Blurred Persistence in Transactional Persistent Memory

Youyou Lu, Jiwu Shu*, Long Sun
Department of Computer Science and Technology, Tsinghua University, Beijing, China
luyouyou@tsinghua.edu.cn, shujw@tsinghua.edu.cn, sun-l12@mails.tsinghua.edu.cn

***Abstract*—Persistent memory provides data persistence at main memory level and enables memory-level storage systems. To ensure consistency of the storage systems, memory writes need to be transactional and are carefully moved across the boundary between the volatile CPU cache and the persistent memory. Unfortunately, the CPU cache is hardware-controlled, and it incurs high overhead for programs to track and move data blocks from being volatile to persistent.**

In this paper, we propose a software-based mechanism, *Blurred Persistence***, to blur the volatility-persistence boundary, so as to reduce the overhead in transaction support.** *Blurred Persistence* **consists of two techniques. First,** *Execution in Log* **executes a transaction in the log to eliminate duplicated data copies for execution. It allows the persistence of volatile uncommitted data, which can be detected by reorganizing the log structure. Second,** *Volatile Checkpoint with Bulk Persistence* **allows the committed data to aggressively stay volatile by leveraging the data durability in the log, as long as the commit order across threads is kept. By doing so, it reduces the frequency of forced persistence and improves cache efficiency. Evaluations show that our mechanism improves system performance by 56.3% to 143.7% for a variety of workloads.**

I. Introduction

Persistent memory is a promising technology to provide data persistence at the main memory level, which recently has been advanced by emerging non-volatile memories (NVMs), such as Phase Change Memory (PCM), Spin-Transfer Torque RAM and Resistive RAM (RRAM). Since data are persistent in main memory, persistent data location gets promoted from the secondary storage to the main memory [1], [2], [3], [4], [5], [6], [7].

In persistent memory, the volatility-persistence boundary has moved to the interface between the *volatile* CPU cache and the *persistent* memory [1], [8], [9], as shown in Figure 1. *Storage consistency*, which ensures that storage systems can recover from unexpected failures, needs to be provided at the memory level in persistent memory. To provide storage consistency, writes from the volatile media (the CPU cache) to the persistent media (the persistent memory) need to be performed in correct order.This *write ordering* leads to frequent I/O halts, and thus significantly degrades system performance [10], [11], [12], [13], [14].

Persistence overhead due to storage consistency gets even higher in persistent memory than in disk-based storage systems. As shown in Figure 1, buffer management in main memory is a white box for disk-based storage systems, while that in the CPU cache is a black box for persistent memory. Pages in main memory are managed by the operating system, and programs can know the status and perform the persistence operation on each page. With persistent memory, the CPU cache is hardware controlled, and programs find it cumbersome to track the status or perform the persistence operation for each cached block. In persistent memory, programs either keep the status of each page in the software, leading to extremely high tracking overhead, or flush the whole cache using cache flush commands (e.g., *clflush* and *mfence*). Since the storage consistency requires frequent persistence operations, system performance dramatically degrades [1], [9], [15], [16], [3].

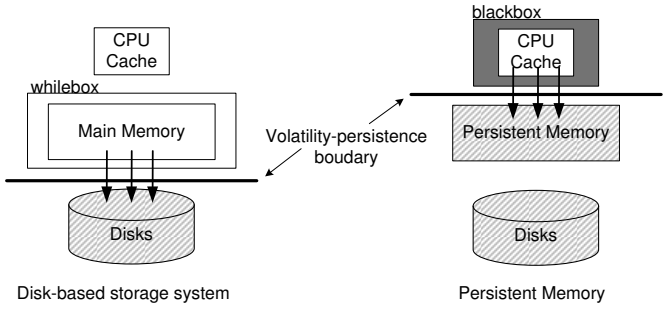


Fig. 1. Volatility-Persistence Boundary in disk-based storage systems and persistent memory.

Recent research has proposed to extend the CPU hardware to mitigate the performance degradation. Approaches of this kind can be divided into two categories. One is to reduce persistence overhead by making the CPU cache non-volatile [17], [8]. The other is to reduce ordering overhead by allowing asynchronous or reordered persistence of transactions [1], [9], [15], [16]. While these approaches effectively improve transaction performance in transactional persistent memory, they require hardware modifications inside CPUs.

In this paper, **our goal** is to design a software-based approach to mitigate the performance degradation in transactional persistent memory. We do so by relaxing the persistence requirements and blurring the volatility-persistence boundary, and call this mechanism *Blurred Persistence*. We have two key observations on the uncommitted data (i.e., data blocks that should stay volatile) and the to-be-persisted data (i.e., data blocks that need to be persisted).

Jiwu Shu is the corresponding author.

978-1-4673-7619-8/15/$31.00

Observation 1. Volatile data can be persisted if they do not damage the persistent data and are detectable after system crashes. Volatile data should be prevented from being written back to persistent memory because of two reasons. First, the uncommitted data that have not been committed can corrupt the persistent data, when they are written back due to cache eviction. In persistent memory, the CPU cache is hardware controlled, and the mapping between data blocks in the CPU cache and those in persistent memory are opaque to the programs. To keep uncommitted data in the CPU cache in order not to be written back and to overwrite/damage the memory data, dividing memory into different areas respectively for uncommitted and to-be-persisted data is an effective approach [3], [7]. When uncommitted data are isolated from to-be-persisted data using different memory areas, the writeback of uncommitted data does not damage the persistent data. However, the isolated memory area brings duplicated data copies among them. Second, the uncommitted data, which are not committed but have been evicted to persistent memory, need to be detected after system crashes. Otherwise, the storage systems in persistent memory have partially updated data, and this leads to inconsistent state. The two problems, however, can be solved by carefully organizing the data structures in persistent memory.

Observation 2. To-be-persisted data may stay volatile if they have persistent copies in other areas. Tracking and forcing persistence of to-be-persisted data also incur high overhead in programs. To make sure that the to-be-persisted data are persisted in time, programs have to record the addresses of these data blocks. When the persistence ordering is required, programs iterate each address and call cache flush commands to force them being written back to persistent memory. The tracking and forced persistence operations lead to poor cache efficiency. However, if the to-be-persisted data have copies elsewhere, which have already been persisted, these do not need to be written back immediately.

Based on the above two observations, we conclude that there are opportunities to relax the volatility or persistence requirements of the uncommitted and to-be-persisted data. Tracking and placing data blocks between the volatile CPU cache and the persistent memory can be relaxed to improve transaction performance. Our proposed *blurred persistence* mechanism has **two key ideas**. First, *Execution in Log (XIL)* allows transactions to be executed in the log area and removes duplicated copies in the execution area. Volatile data are allowed to be persisted in the log area. To enable this, *XIL* reorganizes the log structure to make the uncommitted data detectable in the log. During recovery, the detected uncommitted data can be cleaned from the log while leaving only committed transactions. Second, *Volatile Checkpoint with Bulk Persistence (VCBP)* allows delayed persistence of committed transaction data in each transaction execution and avoids the tracking of to-be-persisted data. This is achieved by making the corresponding log data persistent and maintaining the commit order of checkpointed data across threads. It also aggressively flushes all data blocks from the CPU cache to memory using bulk persistence, with the reorganized memory areas and structures. By doing so, *VCBP* enables more cache evictions and less forced writebacks, and thus improves cache efficiency.

Major contributions of our paper are summarized as follows:

- We identify a major cause of performance degradation while providing storage consistency for persistent memory - tracking and separating uncommitted and to-be-persisted data blocks with a strict boundary.
- We propose *Execution in Log (XIL)*, a technique to enable transaction execution in allocated log area, to allow uncommitted data to be persisted by using a static memory log organization.
- We propose *Volatile Checkpoint with Bulk Persistence (VCBP)* to delay the persistence of checkpoint without tracking data blocks that need to be persisted. This technique allows to-be-persisted data to stay volatile before truncation of the transaction log.
- We implement a transactional persistent memory system, *Blurred-Persistence Persistent Memory (BPPM)*, and evaluate it using a variety of workloads. Results show BPPM gains performance improvement ranging from 56.3% to 143.7%.

The rest of this paper is organized as follows. Section II gives the background of non-volatile memory and transaction recovery. Section III describes the *Blurred Persistence* mechanism, including the *Execution in Log* and *Volatile Checkpoint with Bulk Persistence* techniques. Section IV presents our persistent memory implementation, BPPM, using *Blurred Persistence* mechanism. Then, Section V evaluates BPPM. And finally, Section VI presents related work, and Section VII concludes.

II. Background

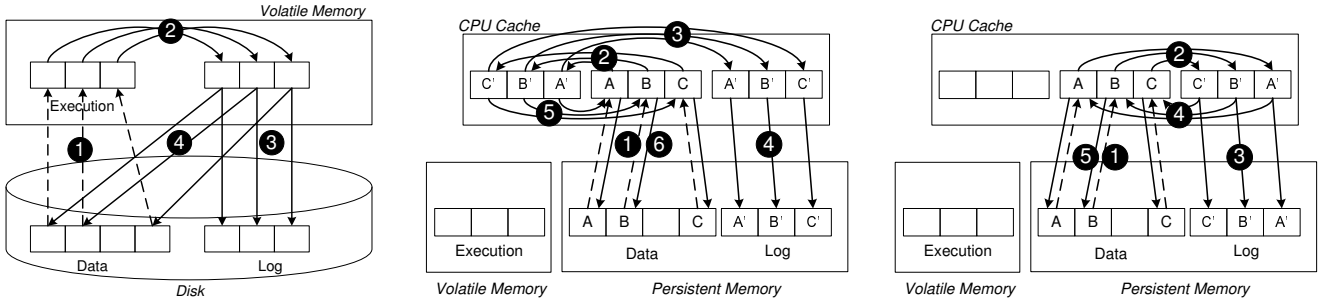
A. Non-Volatile Memory

Byte-addressable non-volatile memories (NVMs) are able to provide data access latency in the order of tens to hundreds of nanoseconds. Phase Change Memory (PCM) [18], [19], [20] is reported to have a read latency of 85ns and a write latency of 100-500ns [21]. Spin-Transfer Torque RAM (STT-RAM) [22] is reported to have read and write latencies less than 20ns [21]. These NVMs not only provide access latency close to that of DRAM, but also show better technology scalability than DRAM. This makes them promising to be used in main memory [22], [18], [19], [20]. In addition, the non-volatility of these NVMs naturally provides data persistence at the memory level, which enables storage systems at main memory level [4], [1], [7], [6], [2], [3], [5].

NVDIMM (Non-Volatile Dual In-line Memory Modules) is another form of byte-addressable non-volatile memory [23]. It is proposed to attach flash memory to the memory bus using byte-addressable interface emulated with DRAM. Data in DRAM are kept persistent using a BBU (Battery Backed Up) or a capacitor when system crashes. NVDIMM provides good performance with data persistence at the memory level.

B. Transaction Recovery

The concept of transaction management originates from database management systems (DBMSs) [24], which introduces transactions to provide ACID properties: atomicity (A),



(a) Transaction Phases in Disk-based Storage Systems

(b) Transaction Phases in Conventional Persistent Memory

(c) Transaction Phases in Blurred-Persistence Persistent Memory

Fig. 2. Transaction Phases in Disk-based Storage Systems, Conventional Persistent Memory, and our proposed Blurred-Persistence Persistent Memory (BPPM).

consistency (C), isolation (I) and durability (D). Transaction management has two components to ensure the four properties: *concurrency control* to allow multiple transactions to be executed concurrently and *transaction recovery* to enable the system recovery from unexpected failures. *Concurrency control* aims to provide execution consistency, which requires each transaction to be atomically executed (i.e., all or none transactional operations are performed) and the concurrently executed transactions are isolated properly. *Transaction recovery* aims to provide persistence consistency, which requires data in persistent storage to be updated atomically and durably. The two components are used together to make sure the concurrently executed transactions are performed correctly to survive system failures, ensuring the ACID properties.

Transactional memory [25], [26], [27] borrows the idea of *concurrency control* to support program concurrency, which requires ACI properties. Recent non-volatile memories provide data persistence (durability) at the main memory level. This leads to opportunities of data recovery at the main memory level. Thus, *transaction recovery* is proposed to be incorporated into transactional memory, which is called *transactional persistent memory*, to provide ACID properties as in database transactions [3], [28].

C. Transactional Persistent Memory

Transactional persistent memory provides both *concurrency control* and *transaction recovery*. To support transaction recovery, a transaction needs to keep at least one version complete in persistent memory. To ensure this property, writes need to be persisted from the CPU cache to the persistent memory in correct order, which is discussed as follows.

Transaction Phases. A transaction has three phases: *execution*, *logging*, and *checkpointing*. In the *execution* phase, data are changed in a volatile area called *execution area* with the execution of program instructions. At this phase, data are prevented from overwriting the old-version data in a persistent place called *data area*, so as to protect the old version. When a transaction is committed, the data can be written to persistent storage. Before being written to the data area, they are first persisted to a persistent area called *log area*; Otherwise, a transaction in the data area may have only part of its data overwritten before system failures, and this hurts the atomicity. Therefore, a transaction first writes its data to the persistent log area when it is committed. This phase is called the *logging* phase. Only after the data have been completely written to the persistent log area, they are checkpointed (i.e., written back from the log area to the data area) to the persistent data area. This phase is called the *checkpointing* phase.

Figure 2 (a) shows the transaction phases in disk-based storage systems, in which the secondary storage is persistent and the main memory is volatile. The volatility-persistence boundary lies between the main memory and the secondary storage. To ensure the storage consistency, the software (i.e., file systems or database management systems) manages the data versions and the write ordering from the main memory to the secondary storage. A transaction is executed following the steps shown in Figure 2 (a). In Step 1, the transaction reads data blocks from the disks to the main memory. In the memory, the transaction is executed in the execution area. After the execution finishes, data pages that need persistence are copied to the operating system page cache, as shown in Step 2. Before these data pages being written to the data area, they are firstly written to the log area, as shown in Step 3. Only when these data pages are completely persisted in the log area, they are checkpointed to the data area, as shown in Step 4.

Figure 2 (b) shows the transaction phases in conventional persistent memory using write-ahead logging, like Mnemosyne [3]. In persistent memory, the volatility-persistence boundary has moved to the line between the volatile CPU cache and the persistent main memory. Since the CPU cache is hardware-controlled, programs separate uncommitted and to-be-persisted data into different memory areas, as shown in Figure 2 (b). In a transaction, data are first read from the *data area* to the CPU cache (as Step 1), and are copied to the *execution area* to be executed (as Step 2); Otherwise, the generated new-version data may be evicted to the memory and overwrite the old-version data in the *execution area*, violating the atomicity property of transactions. When the transaction is committing, the executed data are copied to the *log area* (as Step 3) for logging persistence (as Step 4). After the log has been persisted, the committed data are checkpointed to the *data area* (as Step 5) for checkpoint persistence (as Step 6).

However, it causes high overhead to strictly control the volatility and persistence of each data block. Our goal in this paper is to lower transaction overhead by blurring the volatility-persistence boundary. This can be achieved based on the following two observations. First, transaction data can be executed in the *log area* if the uncommitted data that are evicted to persistent memory are detectable. (Section III-A). Second, checkpointed data, which have persistent copies in

the *log area*, do not need to be immediately flushed back to persistent memory if the overwrite order of those volatile checkpointed data can be maintained correctly (Section III-B). We thus propose the *Blurred Persistence* mechanism for transactional persistent memory. The general framework for our mechanism is illustrated in Figure 2 (c), and details are discussed in Section III.

III. Blurred Persistence

In this section, we propose the *Blurred Persistence* mechanism, which blurs the volatility-persistence boundary in transactional persistent memory, to improve system performance while providing storage consistency. This mechanism consists of two techniques:

1) *Execution in Log (XIL)*, which reorganizes the memory log structure and makes the uncommitted data detectable, to allow the (volatile) uncommitted data to be persisted to the persistent memory.
2) *Volatile Checkpoint with Bulk Persistence (VCBP)*, which leverages the durability of persistent data copies in the log area and maintains the correct overwrite order of the volatile data, to allow the (to-be-persisted) checkpointed data to aggressively stay volatile in the CPU cache.

In this section, we first describe the two techniques, and then discuss the crash recovery.

A. Execution in Log

In persistent memory, uncommitted data should be prevented from being persisted (i.e., being written back from the CPU cache to the persistent memory); Otherwise, the old-version data copies in the persistent memory might be overwritten and corrupted. Since cache replacement in the CPU cache is hardware controlled, software programs are unaware of the cache eviction and are unable to keep the uncommitted data from being persisted. The common approach to separate uncommitted and to-be-persisted data into different memory areas (as shown in Figure 2 (b)) is effective but inefficient due to duplicated data copies. In this section, we argue that duplicating data in different areas is unnecessary in transactional persistent memory. We propose a new technique, *Execution in Log (XIL)*, to allow the (volatile) uncommitted data to be persisted by making them detectable.

With the *Execution in Log (XIL)* technique, a transaction writes its new data directly to the log area rather than to the execution area, no matter whether these data are committed or not. Steps 1 to 3 in Figure 2 (c) illustrate the data flow of a transaction using the *XIL* technique in transactional persistent memory. As shown in the figure, a transaction loads data from the persistent memory and caches them in the CPU cache (shown as Step 1). During the transaction execution, the generated new-version data blocks are allocated with memory space and are written directly in the log area (shown as Step 2). When a transaction commits, these data blocks are forced to be persistent in the persistent log area (shown as Step 3). After the data blocks are persisted in the log area, they are checkpointed (i.e., copied back to their home locations) to the data area (shown as Step 4). And finally, these data blocks are forced to be written back to the persistent data area (shown as Step 5).

The *XIL* technique eliminates data copies to and from the execution area (as illustrated by comparing Figure 2 (b) and (c)). But the challenge of the *XIL* technique is how to detect and remove the uncommitted data blocks that have been written to persistent log area. As shown in step 3 of Figure 2 (c), data blocks from uncommitted transactions may be written to the persistent log area, due to the cache eviction of the CPU cache hardware. The *XIL* technique reorganizes the memory log area and makes these uncommitted data detectable (discussed as follows), to make sure the log area can be used to correct data recovery.

The Log Holes. To support the *XIL* technique, data blocks from uncommitted transactions should be detected and removed from the persistent log area. To make the uncommitted data detectable in persistent memory, *XIL* reorganizes the log structure: (1) Data blocks in the log area are allocated in a log-structured way. Each block has an unique address. With the determined address, an uncommitted block is written to its own location. It neither overwrites any other block nor is overwritten. (2) Uncommitted data blocks are identified using their transaction status metadata in the log, which has associated metadata to indicate the transaction status. For each transaction, there is a block to record these metadata, i.e., a commit record for a committed transaction and an abort record for an aborted transaction. During the normal execution, when a transaction is aborted, its volatile data blocks are discarded without being written back to the persistent memory. This leads to log holes in the log area (i.e., the log blocks that have been allocated but not written to). Log holes are those blocks allocated for uncommitted transactions, but have no data written to.

Figure 3 illustrates an example of the log holes. A transaction, which has four data blocks, is aborted. Among the four blocks, one has been written to the persistent log area due to cache eviction. The other three blocks are discarded in memory without being written back to memory. But they have been allocated with memory area in the log area, and this leads to hole blocks shown as *X* blocks in the figure. Following the four data blocks, an abort record is written at the end to mark the transaction as aborted.

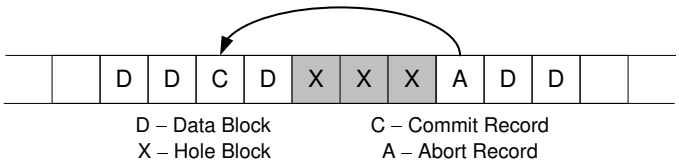


Fig. 3. An Example of Log Holes.

In transactional persistent memory using *XIL* technique, each thread allocates and manages its own persistent log area. The start address of each log is globally visible, so that each log can be read during recovery. Each log consists of a series of 64-bit data and metadata blocks. Since one thread executes one transaction at a time, data blocks for each transaction are written to the log consecutively, followed by one metadata block at the end. Details of the data and metadata block organization are discussed in Section IV. The *XIL* technique

removes the ordering between the persistence of the data blocks and the commit record (i.e., the metadata block) using the torn-bit technique, which is proposed in Mnemosyne [3]. It uses one bit in each data block to indicate the status. Before each run of log writes, the torn bit is set to '0' (or '1'). When these blocks are written, they are set to '1' (or '0'). This bit can be used to detect data blocks that have not been written, i.e., the hole blocks. The *XIL* technique also avoids the use of log head and tail pointers, so as to eliminate the ordering before their updates. To achieve this, the *XIL* technique has to check the log from the beginning and detects the end by itself during recovery.

During recovery, the *XIL* technique needs to correctly process a persistent log area with hole blocks. There are three issues to be addressed: (1) hole blocks should be detectable; (2) uncommitted data blocks that have already been written should be detectable; and (3) valid log blocks that follow the holes should be read and processed. For the first issue, the torn-bit technique can be used to detect data blocks that have not been written, including the hole blocks. For the second issue, *XIL* puts a backpointer in the abort record to point to the commit record of last committed transaction. The backpointer serves as a bridge to straddle the uncommitted blocks (as shown in Figure 3). For the third issue, *XIL* checks the length of each aborted transaction and adds an aborted record for every 64 blocks (the number '64' is adjustable in implementation). During recovery, when an unwritten block is met, the recovery process reads ahead by 64 blocks. Only if there is no aborted record in the 64 blocks, the end of the log is found. There is no valid log block following, and the log scan can be terminated. In addition, as the torn bits in those log holes are not set, they are required to be set during log truncation to ensure the correctness of next run.

B. Volatile Checkpoint with Bulk Persistence

While the *Execution in Log (XIL)* technique allows uncommitted data to be persisted, the *Volatile Check with Bulk Persistence (VCBP)* technique is proposed to allow to-be-persisted data to stay volatile. The *VCBP* technique consists of two steps: *volatile checkpoint* and *bulk persistence*. The *volatile checkpoint* step checkpoints committed data blocks to the data area without forcing them to be written back to persistent memory. It is performed for each transaction execution (shown as Step 4 in Figure 2 (c)). The *bulk persistence* aggressively delays the persistence of checkpointed data until the persistent log area runs out of space. At this time, it forces all data blocks in the CPU cache to be written back to the persistent memory. Persistence of checkpointed data (shown as Step 5 in Figure 2 (c)) is removed during the execution of each transaction.

The *VCBP* technique improves transaction performance without compromising the functionality of the checkpoint operation. The functionality of the checkpointing operation, which makes committed data visible and durable, is still guaranteed in transactional persistent memory using the *VCBP* technique. The *visibility* of committed data is provided with *volatile checkpoint* by copying the committed data to the data area, even though these data blocks are not forced to be persistent. The *durability* of committed data is ensured with the persistent log area, which is not truncated until the *bulk persistence*.

Transaction performance is improved in transactional persistent memory using the *VCBP* technique. This technique not only removes the persistence of checkpointed data from each transaction execution, but also frees programs from tracking (i.e., bookkeeping) of those checkpointed data blocks. It flushes all data blocks, including both the volatile and to-be-persisted data, from the CPU cache to the memory.

The correct issue of the *VCBP* technique is raised from the problem that volatile data blocks, including uncommitted ones, may be written to the persistent memory in both steps, *volatile checkpoint* and *bulk persistence*, of *VCBP*. In the *volatile checkpoint* step, the volatile checkpointed data blocks may have been written to persistent memory due to cache eviction. In the *bulk persistence* step, uncommitted data that do not need persistence are flushed to persistent memory due to the bulk persistence operation. To ensure the correctness of *VCBP* due to blurred the volatility-persistence boundary, two properties are required to be maintained.

Property 1. *Transaction correctness maintains, even if part of its checkpointed data blocks are persisted in advance due to cache eviction.* Checkpointed data blocks are those blocks that are copied to the data area only after their transactions are committed. In other words, these data blocks and all others in their transactions have been completely persisted in the log area. Even if some (not all) checkpointed data blocks are written back due to cache eviction and system crashes, all other data blocks in their transactions can be recovered using persistent log. Thus, the writeback of checkpointed data blocks before bulk persistence does not hurt the completeness of their transactions. Instead, *VCBP* takes more advantage of the writebacks due to cache eviction. It allows the CPU cache to buffer data blocks and write them back on cache conflicts. Cache efficiency is improved compared to the forced writebacks in conventional persistent memory.

Property 2. *Persistent data are protected, even if uncommitted data are forced to be written back due to bulk persistence.* On the bulk persistence operation, uncommitted data are also forced to be written back to memory, as the data blocks that need persistence are not tracked. Since all data in the checkpointing phase are committed data, those uncommitted data include: (1) uncommitted data in the execution phase, and (2) uncommitted data in the logging phase. For uncommitted data in the execution phase, they can be only written to the execution area without hurting any persistent data, which reside in the persistent log and data areas. For uncommitted data in the logging phase, they can be data blocks from uncommitted transactions. They can be detected as discussed in Section III-A. For volatile data in the checkpointing phase, since they are committed, they also do not hurt data in the persistent data area, as discussed for *Property 1*.

In all, *VCBP* improves cache efficiency by removing forced persistence of checkpointed data from each transaction, and frees programs from the complexity of tracking blocks that need persistence.

Overwrite Order of Concurrently-Updated Blocks. With the *VCBP* technique, a data block may have different versions that are committed in different transactions. Since the persistence

of checkpointed data blocks are delayed, different versions of a data block should be written back in correct commit order; Otherwise, a newer version may be overwritten by an earlier one, which leads to inconsistency. Figure 4 shows an example of the overwrite order problem. As shown in the figure, one transaction T_1 commits and checkpoints block A and B (as shown in the left private cache), and another transaction T_2 writes block B and C (as shown in the right private cache). T_2 commits after T_1. Both of them have volatile copies in their own private cache. The *VCBP* technique needs to make sure the data blocks are overwritten in correct commit order, even if a later committed transaction is flushed to memory first. In the illustrated example, it is required that B_1 does not overwrite B_2.

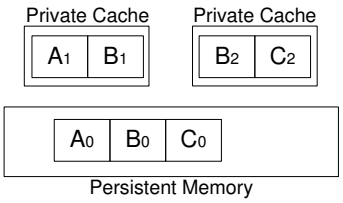


Fig. 4. An Example of the Overwrite Order Problem.

During normal execution of a transaction, the overwrite order is correctly achieved by the cache coherence protocols. When B_2 is checkpointed, it invalidates B_1. For any persistence sequence, only B_2 can be written back. The completeness of the checkpointed data persistence in each transaction is discussed in *Property 1* as above. To ensure the correct overwrite order during recovery, commit sequence between transactions is kept. The *VCBP* technique keeps a global ID as the transaction identifier (TxID), and stores it in the commit record (as discussed in Section IV). The global ID is used to determine the replay sequence of committed transactions during recovery. In this way, the overwrite order of committed transaction is kept, even if their persistence is delayed.

Bulk Persistence vs. Asynchronous Log Truncation. Both of the two techniques move persistence of checkpointed data and log truncation from the critical transaction execution path. *Asynchronous log truncation* forks a thread in background to check the persistence of checkpointed data blocks in each transaction and truncate the log once the checkpointed data blocks are persistent [3]. The difference is that *asynchronous log truncation* still iterates the checks and truncation for transaction one by one while *bulk persistence* removes the tracking and flushes the whole CPU cache without iterating each transaction. *Bulk persistence* also increases the opportunity of (1) write coalescing of data blocks across transactions in the CPU cache, and (2) better cache efficiency with less forced writebacks (but more cache eviction writebacks). Comparetively, *bulk persistence* improves cache efficiency and thereby the transaction performance.

C. Recovery

In persistent memory using *Blurred Persistence* mechanism, programs can not tell the exact location (in the CPU cache or in the persistent memory) of a data block. After unexpected system crashes, uncommitted data blocks may have been written to persistent memory, and checkpointed data may get lost. Therefore, there are two tasks during recovery: (1) finding all uncommitted data that have been persisted, and (2) recovering all checkpointed data that have not been persisted in the data area.

Recovery steps are as follows.

1) *Detection of Uncommitted Data Blocks.* Each log is scanned independently in the first step. The type of each log record (i.e., data record, commit record or abort record) is determined using the metadata in the log record. Since an abort record stores a backpointer to the commit record of the last committed transaction, all data blocks between the commit record and the abort record belong to an uncommitted transaction. As such, the data blocks from uncommitted transactions in the persistent log area are detected.
2) *Commit Sequence Sorting.* In the second step, all committed transactions from different log areas are sorted by the commit sequence, recorded as TxID in each commit record. With the identified commit records from the first step, the recovery process sorts these transactions by the TxID using sort algorithms. After this step, each committed transaction has its global commit sequence.
3) *Replay of Committed Transactions.* In the final step, the committed transactions are replayed by checkpointing their data blocks from the persistent log area to the data area, following the global commit sequence as sorted in the second step. Once these committed data are replayed, the data area are recovered to the latest committed data version.

After all the three steps, all the data blocks in the data area are committed and brought to the newest version. All data blocks are *committed*, because only data blocks from committed transactions are checkpointed during normal execution and only committed transactions have their data blocks replayed to the data area during recovery. All data blocks are *newest*, because all data blocks are checkpointed using *volatile checkpoint*, once a transaction is committed during normal execution, and all committed transactions in the log are replayed in the global sequence during recovery. As such, the data area is guaranteed to be consistent.

IV. Implementation

In this section, we describe the implementation of our persistent memory system that uses *Blurred Persistence* mechanism, which we call *Blurred-Persistence Persistent Memory (BPPM)*.

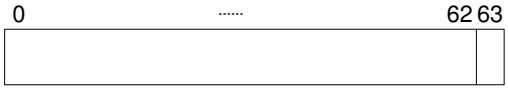
A. Overview of BPPM

We implement BPPM based on a software transactional memory implementation, TinySTM [27]. To leverage BPPM for ACID properties support, a program only needs to be inserted with transaction primitives, which is a light-weight revision. BPPM uses Intel STM compiler [29] to compile programs with transactional annotation using the inserted transaction primitives. The Intel STM compiler compiles the programs and generates transactions. And then, each transaction is ensured with ACID properties in BPPM.

To provide ACID properties for storage consistency, BPPM extends the transactional memory system with persistence support to make the system recoverable. First, versions are kept atomic and durable in persistent memory instead of in the volatile CPU cache. To achieve this, data blocks in the volatile CPU cache are persisted to the log area in persistent memory when a transaction commits, and are not truncated until they are checkpointed and persisted to the data area in persistent memory. As shown in Figure 2 (c), when a transaction commits, data blocks are persisted (as in Step 3) using *clflush* and *mfence* commands. For a checkpoint operation in a transaction, persistence of these data blocks (originally shown as Step 5) is delayed but ensured using the *bulk persistence* operation of the *VCBP* technique. Data blocks in the log area are not truncated until the *bulk persistence* operation. Second, the allocated log area is globally visible. Even though each thread allocates its own log area, the address of the log area is fixed in the persistent memory. After system crashes, these log areas can be located and scanned for recovery. Third, a global sequence for transactions in all log areas is required, so that the commit sequence of transactions across threads can be determined. For transactions with overlapped writes (i.e., two transactions write to the same data block), only when the commit sequence is determined, they can be recovered correctly during recovery.

B. Log Organization

Each thread allocates its own log area. Each log area consists of a series of log records. A log record has a 64-bit data block (as shown in Figure 5) and a 64-bit metadata block (as shown in Figure 6).



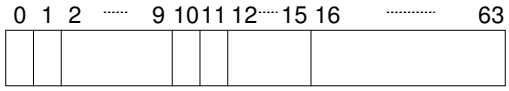
a 64-bit Data Block

Bit(s)	Name	Description
0-62	Data	8-byte data with one bit stored as the tail bit in the metadata block
63	TornBit	the torn bit

Fig. 5. Data Block Format in a Log Record.

In Figure 5, there are 8-byte data, with one bit named *TailBit* borrowed from the metadata block, and one *TornBit* flag. The *TornBit* flag is used to check whether the data block has been written. In persistent memory, a write of a 64-bit block can be an atomic operation [1], [30], [31]. By setting and checking the *TornBit* flag before and after each log run, the unwritten blocks are found (as discussed in Section III-A). In this way, the atomicity of a block write is detectable. Since the *TornBit* flag consumes one bit in the data block, one bit from the metadata block (*TailBit* as shown in Figure 6) is borrowed to make the 8-bytes complete.

In addition to the *TornBit* and *TailBit* as discussed above, there are several flags in the metadata block (as shown in Figure 6) that are used to describe the data block. The *MASK* is a bitmap to indicate which bytes in the data block are valid. It has eight bits, and each bit is corresponded to each byte in the data block. The *FLG_DC* is a flag to indicate whether the data block is a commit/abort record or a data block. And



a 64-bit Metadata Block

Bit(s)	Name	Description
0	TornBit	the torn bit
1	TailBit	the tail bit of the data block
2-9	MASK	valid bitmap of each byte in the data block
10	FLG_DC	bit to indicate whether this is a commit/abort record or a data record
11	FLG_CA	bit to indicate whether this is a commit record or an abort record
12-15	RESV	reversed, not used
16-63	ADDR	address of the data block

Fig. 6. Metadata Block Format in a Log Record.

the *FLG_CA* is further used to differentiate the commit record from the abort record. In a data record, its data block keeps the real data value. In a commit record, its data block keeps the transactional identifier, *TxID*, which is a global ID to determine the commit sequence as discussed in Section IV-A. In an abort record, its data block keeps the backpointer to straddle the aborted records as discussed in Section III-A. Besides the four reserved bits, *RESV*, there is a 48-bit *ADDR* to keep the home location address of the data block.

C. Command Support in the CPU Cache

To persist data from the CPU cache to the persistent memory in software, several commands in the CPU cache need to be enhanced. We use the *clflush* command to force a data block with a specific address in each level of the CPU cache to be written back to the memory, and the *mfence* to prevent the reordering, similar to [2], [5]. We also use the *wbinvd* command to flush all data blocks in the CPU cache to be written back [30], [31]. The *wbinvd* is used for bulk persistence, in which we ensure that there is no side effect on correctness (as discussed in Section III-B). It has been pointed out that *clflush* and *mfence* commands do not guarantee the durability due to the buffer queues in memory controller [7]. Simple enhancement can be easily made to the CPU cache to ensure durability [7]. We believe that the enhancement could also be generalized to the *wbinvd* command for durability.

V. Evaluation

To evaluate the benefits of BPPM, we are going to answer the following questions:

1) How does BPPM benefit from the proposed techniques, *XIL* and *VCBP*?
2) How is BPPM sensitive to the variation of the value size, the transaction size, the size of the log area, the memory latency, and the transaction idle time?

In this section, we first describe the experimental setup before answering the above two questions.

A. Experimental Setup

BPPM Settings. In the evaluation, we compare BPPM with a baseline (BASE) system, the Mnemosyne (MNE) system, and a no-persistence (NP) system. The baseline (BASE) system is a

conventional transactional persistent memory implementation as shown in Figure 2 (b). The Mnemosyne (MNE) system is an optimized implementation of the baseline system, which uses write combing technique [30], [31] to bypass the CPU cache to provide faster log persistence [3]. The no-persistence (NP) system is an ideal transactional persistent memory system which has no persistence operations. The performance is an upper bound of all solutions of persistence optimizations in transactional persistent memory. To evaluate the benefits from different techniques in BPPM, we also perform experiments for BPPM in different modes: *BP(XIL)*, *BP(VCBP)*, and *BPPM*. *BP(XIL)* is referred to as BPPM with only *XIL* technique used. Similarly, *BP(VCBP)* is referred to as BPPM with only *VCBP* technique used. And *BPPM* is referred to as BPPM with both techniques used.

All the evaluations are conducted on a server with a 2.4GHz quad-core AMD Opteron Processor and a 16GB DRAM memory. We emulate the latency of NVM by adding extra latency in each memory flush operation[1]. In the evaluation, the log size is set to one megabyte, and the memory latency is set to 150 nanoseconds by default.

Workloads. Table I lists the workloads that are used in the evaluations, including basic data structures and well-known key-value stores. We evaluate the performance of transactional operations on basic data structures, such as random swaps in a large data array, insert/delete operations in a hash table, and insert/delete operations in a red-black tree, which are also used in previous transactional persistent memory works [4], [9], [8]. The size of all values in these data structures is set to 64 bits by default in the evaluation. We also implement a B+ tree with a node size of 4 KB, and evaluate its transactional operations. Each node in the B+ tree has 200 key-value pairs, and each key or value has a size of 8 bytes. A transaction in the B+ tree evaluation consists of multiple key-value insert or delete operations. In addition, we also evaluate transaction performance of operations on a well-known key-value (KV) system, Tokyo Cabinet [32], to understand the benefits of BPPM in real key-value systems.

TABLE I. WORKLOADS.

Workloads	Description
SPS [4]	Random swaps of array entries
Hash [4]	Insert/delete entries in a hash table
RBTree [4]	Insert/delete nodes in a red-black tree
B+Tree [9]	Insert/delete nodes in a B+ tree
KVStore [32]	Key-value operations on TokyoCabinet

B. Overall Performance

1) Single-thread Evaluation: To focus on the persistence effect, we first run all benchmarks in a single thread to avoid the effects from the concurrency control. We compare BPPM in different modes, including the*BP(XIL)*, *BP(VCBP)*, and *BPPM*, with the baseline (BASE), Mnemosyne (MNE), and no-persistence (NP) systems.

[1]While the simulated memory latencies introduced by software may not accurately reflect the actual latencies when evicting cache lines to persistent memory, we believe that the relative findings across multiple solutions are likely to still hold.

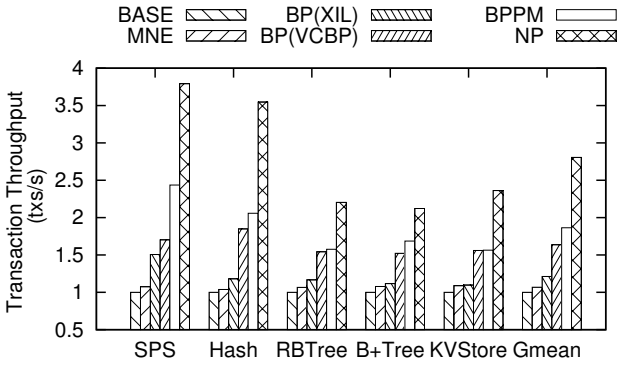


Fig. 7. Transaction Throughput in Single-Thread Mode.

Figure 7 shows the normalized transaction throughput of the afore-mentioned systems using different workloads. All transaction throughputs are normalized to that of the baseline system (BASE). Two observations are in order.

(1) The *Blurred Persistence* mechanism (shown as the white bar in Figure 7) improves system performance significantly over the baseline (BASE) and Mnemosyne (MNE) systems. For the evaluated workloads, the performance improvement in BPPM ranges from 56.3% to 143.7% compared with the baseline (BASE) system, with an average of 86.3%. Compared with Mnemosyne, the performance improvement in BPPM also can be as high as 74.6% on average. While the persistence support in persistent memory has 62.1% performance degradation (by comparing the BASE system to the NP system), BPPM almost halves this overhead.

(2) Both the *XIL* and *VCBP* techniques (respectively shown as the third and fourth bars in each cluster in Figure 7) improve the performance of persistent memory. With the *XIL* technique, performance improvement of the evaluated workloads ranges from 9.7% to 50.5%, with an average of 21.3%. With the *VCBP* technique, performance improvement ranges from 52.2% to 84.8%, with an average of 63.5%. Both the two techniques are effective in performance improvement of transactional persistent memory.

2) Multi-thread Evaluation: To evaluate the performance with both persistence and concurrency control effects, we run benchmarks in multiple threads and vary the number of threads to 1, 2, 4, and 8. We show the multi-thread evaluation results for the hash table and RBTree workloads, while the others have similar patterns and are omitted.

Figure 8 shows the transaction throughputs and the corresponding abort ratios for the hash table and RBTree workloads. The top half of the figure shows the transaction throughputs of each workload with different number of threads. As shown in the left top of Figure 8, the transaction throughput of each evaluated system increases when the number of threads is increased from 1 to 4, but drops when the number of threads is further increased to 8. The reason is that the abort ratio increases dramatically from about 1% to over 50% when the number of threads goes from 4 to 8, as shown in the left bottom of Figure 8. Even though the performance in all evaluated systems drops, BPPM reduces persistence overhead constantly by about 40% in the hash table workload. The right half of

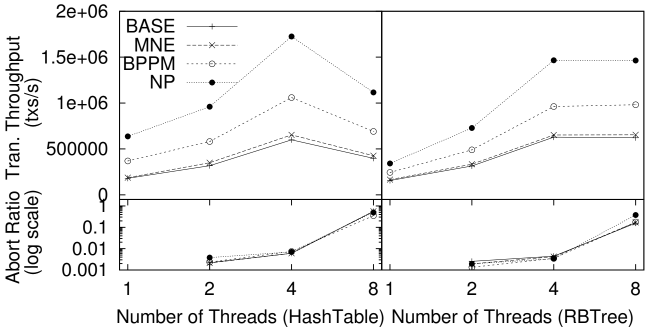


Fig. 8. Transaction Throughput in Multi-Thread Mode.

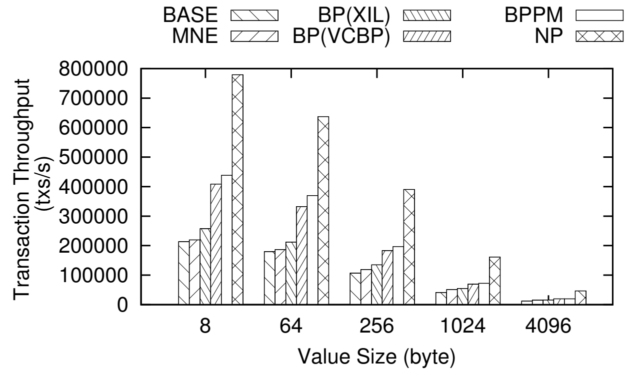


Fig. 9. Sensitivity to the Value Size.

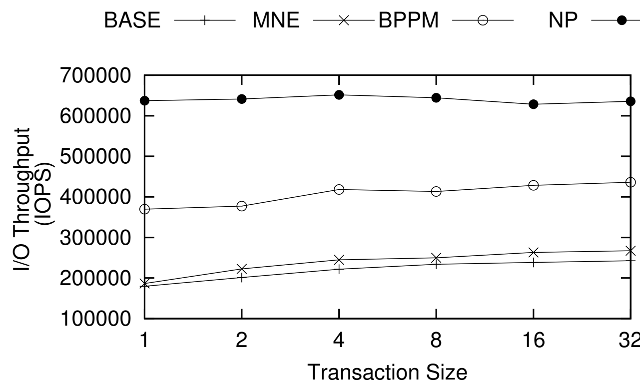


Fig. 10. Sensitivity to the Transaction Size.

Figure 8 shows similar results for the RBTree workload. In the RBTree workload, BPPM reduces persistence overhead by about 43% constantly when the number of threads goes from 1 to 8.

C. Sensitivity Analysis

We evaluate the sensitivity of BPPM to different settings, including the value size of each data structure, the size of the log area, the memory latency, and the transaction idle time. In this subsection, we vary the settings in the hash table workload for the sensitivity evaluation. Since we focus on the persistence overhead that is added to persistent memory, we use a single thread to run the benchmark for the following evaluations.

1) Sensitivity to the Value Size: We measure the transaction throughput of the hash table workload by varying the value size in its data structure from 8, 64, 256, 1024 to 4096 bytes.

Figure 9 shows the transaction throughput of the hash table workload with different value sizes. From this figure, we have two observations. First, as the value size increases, the transaction throughput (in terms of transactions per second) drops, but the byte throughput (in terms of bytes per second) increases. The byte throughput is calculated by multiplying the transaction throughput with the value size. The reason for the increase of the byte throughput is that the amortized persistence overhead per byte decreases. With larger value sizes, a transaction can execute larger bytes before a persistence is required. When the persistence overhead is amortized to each byte, the cost is lowered down. Second, performance improvement in BPPM (in terms of transaction overhead that is reduced in BPPM) drops smoothly from 39.7% with an 8-byte value size to 23.0% with a 4096-byte value size. We conclude that BPPM gains more benefits in workloads with smaller value sizes.

2) Sensitivity to the Transaction Size: To understand the impact of the transaction size, we measure the I/O throughput of the hash table workload by varying the transaction size (i.e., the number of operations in a transaction). I/O throughput is calculated by multiplying the transaction throughput with the transaction size.

Figure 10 shows the I/O throughput of the hash table workload with different transaction sizes. From the figure, we observe that the I/O throughput of the no-persistence (NP) system keeps almost constant while I/O throughputs of the others improve smoothly. The reason why performance in the baseline (BASE), Mnemosyne (MNE) and BPPM systems improves is similar to that given in the value size evaluation. When the transaction size increases, a transaction executes more I/Os before a persistence is required. Due to the reduced amortized overhead, the I/O throughput is improved smoothly. In contrast, persistence operations are removed in the no-persistence (NP) system. Therefore, performance in the no-persistence (NP) system can be hardly affected by the transaction size.

3) Sensitivity to the Log Size: In BPPM, *bulk persistence* is triggered when the log area runs out of space. For this reason, performance of BPPM can be affected by the size of the log area. Meanwhile, different log sizes lead to different recovery times during recovery, as BPPM has to scan the log area for the recovery. To study the impact of the log size, we vary the log size from 0.1 megabytes to 2 megabytes (log size is set to 1 megabyte by default in other evaluations) to measure its implications on transaction throughputs.

Figure 11 shows the transaction throughputs for all evaluated workloads under different log size settings. As shown in the figure, the performance of each workload changes slightly as the log size increases. With a larger log size, *bulk persistence* can be performed less frequently. The frequency of forced writebacks is reduced, and the cache efficiency is improved. As such, the increase of the log size brings more benefits to the transaction performance to a certain degree.

Figure 12 shows the recovery time of BPPM for different log size settings. We show only the results for one workload, because the recovery times in the evaluated workloads are

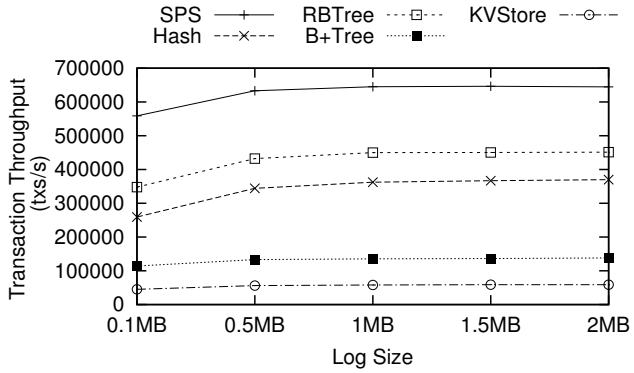


Fig. 11. Impact of Log Size on Transaction Throughput.

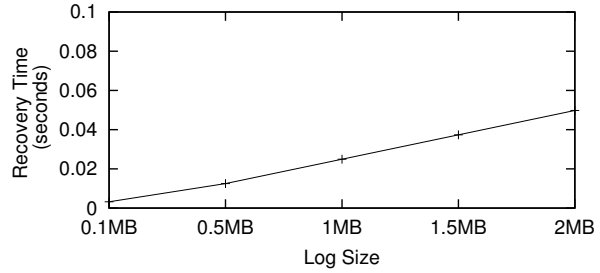


Fig. 12. Impact of Log Size on Recovery Time.

close to each other. As shown in the figure, the recovery time increases almost linearly from 3.2 microseconds with a log size of 0.1 megabytes to 49.8 microseconds with a log size of 2 megabytes. Even with the log size of 2 megabytes, the recovery time is in the order of tens of microseconds, which can be regarded as constant recovery.

We conclude that, with increased size of the log area, the performance in BPPM can be slightly improved, while the recovery is still fast.

4) Sensitivity to the Memory Latency: To study the impact from different non-volatile memory technologies that have different memory access latencies, we set the memory write latency to 35, 95, 150, 1000, and 2000 nanoseconds to measure its implication on transaction throughputs. Since BPPM is implemented to run directly on real servers, we add the latency to each memory flush operation. But note that the latency is not added to memory writes due to cache eviction in the evaluation, because programs are not aware of the cache eviction in the CPU cache hardware.

Figure 13 shows transaction throughputs of the hash table workload with different memory latency settings. In this figure, we omit the performance for the no-persistence (NP) system, because the memory write latency has little effect on the no-persistence system. As the no-persistence system has no persistence operations, it is not sensitive to the memory latency. The figure shows transaction throughputs of other systems. From the figure, we have two observations.

(1) First, in general, these protocols have worse transaction throughputs when the memory latency is higher. However, the BPPM system has poorer performance than the *BP(VCBP)* system when memory latency is high. The reason is that

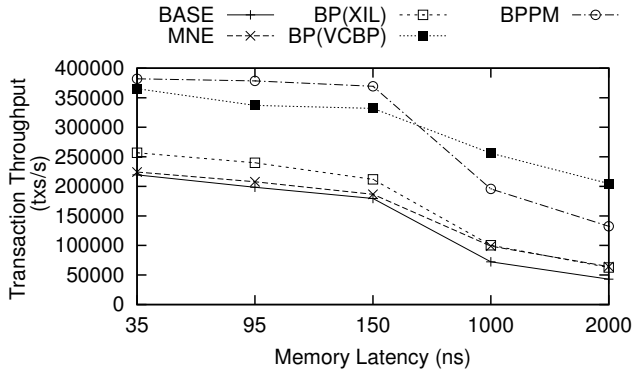


Fig. 13. Sensitivity to the Memory Latency.

the BPPM system forces uncommitted data blocks to be written back to persistent memory and has higher persistence overhead. In BPPM, the *XIL* technique executes data in the log area, which is allocated with memory space in the slow persistent memory rather than in the fast volatile memory (e.g., DRAM). The *bulk persistence* forces all data blocks in the CPU cache, including these uncommitted data in the log area, to be written back. The uncommitted data are forced to be written back to slow persistent memory instead of fast volatile memory (e.g., DRAM), and this leads to degraded performance.

(2) Second, the performance of persistent memory gets worse with higher memory write latency, but the performance benefits from the *blurred persistence* mechanism become higher. The BPPM system outperforms the baseline system by 74.1% when the latency is 35ns, but triples the performance of the baseline system when the latency is 2000ns. The BP(VCBP) system, the BPPM system with only *VCBP* technique used, can even has nearly five times of the performance of the baseline system.

5) Sensitivity to the Transaction Idle Time: Transaction idle time is the time when a program does not execute instructions. It affects the performance of asynchronous operations in transactions. To study this effect, we set the value size to 1024 bytes and vary the percentage of transaction idle time from 90% to 0% (no idle time) to evaluate the sensitivity. In addition to the baseline (BASE), BPPM and no-persistence (NP) systems, we also evaluate the Mnemosyne (MNE) system with synchronous and asynchronous log truncation methods [3], which are respectively denoted as *MNE-SYNC* and *MNE-ASYNC* in Figure 14.

Figure 14 depicts transaction throughputs of the evaluated systems with different percentages of the idle time. As the percentage of the idle time goes down, which means busier programs, the transaction throughput goes up. But, with lower idle time percentage, MNE-ASYNC has poorer performance than MNE-SYNC, which is consistent with the results reported in previous work [3]. This is because the background log truncation thread competes with the foreground transaction execution threads. Comparatively, BPPM has consistently better performance than Mnemosyne with both SYNC and ASYNC log truncation. The reason is that BPPM removes the bookkeeping of to-be-persisted data blocks and has no background threads. Therefore, BPPM gains consistent performance benefits with the optimization dimension of *blurred persistence*.

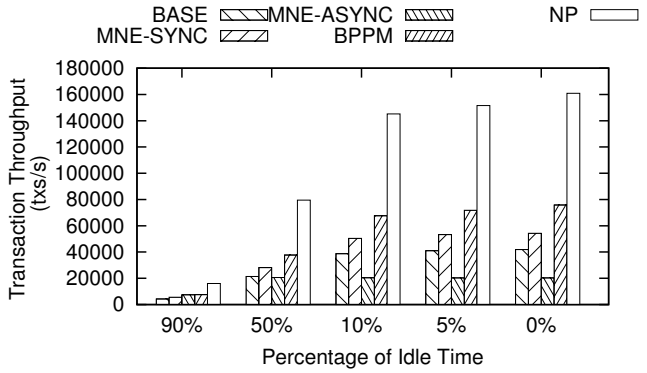


Fig. 14. Sensitivity to the Transaction Idle Time.

VI. Related Work

The transaction mechanism has been widely used to ensure storage consistency in both database management systems [33], [34], [24] and file systems [35], [36]. The design of the transaction mechanism has evolved as the storage media moves from magnetic disks to emerging non-volatile memories. Flash memory has the *no-overwrite* property, i.e., a flash page can not be overwrite until it is erased. To hide the long latency of the erase operation, page writes are redirected to new free pages in flash storage. With this out-of-place update way, both the new and the old versions are kept. Data versioning in transactions is naturally supported. This enables efficient transaction protocols inside storage devices [12], [13], [14], [37], [38], [39], which are designed directly on flash memory to leverage its no-overwrite property and reduce transaction overhead by removing the journal or log writes.

Emerging byte-addressable non-volatile memories (NVMs) are further accelerating the architectural change in transaction design. Transaction designs can be classified into three categories when NVMs are used differently in a storage system. First, when NVMs are used in secondary storage, the internal bandwidth inside a storage device (due to the internal parallelism) is much higher than the device bandwidth. MARS is a transaction protocol that is proposed to copy data for transactions inside devices to exploit the internal bandwidth [40]. Second, when NVMs are used as persistent cache (at memory level) to secondary storage, the persistence at memory level provided by NVMs can be used to persistently keep the transactional data to reduce the transaction overhead in persisting data to the secondary storage [41], [42]. Third, when NVMs are used for data storage at the memory level (a.k.a., *persistent memory*), the transaction mechanism needs to be designed when the data are written back from the CPU cache to the persistent memory. Cache management in the CPU cache is quite different from that in the main memory. Transaction design in persistent memory is a challenge, and has been intensively researched [1], [2], [3], [4], [5], [15], [17], [43], [8], [7], [9], [28], [44]. Our proposed BPPM is used in persistent memory, i.e., the third category of the NVM usage.

In persistent memory, the high transaction overhead comes from not only the persistence overhead that writes multiple levels of the CPU cache, but also the ordering overhead that requires I/Os to be performed in strict order. Thus, existing approaches that reduce the transaction overhead can be classified into the following three categories:

Reducing the Persistence Latency Overhead. In persistent memory, data persistence is achieved by forcing data to be written back to memory through multiple levels (e.g., L1, L2 and LLC) of the CPU cache. Making some or all of the levels non-volatile can reduce the persistence overhead. Kiln [8] is a recently proposed protocol to use fast non-volatile memory as the last-level cache (LLC) in the CPU cache to reduce the persistence overhead [8]. Whole system persistence [17] takes this approach to an extreme. It makes all levels of cache non-volatile and uses backed battery to transfer data through data buses safely even on power failures. In contrast, our proposed BPPM does not require new hardware, but instead, reduces persistence overhead by blurring the volatility-persistence boundary.

In some non-volatile memories like PCM, write operations can be performed faster with lower retention and reliability requirements. Leveraging this property, DP^2 [44] differentially writes the log records and the data records respectively using different write speeds. It also schedules differently for the two kinds of operations. The persistent overhead of the log records is reduced. In contrast, our proposed BPPM is a software approach, which does not rely on specific write properties of NVMs.

Hardware Approaches in Reducing the Ordering Overhead. Relaxing the ordering requirements is another approach to reduce transaction overhead in persistent memory. Since the ordering is ensured at the boundary between the CPU cache and the memory, keeping ordering inside the CPU cache hardware is an efficient approach. BPFS [1] introduces the *epoch* semantic in the CPU cache. Any I/O after the epoch is allowed to be performed only after all I/Os before the epoch are complete. Ordering is kept using epoch in the hardware, and this frees programs from costly waiting in ordering keeping. Similarly, CLC [15] keeps the ordering in the CPU cache with status checking for programs.

Strand Persistency [16] is a relaxed consistency model for persistent memory. It splits I/O dependencies into concurrent threads using program semantics, which allows reordering of I/Os that have no dependency.

LOC [9] introduces the speculative techniques to the CPU cache hardware for I/O persistence. LOC allows reordering of I/Os to persistent memory, but makes the commit sequence visible in program order. PTM [28] takes a similar approach to extend the CPU cache hardware for consistency issues.

BPPM differs from above techniques in two aspects: (1) BPPM is a software approach, while the above mentioned techniques demand for hardware support; (2) BPPM relaxes the persistence requirement, eliminating unnecessary persistence operations, while the above mentioned techniques relax the ordering requirement to allow the reordering of persistence operations.

Software Approaches in Reducing the Ordering Overhead. Ordering overhead in transactions has long been a design challenge in storage systems. New commit protocols, using checksums [45], backpointers [46], [47], and counters [48], [13], have been proposed in traditional storage systems to remove the ordering of the commit records, which forces waiting

for the completeness of all log records. In traditional storage systems, researchers have also studied the asynchronous ordering keeping techniques, which only maintain update dependencies and delay the persistence of update operations [10], [11].

In persistent memory, Mnemosyne [3] proposes two techniques, *torn-bit* and *asynchronous checkpoint*, to reduce ordering overhead in persistent memory. The *torn-bit* technique removes the use of commit record and thus the ordering overhead before commit operations. This technique can be well incorporated and has also been used in BPPM. The *asynchronous checkpoint* technique asynchronously writes the checkpointed data to their home locations in the background, and this benefits programs with more idle time. This technique is not used in BPPM, due to high overhead in tracking those checkpointed data and loss of cache coalescing opportunities across transactions. While Mnemosyne uses asynchronous techniques to hide the write overhead in transactions, our proposed BPPM achieves the same goal by avoiding unnecessary persistence operations, i.e., redundant persistence of duplicated copies that are introduced by strict isolation of the uncommitted data from the to-be-persisted data.

VII. Conclusion

Persistent memory enables memory-level storage, but needs to ensure storage consistency by carefully persisting data blocks to persistent memory in time and in correct order. Strictly tracking and placing data blocks in the volatile CPU cache or in the persistent memory are costly, as the CPU cache is hardware-controlled and unaware of data locations. Our proposed *Blurred Persistence* mechanism blurs the volatility-persistence boundary to reduce this overhead. With this mechanism, uncommitted data blocks (i.e., the volatile data) are allowed to be persisted, only if they are detectable in persistent memory. Checkpointed data (i.e., the to-be-persisted data) are allowed to stay volatile leveraging the durable copies in the log, if they are ensured to be persisted in the correct commit order across threads. The costly bookkeeping of those blocks that need persistence is also removed with bulk persistence, due to the allowance of persistence for uncommitted data. Evaluations of our BPPM implementation show that *Blurred Persistence* is an effective and efficient software-based mechanism for transactional persistent memory.

Acknowledgments

We would like to thank our shepherd Raju Rangaswami and the anonymous reviewers for their comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61433008), the National High Technology Research and Development Program of China (Grant No. 2012AA011003), Samsung Electronics Co., Ltd., Huawei Technologies Co., Ltd., and the State Key Laboratory of High-end Server and Storage Technology (Grant No. 2014HSSA02).

References

[1] 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 *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, 2009, pp. 133–146.

[2] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory." in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*. Berkeley, CA: USENIX, 2011, pp. 61–75.

[3] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2011, pp. 91–104.

[4] 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 *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2011, pp. 105–118.

[5] X. Wu and A. L. N. Reddy, "SCMFS: A file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. New York, NY, USA: ACM, 2011, pp. 39:1–39:11.

[6] 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 *Proceedings of Fifth Workshop on Energy Efficient Design*, 2013.

[7] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*. New York, NY, USA: ACM, 2014, pp. 15:1–15:15.

[8] 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 *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2013, pp. 421–432.

[9] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Proceedings of the IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE, 2014.

[10] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang, "Generalized file system dependencies," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, 2007, pp. 307–320.

[11] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the sync," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Berkeley, CA: USENIX, 2006, pp. 1–14.

[12] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Berkeley, CA: USENIX, 2008, pp. 147–160.

[13] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *Proceedings of the IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 115–122.

[14] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "High-performance and lightweight transaction support in flash-based SSDs," *IEEE Transactions on Computers*, 2015, to appear.

[15] I. Moraru, D. G. Andersen, M. Kaminsky, N. Binkert, N. Tolia, R. Munz, and P. Ranganathan, "Persistent, protected and cached: Building blocks for main memory data stores," Parallel Data Laboratory, Carnegie Mellon University, Tech. Rep. CMU-PDL-11-114v2, 2011.

[16] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014, pp. 265–276.

[17] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2012, pp. 401–410.

[18] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 2009, pp. 2–13.

[19] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 2009, pp. 24–33.

[20] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 2009, pp. 14–23.

[21] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 4, pp. 1–134, 2011.

[22] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *Proceedings of 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 256–267.

[23] "NVDIMM," http://en.wikipedia.org/wiki/NVDIMM.

[24] R. Ramakrishnan and J. Gehrke, *Database management systems*. Osborne/McGraw-Hill, 2000.

[25] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 1993, pp. 289–300.

[26] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory (Synthesis Lectures on Computer Architecture)*, 2010.

[27] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2008, pp. 237–246.

[28] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, "Persistent transactional memory," *Computer Architecture Letters*, 2014.

[29] "Intel© c++ stm compiler, prototype edition," https://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition, 2014.

[30] "AMD64 architecture programmers manual volume 3: General purpose and system instructions," 2011.

[31] "Intel architecture instruction set extensions programming reference, 319433-015," 2013.

[32] "Tokyo Cabinet: a modern implementation of DBM," http://fallabs.com/tokyocabinet/, 2014.

[33] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system R database manager," *ACM Computing Surveys*, 1981.

[34] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, 1992.

[35] S. C. Tweedie, "Journaling the linux ext2fs filesystem," in *The Fourth Annual Linux Expo*, 1998.

[36] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. Soules, and C. A. Stein, "Journaling versus soft updates: Asynchronous meta-data protection in file systems." in *Proceedings of 2000 USENIX Annual Technical Conference*. Berkeley, CA: USENIX, 2000, pp. 71–84.

[37] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2011, pp. 301–311.

[38] Y. Lu, J. Shu, and P. Zhu, "TxCache: Transactional cache using byte-addressable non-volatile memories in SSDs," in *Proceedings of the 3rd IEEE Nonvolatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2014.

[39] Y. Lu, J. Shu, J. Guo, and P. Zhu, "Supporting system consistency with differential transactions in flash-based SSDs," *IEEE Transactions on Computers*, 2015, to appear.

[40] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From ARIES to MARS: Transaction support for next-generation, solid-state drives," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013, pp. 197–212.

[41] D. E. Lowell and P. M. Chen, "Free transactions with Rio Vista," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, 1997, pp. 92–101.

[42] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, "Lightweight recoverable virtual memory," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 1, pp. 33–57, Feb. 1994.

[43] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software persistent memory," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*. Boston, MA: USENIX, 2012, pp. 319–331.

[44] L. Sun, Y. Lu, and J. Shu, "DP2: Reducing transaction overhead with differential and dual persistency in persistent memory," to appear in Proceedings of the ACM International Conference on Computing Frontiers (CF). ACM, 2015.

[45] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Iron file systems," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, 2005, pp. 206–220.

[46] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. Berkeley, CA: USENIX, 2012.

[47] Y. Lu, J. Shu, and W. Wang, "ReconFS: A reconstructable file system on flash storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*. Berkeley, CA: USENIX, 2014, pp. 75–88.

[48] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. Berkeley, CA: USENIX, 2013.