared with using WT mode, adaptive sync can achieve up to 6.99%, 5.64%, and 10.38% improvement in throughput on average for insert, update, and delete, respectively. With SQLite write-ahead logging, compared with baseline configuration, adaptive sync can achieve up to 77.34%, 77.14%,

96.15% improvement on average in throughput for insert, update, and delete, respectively. Compared with using WT mode, adaptive sync can achieve up to 16.72%, 9.76%, and 1.93% improvement in throughput on average for insert, update, and delete, respectively. Theoretically speaking, adaptive sync can have more obvious effect on mixed workloads consisting of write-asymmetric programs running simultaneously.

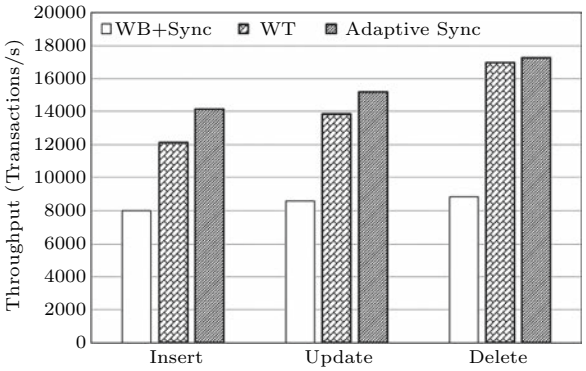

Fig.18. Throughput comparison among WB+sync, only WT caching mode, and adaptive sync under SQLite write-ahead logging mode.

6.4 Overall Performance

Combined with the two proposed techniques, we evaluate the overall performance under two SQLite journal modes. As shown in Fig.19, under SQLite rollback logging, the overall performance of our design is 2.15x, 2.14x, and 2.26x of that of conventional transactional persistent memory for insert, update, and delete, respectively. As shown in Fig.20, under SQLite write-ahead logging, the overall performance of our design is 2.20x, 2.24x, and 2.39x of that of conventional transactional persistent memory for insert, update, and delete, respectively.

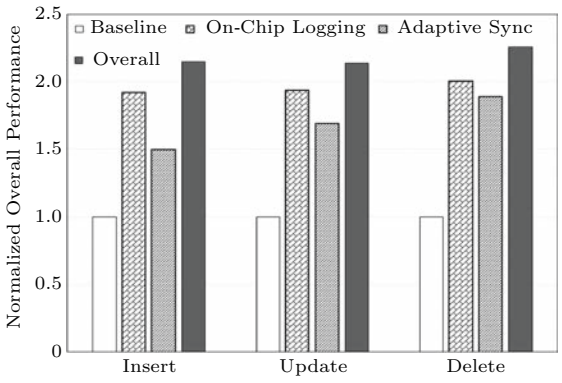

Fig.19. Overall performance under four configurations under SQLite rollback logging mode.

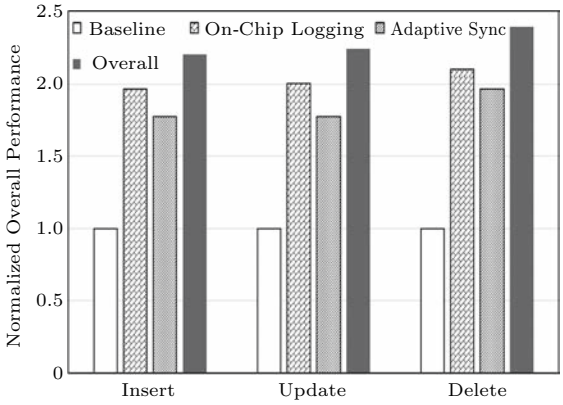

Fig.20. Overall performance under four configurations under SQLite write-ahead logging mode.

7 Related Work

There have been many proposals for reducing consistency overhead at the hardware or software level[18,20,28-30,37-38]. Condit *et al.*[28] adopted an epoch barrier mechanism to order memory writes to minimize the flush traffic. Wang and Johnson[37] advocated to leverage existing hardware support for write combining to protect committed work upon a failure. Lu *et al.*[30] introduced LOC, which satisfies the ordering constraints of persistent memory writes at lower performance degradation. Pelley *et al.*[29] introduced relaxed memory persistency model to improve persist concurrency. Yang *et al.*[20] 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 and Shu[38] studied how PM 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[33] employs a single WB mode for the whole memory. In our work, we adopt adaptive caching modes in terms of memory access patterns. Besides, recent research leveraged non-volatile cache to ensure crash consistency[17,21]. Wang *et al.*[17] implemented transactional persistent memory by modifying cache architecture. Zhao *et al.*[21] proposed to place the updated data in non-volatile cache and then write back to persistent memory whenever relevant transaction is committed, thereby ensuring consistency without using copy-on-write or write-ahead logging. In our work, it does not require to modify cache architecture.

8 Conclusions

Sync is imperative for enforcing ordering constraints and making data updates durable. However, sync cost incurred by flushing CPU cache lines to persistent memory has become the performance bottleneck for single-level store in mobile systems. To mitigate this issue, we proposed two techniques. First, we used STT-RAM as scratchpad memory to store log data on chip, thereby eliminating sync cost compared with off-chip logging. Second, we presented an adaptive sync based on caching modes in terms of data access patterns to eliminate unnecessary sync cost. The evaluation results indicated that the two techniques helped improve the overall performance. Although we targeted mobile systems, the proposed techniques are also applicable to servers in datacenters.

Acknowledgement We would like to thank Dr. You-You Lu at Tsinghua University, Beijing, and the anonymous reviewers for their helpful comments and suggestions.

References

[1] Jeong S, Lee K, Lee S, Son S, Won Y. I/O stack optimization for smartphones. In *Proc. the 2013 USENIX Annual Technical Conference*, Jun. 2013, pp.309-320.

[2] Fryer D, Sun K, Mahmood R, Cheng T, Benjamin S, Goel A, Brown A D. Recon: Verifying file system consistency at runtime. In *Proc. the 10th USENIX Conference on File and Storage Technologies*, Feb. 2012, Article No. 7.

[3] Pillai T S, Chidambaram V, Alagappan R, Al-Kiswany S, Arpaci-Dusseau A C, Arpaci-Dusseau R H. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proc. the 11th USENIX Conference on Operating Systems Design and Implementation*, Oct. 2014, pp.433-448.

[4] Lu L, Arpaci-Dusseau A C, Arpaci-Dusseau R H, Lu S. A study of linux file system evolution. In *Proc. the 11th USENIX Conference on File and Storage Technologies*, Feb. 2013, pp.31-44.

[5] Zheng M, Tucek J, Qin F, Lillibridge M. Understanding the robustness of SSDS under power fault. In *Proc. the 11th USENIX Conference on File and Storage Technologies*, Feb. 2013, pp.271-284.

[6] Narayanan D, Hodson O. Whole-system persistence. In *Proc. the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012, pp.401-410.

[7] Pillai T S, Chidambaram V, Alagappan R, Al-Kiswany S, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Crash consistency. *Communications of the ACM*, 2015, 58(10): 46-51.

[8] Raoux S, Burr G, Breitwisch M, Rettner C, Chen Y, Shelby R, Salinga M, Krebs D, Chen S H, Lung H, Lam C. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 2008, 52(4.5): 465-479.

[9] Hosomi M, Yamagishi H, Yamamoto T, Bessho K, Higo Y, Yamane K, Yamada H, Shoji M, Hachino H, Fukumoto C, Nagao H, Kano H. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. In *Proc. the International Electron Devices Meeting*, Dec. 2005, pp.459-462.

[10] Strukov D B, Snider G S, Stewart D R, Williams R S. The missing memristor found. *Nature*, 2008, 453(7191): 80-83.

[11] Liu R S, Shen D Y, Yang C L, Yu S C, Wang C Y M. NVM duet: Unified working memory and persistent store architecture. In *Proc. the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2014, pp.455-470.

[12] Meza J, Luo Y, Khan S, Zhao J, Xie Y, Mutlu O. A case for effcient hardware-software cooperative management of storage and memory. In *Proc. the 5th Workshop on Energy-Efficient Design*, Jun. 2013.

[13] Swanson S, Caulfield A. Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage. *Computer*, 2013, 46(8): 52-59.

[14] Xia F, Jiang D J, Xiong J, Sun N H. A survey of phase change memory systems. *Journal of Computer Science and Technology*, 2015, 30(1): 121-144.

[15] Herlihy M, Moss J E B. Transactional memory: Architectural support for lock-free data structures. In *Proc. the 20th Int. Symp. Computer Architecture*, May 1993, pp.289-300.

[16] Felber P, Fetzer C, Riegel T. Dynamic performance tuning of word-based software transactional memory. In *Proc. the 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, Feb. 2008, pp.237-246.

[17] Wang Z, Yi H, Liu R *et al*. Persistent transactional memory. *IEEE Computer Architecture Letters*, 2015, 14(1): 58-61.

[18] Lu Y, Shu J, Sun L. Blurred persistence in transactional persistent memory. In *Proc. the 31st Symposium on Mass Storage Systems and Technologies*, May 30-June 5, 2015.

[19] Hagmann R. Reimplementing the cedar file system using logging and group commit. In *Proc. the 11th ACM Symp. Operating Systems Principles*, Nov. 1987, pp.155-162.

[20] Yang J, Wei Q, Chen C, Wang C, Yong K L, He B. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, Feb. 2015, pp.167-181.

[21] Zhao J, Li S, Yoon D H *et al*. Kiln: Closing the performance gap between systems with and without persistence support. In *Proc. the 46th Annual IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2013, pp.421-432.

[22] Banakar R, Steinke S, Lee B S, Balakrishnan M, Marwedel P. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proc. the 10th Int. Symp. Hardware/Software Codesign*, May 2002, pp.73-78.

[23] Volos H, Tack A J, Swift M M. Mnemosyne: Lightweight persistent memory. In *Proc. the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011, pp.91-104.

[24] Coburn J, Caulfield A M, Akel A *et al*. NV-Heaps: Making persistent objects fast and safe with next-generation, nonvolatile memories. In *Proc. the 16th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Mar. 2011, pp.105-118.

[25] Wheeler R. fs_mark: A file-system stress test, Mar. 2013. http://sourceforge.net/projects/fsmark. Jun. 2015.
[26] Kim W, Nam B, Park D, Won Y. Resolving journaling of journal anomaly in android I/O: Multi-version B-tree with lazy split. In *Proc. the 12th USENIX Conference on File and Storage Technologies*, Feb. 2014, pp.273-285.
[27] Zhang Y, Swanson S. A study of application performance with non-volatile main memory. In *Proc. the 31st Symposium on Mass Storage Systems and Technologies*, May 30-June 5, 2015.
[28] Condit J, Nightingale E B, Frost C *et al*. Better I/O through byte-addressable, persistent memory. In *Proc. the 22nd ACM Symp. Operating Systems Principles*, Oct. 2009, pp.133-146.
[29] Pelley S, Chen P M, Wenisch T F. Memory persistency. In *Proc. the 41st Annual International Symposium on Computer Architecuture*, Jun. 2014, pp.265-276.
[30] Lu Y, Shu J, Sun L, Mutlu O. Loose-ordering consistency for persistent memory. In *Proc. the 32nd IEEE Int. Conf. Computer Design*, Oct. 2014, pp.216-223.
[31] Intel. Intel® architecture instruction set extensions programming reference, Oct. 2014. https://software.intel.com/enus/isa-extensions. Jun. 2015.
[32] Bhandari K, Chakrabarti D R. Implications of CPU caching on byte-addressable non-volatile memory programming. Technical Report HPL-2012-236, HP Laboratories, 2012.
[33] Rao D S, Kumar S, Keshavamurthy A *et al*. System software for persistent memory. In *Proc. the 9th European Conf. Computer Systems*, Apr. 2014, Article No. 15.
[34] Smullen C, Mohan V, Nigam A, Gurumurthi S, Stan M. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *Proc. the 17th International Conference on High Performance Computer Architecture*, Feb. 2011, pp.50-61.
[35] Intel. Intel® 64 and IA-32 architectures software developer manuals, Sept. 2014. https://software.intel.com/enus/isa-extensions. Jun. 2015.
[36] Jeong S, Lee K, Hwang J *et al*. AndroStep: Android storage performance analysis tool. In *Proc. the 1st European Workshop on Mobile Engineering*, Feb. 2013, pp.327-340.
[37] Wang T, Johnson R. Scalable logging through emerging non-volatile memory. *Proc. the VLDB Endowment*, 2014, 7(10): 865-876.
[38] Sun L, Lu Y, Shu J. DP^2: Reducing transaction overhead with differential and dual persistency in persistent memory. In *Proc. the 12th ACM International Conference on Computing Frontiers*, May 2015, Article No. 24.

Yuan-Chao Xu received his B.S. degree in computer science from Beijing Institute of Technology, Beijing, in 1998, M.S. degree in computer science from Beihang University, Beijing, in 2002, and Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2012, respectively. He is now an assistant professor in the College of Information Engineering at Capital Normal University, Beijing. His current research interests include operating system support for non-volatile memory technologies and computer architecture for emerging technologies. He is a member of CCF and ACM.

Hu Wan received her B.E. degree in computer science from the Capital Normal University, Beijing, in 2013. He is now a Master candidate at the Capital Normal University.

Ke-Ni Qiu received her B.S. degree in instructional technology from the Department of Information Technology, Central China Normal University, Wuhan, in 2001, M.S. degree in computer science from the University of Chinese Academy of Sciences, Beijing, in 2004, and Ph.D. degree in computer science from the City University of Hong Kong, Hong Kong, in 2014 respectively. She is now an associate professor in the College of Information Engineering at the Capital Normal University, Beijing. Her current research interests include embedded systems and non-volatile memory optimizations. She is a member of CCF and ACM.

Tao Li received his Ph.D. degree in computer engineering from the University of Texas at Austin, Austin, in 2004. He is currently a full professor in the Department of Electrical and Computer Engineering at the University of Florida, Gainesville. His research interests include computer architecture, memory and storage system design, virtualization, energy-efficient sustainable computing, and so on. He is a member of ACM and IEEE.

Wei-Gong Zhang received his B.S. degree in computer science from the Department of Computer Science and Technology, Dalian University of Technology, Dalian, in 1989 and his Ph.D. degree in microelectronics from Xidian University, Xi'an, in 2005, respectively. He is currently a full professor in the College of Information Engineering at the Capital Normal University, Beijing. His research interests include reliable embedded systems and device interconnection. He is a senior member of CCF.