Blurred Persistence: Efficient Transactions in Persistent Memory

YOUYOU LU, JIWU SHU, and LONG SUN, Tsinghua University

Persistent memory provides data durability in main memory and enables memory-level storage systems. To ensure consistency of such storage systems, memory writes need to be transactional and are carefully moved across the boundary between the *volatile* CPU cache and the *persistent* main memory. Unfortunately, cache management in the CPU cache is hardware-controlled. Legacy transaction mechanisms, which are designed for disk-based storage systems, are inefficient in ordered data persistence of transactions in persistent memory. In this article, we propose the *Blurred Persistence* mechanism to reduce the transaction overhead of persistent memory by blurring the volatility-persistence boundary. *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 persistence of the volatile uncommitted data, which are detectable with reorganized 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.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Persistent memory*; D.3.4 [**Programming Languages**]: Processors—*CPU cache management*

General Terms: Design, Performance

Additional Key Words and Phrases: Persistent memory, transaction consistency, persistence

ACM Reference Format:
Youyou Lu, Jiwu Shu, and Long Sun. 2016. Blurred persistence: Efficient transactions in persistent memory. Trans. Storage 12, 1, Article 3 (January 2016), 29 pages.
DOI: http://dx.doi.org/10.1145/2851504

1. INTRODUCTION

Persistent memory is a promising technology to provide data durability at the main-memory level, which recently has been advanced by emerging nonvolatile memories (NVMs), such as Phase Change Memory (PCM), Spin-Transfer Torque RAM (STT-RAM) and Resistive RAM (RRAM). In persistent memory, data durability is ensured in main memory. This makes it possible to build persistent storage systems in the main memory rather than in the secondary storage [Condit et al. 2009; Venkataraman et al.

An earlier version of this article appeared in *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST'15)* [Lu et al. 2015c].
This work is supported by the National Natural Science Foundation of China (Grant No. 61502266, 61232003), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), and China Postdoctoral Science Foundation (Grant No. 2015M580098).
Authors' addresses: Y. Lu, J. Shu (corresponding author), L. Sun, Department of Computer Science and Technology, Tsinghua University, Beijing, China; emails: luyouyou@tsinghua.edu.cn, shujw@tsinghua.edu.cn, sun-l12@mails.tsinghua.edu.cn.

DOI: http://dx.doi.org/10.1145/2851504

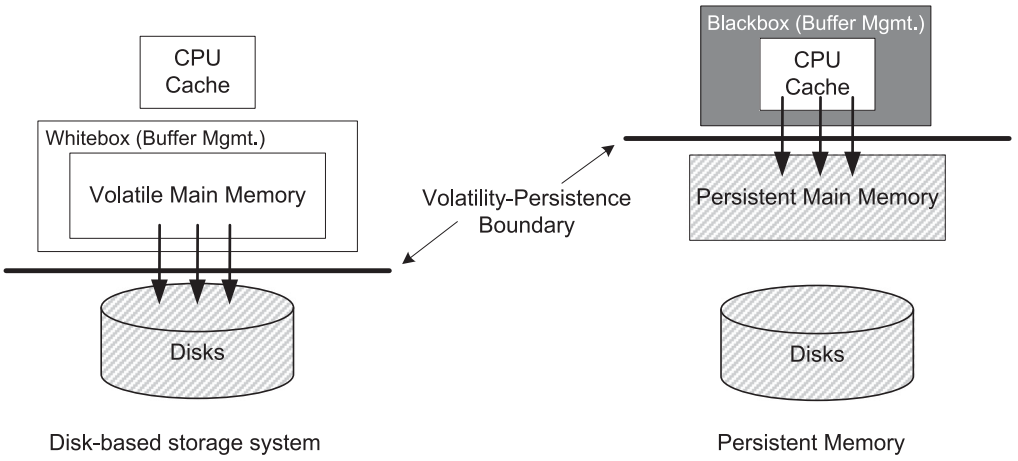


Fig. 1. Volatility-persistence boundary in disk-based storage systems and persistent memory.

2011; Volos et al. 2011; Coburn et al. 2011; Wu and Reddy 2011; Meza et al. 2013; Dulloor et al. 2014].

In persistent memory, the volatility-persistence boundary (e.g., the boundary between the volatile and the persistent storage media) has moved to the interface between the *volatile* CPU cache and the *persistent* main memory[1] [Condit et al. 2009; Zhao et al. 2013; Lu et al. 2014a], 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, persistence operations, which write back data from volatile media (the CPU cache) to persistent media (the persistent main memory), need to be performed in correct order. This *write ordering* leads to frequent I/O halts, thus significantly degrades system performance [Frost et al. 2007; Nightingale et al. 2006; Prabhakaran et al. 2008; Lu et al. 2013a, 2015a].

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. In 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. As a consequence, 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 storage consistency requires frequent persistence operations, system performance dramatically degrades [Condit et al. 2009; Lu et al. 2014a; Moraru et al. 2011; Pelley et al. 2014; Volos et al. 2011].

Recent research has proposed extending 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 nonvolatile [Narayanan and Hodson 2012; Zhao et al. 2013]. The other is to reduce ordering overhead by allowing asynchronous or reordered persistence of transactions [Condit et al. 2009; Lu et al. 2014a; Moraru et al. 2011; Pelley et al. 2014]. While these approaches effectively

[1]In this article, we refer to the NVM media of main memory as *persistent main memory*, while referring to the persistent storage system in main memory as *persistent memory*.

improve transaction performance in transactional persistent memory, they require hardware modifications inside CPUs.

In this article, 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, which we call *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).

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 main memory for 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, cache management in the CPU cache is hardware controlled, and the mappings between data blocks in the CPU cache and those in persistent main memory are opaque to the programs. To keep uncommitted data in the CPU cache in order not to be written back and overwrite/damage the memory data, dividing memory into different areas for uncommitted and to-be-persisted data is an effective approach [Volos et al. 2011; Dulloor et al. 2014]. 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 main memory, need to be detected after system crashes. Otherwise, the storage systems in persistent main memory have partially updated data, which leads to inconsistent state. The two problems, however, can be solved by carefully organizing the data structures in persistent main 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 into being written back to persistent main 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 these 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 main 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, thus improves cache efficiency.

Major contributions of our article 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)*, which enables transaction execution in allocated log area, to allow uncommitted data to be persisted by making them detectable using redesigned log organization.
—We propose *Volatile Checkpoint with Bulk Persistence (VCBP)* to delay the persistence and remove the tracking of data blocks that need to be persisted. This technique allows these to-be-persisted data to stay volatile before log truncation.
—We implement a transactional persistent memory system, *Blurred-Persistence Persistent Memory (BPPM)*, and evaluate it using a variety of workloads. Results show that BPPM gains performance improvement ranging from 56.3% to 143.7%.

The rest of this article is organized as follows. Section 2 gives the background of NVM and transaction recovery. Section 3 describes the *Blurred Persistence* mechanism, including the *XIL* and *VCBP* techniques. Section 4 presents our persistent memory implementation, BPPM, using the *Blurred Persistence* mechanism. Section 5 evaluates BPPM. Section 6 presents related work, and Section 7 contains our conclusions.

2. BACKGROUND

2.1. Nonvolatile Memory

Byte-addressable NVMs are able to provide data access latency in the order of tens to hundreds of nanoseconds. PCM [Lee et al. 2009; Qureshi et al. 2009; Zhou et al. 2009] is reported to have a read latency of 85ns and a write latency of 100ns to 500ns [Qureshi et al. 2011]. STT-RAM [Kultursay et al. 2013] is reported to have read and write latencies of less than 20ns [Qureshi et al. 2011]. 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 [Kultursay et al. 2013; Lee et al. 2009; Qureshi et al. 2009; Zhou et al. 2009]. In addition, the nonvolatility of these NVMs naturally provides data persistence at the memory level, which enables storage systems at the main-memory level [Coburn et al. 2011; Condit et al. 2009; Dulloor et al. 2014; Meza et al. 2013; Venkataraman et al. 2011; Volos et al. 2011; Wu and Reddy 2011].

Nonvolatile Dual Inline Memory Modules (NVDIMM) is another form of byte-addressable NVM [Wikipedia 2015]. It is proposed to attach flash memory to the memory bus using a byte-addressable interface emulated with DRAM. Data in DRAM are kept persistent using a Battery Backed Up (BBU) or a capacitor when the system crashes. NVDIMM provides good performance with data persistence at the memory level.

2.2. Transaction Recovery

The concept of transaction management originates from database management systems (DBMSs) [Ramakrishnan and Gehrke 2000], which introduces transactions to provide ACID properties: atomicity (A), 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 to recover from unexpected failures. *Concurrency control* aims to provide execution consistency, which requires each transaction to be atomically executed (i.e., all or no 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 that the concurrently executed transactions are performed correctly to survive system failures, ensuring the ACID properties.

Transactional memory [Herlihy and Moss 1993; Harris et al. 2010; Felber et al. 2008] borrows the idea of *concurrency control* to support program concurrency, which requires ACI properties. Recent NVMs 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 [Volos et al. 2011; Wang et al. 2014]. Transactional persistent memory provides both *concurrency control* and *transaction recovery*. In this article, we focus on the efficient transaction recovery part in transactional persistent memory.

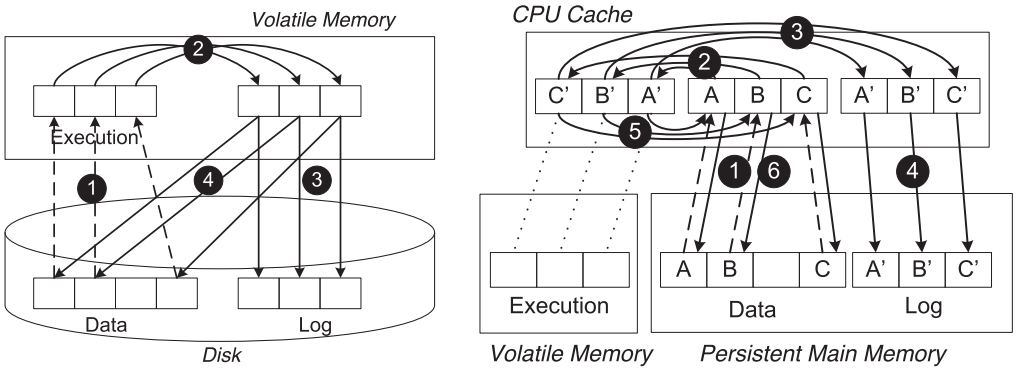
2.3. Transaction Phases

To support transaction recovery, a transaction needs to keep at least one complete version in persistent main memory. To ensure this property, writes need to be persisted from the CPU cache to the persistent main memory in correct order; this forms the transaction phases, to be discussed next.

A transaction has three phases in general: *execution*, *logging*, and *checkpointing*[2]. There are three memory or storage areas for each phase: the *execution area*, *log area*, and *data area*. To keep at least one complete persistent version of transaction data at any time, both the data area and log area must be allocated in persistent storage. In the *execution* phase, transaction data are updated in the execution area with the execution of program instructions. In this phase, transaction data are prevented from overwriting the old-version data in the data area, to keep the persistent old-version data complete. When a transaction is committed, it enters the *logging* phase. In this phase, transaction data are first persisted to the log area. At this time, transaction data cannot be written to the data area, in order to keep the persistent old-version data complete. Only after the data have been completely written to the persistent log area are they 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. In this phase, it is ensured that the persistent new-version data (in the log area) is complete. As such, transaction data are atomically and durably updated to a consistent state with the three phases.

Figure 2(a) illustrates 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. Since the data cache in the main memory is managed by the software (e.g., the operating system), data persistence can be accurately controlled in the software. 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. The memory area for the transaction data is the execution area; the transaction updates data in the execution area. The operating system prevents the updated data from damaging data blocks in the persistent data 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 are written to the data area, they are first written to the log area, as shown in Step 3. Only when these data pages are completely persisted in the log area are they checkpointed to the data area, as shown in Step 4. The operating system tracks the statuses of

[2]Transaction mechanisms have different designs. For simplicity, we use the commonly used write-ahead logging (WAL, i.e., journaling in file systems) as the example.



(a) Transaction Phases in Disk-based Storage Systems

(b) Transaction Phases in Conventional Persistent Memory

Fig. 2. Transaction phases in disk-based storage systems and conventional persistent memory.

transaction data and performs persistence operations in Steps 3 and 4. Due to the software-controlled cache management in main memory, transaction overhead in disk-based storage systems is relatively lower than that in persistent memory.

Figure 2(b) shows the transaction phases in conventional persistent memory using write-ahead logging, such as Mnemosyne [Volos et al. 2011]. In persistent memory, the volatility-persistence boundary has moved to the line between the volatile CPU cache and the persistent main memory. Since cache management in the CPU cache is hardware-controlled, programs have to divide memory spaces into different memory areas to isolate transaction data in different phases, as shown in Figure 2(b). In a transaction, data are first read from the *data area* to the CPU cache (Step 1), and are copied to the *execution area* to be executed (Step 2). If the execution area shares data copies with the data area, some updated data blocks may be evicted to the main memory due to cache eviction, while the others are not. This makes the persistent old-version data in the data area incomplete, violating the atomicity property of transactions.When the transaction is committing, the executed data are copied to the *log area* (Step 3) for log persistence (Step 4). After the log blocks have been persisted, the committed data are checkpointed to the *data area* (Step 5) for checkpoint persistence (Step 6). In persistent memory, transaction data are copied multiple times, because the (software) programs not only are not aware of the cache eviction, but also are not able to control the data eviction. For this reason, transaction overhead is relatively high in persistent memory.

In conclusion, the high transaction overhead in persistent memory comes from the strict data persistence control across the volatility-persistence boundary. Our goal in this article is to lower transaction overhead by blurring the volatility-persistence boundary. We thus propose the *Blurred Persistence* mechanism for transactional persistent memory, as discussed in Section 3.

3. 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:

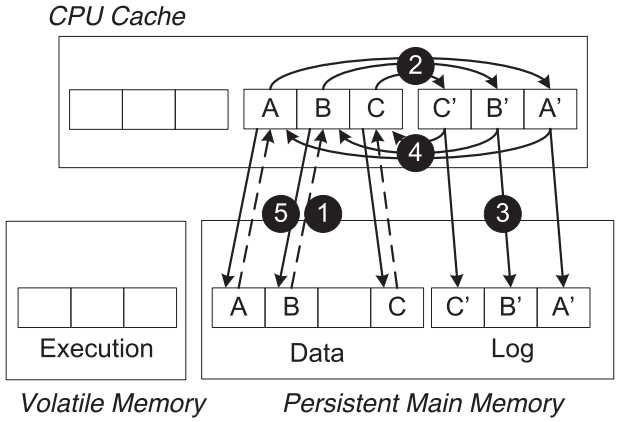


Fig. 3. Transaction phases in Blurred-Persistence Persistent Memory (BPPM).

(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 main 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 present the design overview of blurred persistence. We then describe the two techniques and the adaptive BPPM design. We conclude the section by discussing the crash recovery.

3.1. Design Overview of Blurred Persistence

Figure 3 illustrates the transaction phases in persistent memory with the Blurred Persistence mechanism. The fundamental idea behind Blurred Persistence is blurring the volatility-persistence boundary, in other words, allowing uncommitted data to be persisted in advance and committed data for delayed persistence. In different phases of a transaction, Blurred Persistence uses different techniques to reduce the persistence overhead of transactions. Blurred Persistence consists of two techniques, XIL and VCBP. XIL redesigns the data persistence behaviors in the execution and log phases (shown in Steps 1 to 3 in Figure 3), and VCBP redesigns the data persistence behaviors in the checkpointing phase (shown in Steps 4 and 5 in Figure 3).

XIL removes the execution area in persistent memory and directly executes transactions in the log area. It allows volatile uncommitted data to be persisted before transaction commits. In conventional 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 in the persistent main 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. The goal is to prevent uncommitted data from being written back unconsciously and corrupting the persistent data version. XIL breaks this limitation in the execution and logging phases with the following guarantees: (1) the uncommitted data blocks do not overwrite the

persistent data version in the data area; and (2) the uncommitted data blocks that are written back to the log area in advance are detectable. As shown in Steps 1 to 3 of Figure 3, a transaction reads data blocks from the data area and copies to the log area, then directly updates data blocks in the log area. Before the transaction is committed, the uncommitted data blocks are allowed to be evicted to the persistent log area due to the cache eviction of the CPU cache hardware. Details are given in Section 3.2.

While the XIL technique allows uncommitted data to be persisted, the 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 (Step 4 in Figure 3). 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 main memory. Persistence of checkpointed data (Step 5 in Figure 3) is removed during the execution of each transaction. Details are given in Section 3.3.

XIL and VCBP are designed for different phases of a transaction. The two techniques can be combined to form the Blurred Persistence mechanism. The combination of the two techniques is discussed in Section 3.4.

Blurred Persistence vs. Steal/No-Force Policies. The similarity between Blurred Persistence and the Steal/No-Force policies is that both improve the buffer management efficiency for transactions. However, Blurred Persistence improves buffer management of the CPU cache in persistent memory, while the Steal/No-Force policies improve buffer management of main memory in traditional database management systems. Due to the architectural difference of the two storage systems, they face different problems, thus are designed differently. In Blurred Persistence, XIL allows uncommitted data to be persisted to the log area before the transaction is committed, by redesigning the log organization. The Steal policy also allows uncommitted data to be persisted before the transaction is committed, but instead of to the log area, these uncommitted data are allowed to be persisted to the data area by introducing the *undo* log. VCBP in Blurred Persistence allows committed data to be lazily persisted. The No-Force policy has the same intention. But the No-Force policy can easily track the to-be-persisted committed data in the volatile main memory, while it is costly for programs to do such work in persistent memory due to the hardware-controlled CPU cache. Therefore, VCBP is designed in a more aggressive way, which periodically flushes the whole CPU cache while ensuring the correctness. In conclusion, Blurred Persistence has the same design goal in improving buffer management of transactions as the Steal/No-Force policies. Blurred Persistence is designed for persistent memory, however, for which the Steal/No-Force policies are not suitable.

3.2. Execution in Log

XIL revises the data persistence behaviors in the execution and log phases of a transaction in persistent memory. With the 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 3 illustrate the dataflow of a transaction using the XIL technique in transactional persistent memory. As shown in the figure, a transaction loads data from the persistent main memory and caches them in the CPU cache (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 (Step 2). When a transaction commits, these data blocks are forced to be persistent in the persistent log area (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 (Step 4).

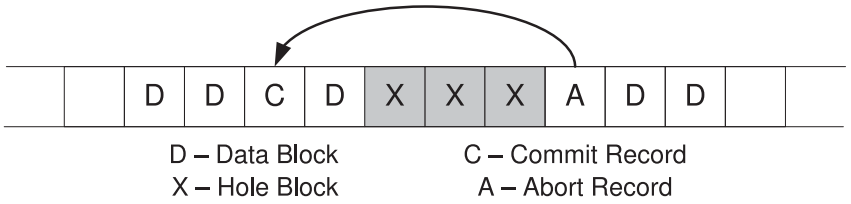


Fig. 4. An example of log holes.

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 Figures 2(b) and 3). The challenge of the XIL technique is how to detect and remove the uncommitted data blocks that have been written to the persistent log area. As shown in Step 3 of Figure 3, 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 later), to make sure that 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 main memory, *XIL* reorganizes the log structure: (1) Data blocks in the log area are allocated in a log-structured way. Each block has a 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 main 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 4 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. They have been allocated with memory area in the log area, however, which 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.

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 64b 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 4. 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 [Volos et al. 2011]. 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, that is, the hole blocks. The XIL technique also avoids the use of log head and tail pointers 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 the last committed transaction. The backpointer serves as a bridge to straddle the uncommitted blocks (as shown in Figure 4). 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 is the end of the log 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 the next run.

3.3. Volatile Checkpoint with Bulk Persistence

VCBP redesigns the data persistence behaviors in the checkpointing phase of a transaction in persistent memory. 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 correctness issue of the VCBP technique is raised from the problem that volatile data blocks, including uncommitted ones, may be written to the persistent main 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 main memory due to cache eviction. In the bulk persistence step, uncommitted data that do not need persistence are flushed to persistent main memory due to the bulk persistence operation. To ensure the correctness of VCBP due to the blurred 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 the persistent log.

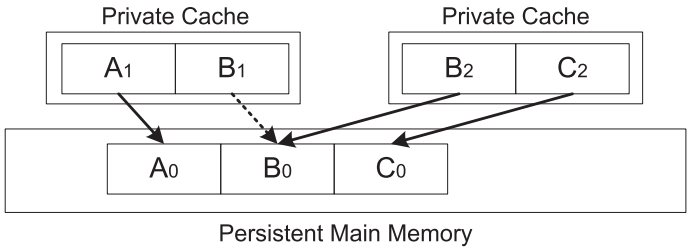


Fig. 5. An example of the overwrite order problem.

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.

In 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 written only 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 3.2. 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 is 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 5 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 have volatile copies in their own private cache. The VCBP technique needs to make sure that 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.

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* earlier. To ensure the correct overwrite order during recovery, the 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 4). The global ID is used to determine the replay sequence of committed transactions during recovery. In this way,

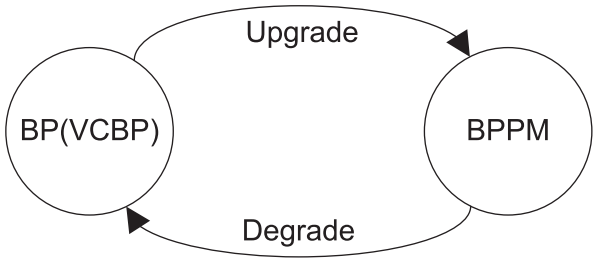


Fig. 6. Mode switch in adaptive BPPM (BPPM-A).

the overwrite order of the committed transactions is kept, even if their persistence is delayed.

Bulk Persistence vs. Asynchronous Log Truncation. Both of these techniques move persistence of checkpointed data and log truncation from the critical transaction execution path. *Asynchronous log truncation* forks a thread in the background to check the persistence of checkpointed data blocks in each transaction and truncate the log once the checkpointed data blocks are persistent [Volos et al. 2011]. 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). Comparatively, bulk persistence improves cache efficiency and, thereby, the transaction performance.

3.4. Adaptive BPPM

The XIL and VCBP techniques are designed for different phases of a transaction. XIL is designed for the execution and logging phases, while the VCBP is for the checkpointing phase. Thus, the two techniques can be simply combined to form the Blurred Persistence mechanism. We refer to the persistent memory that uses both XIL and VCBP as the Blurred-Persistence Persistent Memory (BPPM), and the persistent memory that uses only XIL and only VCBP as BP(XIL) and BP(VCBP), respectively.

While XIL and VCBP can be combined easily, the two techniques may have a negative impact on each other when the memory latency is high (see Figure 16 for evaluated results). Compared to the conventional persistent memory Mnemosyne [Volos et al. 2011], which leverages the write combing command to improve the log performance, XIL writes the data in the CPU cache. The bulk persistence step in VCBP forces all data blocks in the CPU cache to be written back to the main memory, including the data blocks that are written in XIL. For this reason, the benefits of XIL may be eaten by the cost to write the XIL data blocks to persistent main memory, when memory latency is high. To solve this problem, we propose adaptive BPPM, named BPPM-A, which runs BPPM in an adaptive way. When memory latency is low, BPPM is run in the normal mode, with both XIL and VCBP used. When memory latency is high, BPPM is degraded to the VCBP mode, in which BPPM uses only the VCBP technique but no XIL. Figure 6 illustrates the mode switch between the two modes. To support BPPM-A, it has to decide when and how to switch between the BPPM and VCBP modes.

There are two strategies to decide when to switch between the two modes. One is the static strategy, which configures the running mode during program compilation. In this strategy, there is a memory latency threshold. BPPM-A is run in the BPPM mode when the memory latency is less than the threshold, and in the VCBP mode when the

memory latency is higher than the threshold. The other is the dynamic way, which enables mode switching during program execution. In this strategy, the program runs the two modes respectively for a small set of the transactions, then chooses the better mode.

The dynamic strategy has to decide how to switch between the two modes, while the static one can choose one mode during compilation. The main difference between BPPM and VCBP is the log organization in addition to the log write way (cache write with clflush vs. write combing). To support mode switching between the two modes, the log has to support the two kinds of log organizations. Since BPPM is run in only one of the two modes at one time and the log data blocks are sequentially appended, it needs to keep only the address of the log block that starts the mode switch. This information can be stored in a metadata block at the beginning of the log area. During the mode switch, the log writes are synchronized in order not to incur inconsistency between the two modes. As such, the adaptive BPPM can switch to the mode with better performance.

3.5. Recovery

In persistent memory using the Blurred Persistence mechanism, programs cannot tell the exact location (in the CPU cache or in the persistent main memory) of a data block. After unexpected system crashes, uncommitted data blocks may have been written to persistent main 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. Each type of 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 areas are recovered to the latest committed data version.

After all these 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.

4. IMPLEMENTATION

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

4.1. Overview of BPPM

We implement BPPM based on a software transactional memory implementation, TinySTM [Felber et al. 2008]. To leverage BPPM for ACID properties support, a program needs to be inserted only with transaction primitives, which is a lightweight revision. BPPM uses an Intel STM compiler [Intel 2014] to compile programs with transactional annotation using the inserted transaction primitives. The Intel STM compiler compiles the programs and generates transactions. Then, each transaction is ensured with ACID properties in BPPM.

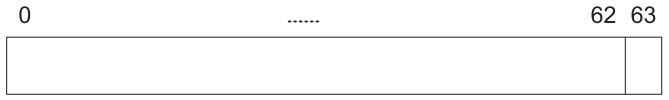
To provide ACID properties for storage consistency, BPPM extends the transactional memory system with persistence support to make the system recoverable. First, versions are kept atomic and durable in persistent main memory instead of in the volatile CPU cache. To achieve this, data blocks in the volatile CPU cache are persisted to the log area in persistent main memory when a transaction commits, and are not truncated until they are checkpointed and persisted to the data area in persistent main memory. As shown in Figure 3, 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. In BPPM, persistence of checkpointed data blocks is performed for multiple transactions rather than for each transaction. 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 main memory. After a 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 overlapping writes (i.e., two transactions write to the same data block), only when the commit sequence is determined can they be recovered correctly during recovery.

To support adaptive mode switching, BPPM-A chooses the dynamic strategy to decide when to check the mode switch. BPPM-A introduces a warm-up phase at the beginning of program execution. In the warm-up phase, BPPM-A runs both BPPM and VCBP modes, and compares their performance to choose a better one. After that, BPPM-A is run in the better mode. This warm-up phase can also be periodically performed in the long program runs to adaptively switch to the better one with workload changes.

4.2. Log Organization

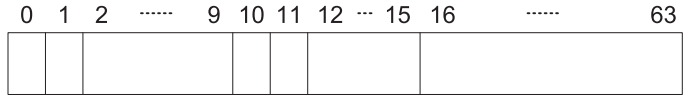
4.2.1. XIL Support. Each thread allocates its own log area. Each log area consists of a series of log records. A log record has a 64b data block (as shown in Figure 7) and a 64b metadata block (as shown in Figure 8).

In Figure 7, there are 8B 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 64b block can be an atomic operation [Condit et al. 2009; AMD 2011; Intel 2013]. By setting and checking the *TornBit* flag before and after each log run, the unwritten blocks are found (as discussed in Section 3.2). In this way, the atomicity of a block write is detectable. Since the *TornBit*



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. 7. Data block format in a log record.



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. 8. Metadata block format in a log record.

flag consumes one bit in the data block, one bit from the metadata block (*TailBit* as shown in Figure 8) is borrowed to form a 64b space for the 8B data.

In addition to the *TornBit* and *TailBit*, there are several flags in the metadata block (as shown in Figure 8) 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 8b, and each bit corresponds 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. 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 4.1. In an abort record, its data block keeps the backpointer to straddle the aborted records, as discussed in Section 3.2. Besides the four reserved bits, *RESV*, there is a 48b *ADDR* to keep the home location address of the data block.

4.2.2. Adaptive BPPM Support. To support BPPM-A, one 64b metadata block is added in the head of each log area. This metadata block keeps a 48b address and a 1b mode flag, while the other 15 bits are reserved. The 48b address is the memory address inside the log that the log organization is changed with the mode switch in BPPM-A. The log records before the address are invalid and can be recycled, and the log records after the address are valid. The mode flag is used to indicate which mode (VCBP or BPPM) has the valid log records. During the recovery, this metadata block in the head of each log can be used to determine the running mode of BPPM-A and to locate the valid log records.

4.3. Command Support in the CPU Cache

To persist data from the CPU cache to the persistent main 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 Venkataraman et al. [2011] and Wu and Reddy [2011]. We also use the *wbinvd* command to flush all data blocks in the CPU cache to be written back [AMD 2011; Intel 2013]. The *wbinvd* command is used for bulk persistence, in which we ensure that there is no side effect on correctness (as discussed in Section 3.3). It has been pointed out that *clflush* and *mfence* commands do not guarantee durability due to the buffer queues in the memory controller. Simple enhancement can be made easily to the CPU cache to ensure durability [Dulloor et al. 2014]. We believe that the enhancement could also be generalized to the *wbinvd* command for durability. In this implementation, we assume that the *clflush* and *wbinvd* commands guarantee data durability.

5. 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, transaction size, size of the log area, and the transaction idle time?
(3) How is BPPM performed adaptively with different memory latencies?

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

5.1. 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 BASE system is a conventional transactional persistent memory implementation, as shown in Figure 2(b). The MNE system is an optimized implementation of the baseline system, which uses the write-combing technique [AMD 2011; Intel 2013] to bypass the CPU cache to provide faster log persistence [Volos et al. 2011]. The no-persistence (NP) system is an ideal transactional persistent memory system that 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 the XIL technique used. Similarly, BP(VCBP) is referred to as BPPM with only the VCBP technique used. 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[3]. In the evaluation, the log size is set to 1MB, and the memory latency is set to 150ns by default.

Workloads. Table I lists the workloads that are used in the evaluations, including basic data structures and well-known key-value (KV) 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

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

Table I. Evaluated Workloads

Workloads	Description
SPS [Coburn et al. 2011]	Random swaps of array entries
Hash [Coburn et al. 2011]	Insert/delete entries in a hash table
RBTree [Coburn et al. 2011]	Insert/delete nodes in a red-black tree
B+Tree [Lu et al. 2014a]	Insert/delete nodes in a B+ tree
KVStore [FAL Lab 2014]	Key-value operations on Tokyo Cabinet

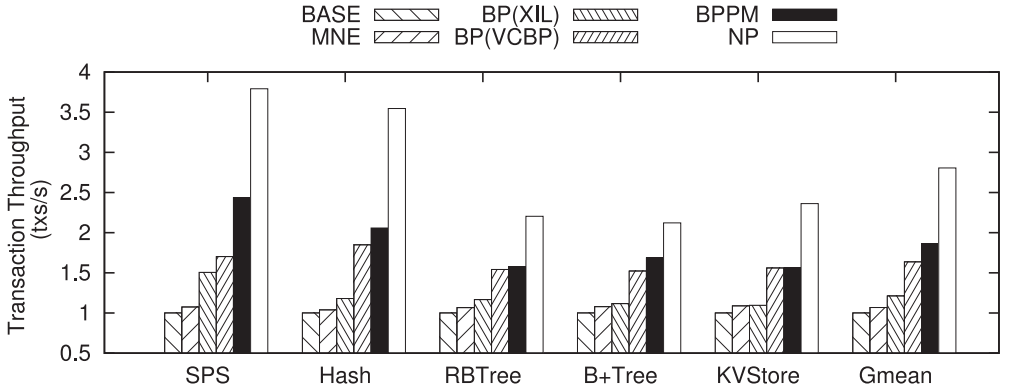


Fig. 9. Transaction throughput in single-thread mode.

works [Coburn et al. 2011; Lu et al. 2014a; Zhao et al. 2013]. The size of all values in these data structures is set to 64b by default in the evaluation. We also implement a B+ tree with a node size of 4KB, and evaluate its transactional operations. Each node in the B+ tree has 200 KV pairs, and each key or value has a size of 8B. A transaction in the B+ tree evaluation consists of multiple KV insert or delete operations. In addition, we also evaluate transaction performance of operations on a well-known KV system, Tokyo Cabinet [FAL Lab 2014], to understand the benefits of BPPM in real KV systems. In the evaluation, all the workloads are run 5,000ms for each setting. The total number of transactions in each workload is in the order of millions.

5.2. Overall Performance

We evaluate the performance of BPPM in both single-thread mode and multithread mode by comparing BPPM in different modes, including BP(XIL), BP(VCBP), and BPPM, with the BASE, MNE, and NP systems.

5.2.1. 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.

Figure 9 shows the normalized transaction throughput of the aforementioned systems using different workloads. All transaction throughputs are normalized to that of the BASE system. Two observations are in order.

(1) The Blurred Persistence mechanism (shown as the black bar in Figure 9) improves system performance significantly over the BASE and MNE systems. For the evaluated workloads, the performance improvement in BPPM ranges from 56.3% to 143.7% compared with the BASE system, with an average of 86.3%. Compared with MNE, the performance improvement in BPPM also can be as high as 74.6%, on average. While the persistence support in persistent memory has a 62.1% performance degradation (by comparing the BASE system to the NP system), BPPM almost halves this overhead.

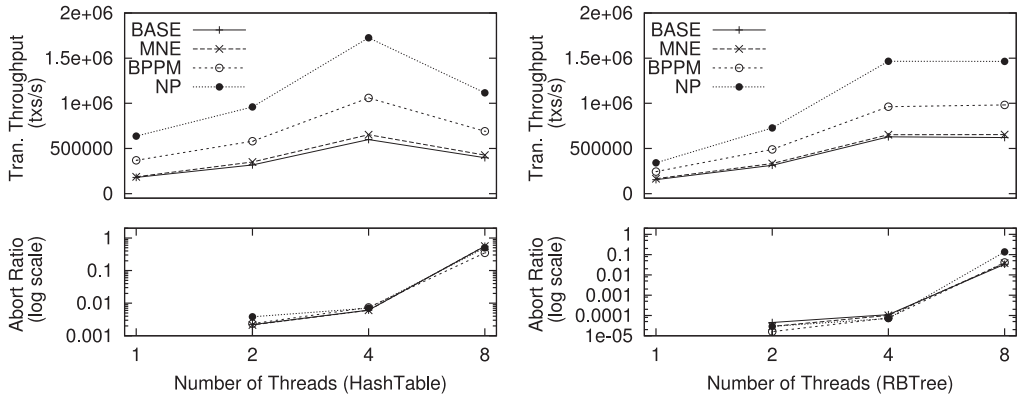


Fig. 10. Transaction throughput in multithread mode.

(2) Both the XIL and VCBP techniques (shown as the third and fourth bars, respectively in each cluster in Figure 9) 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 techniques are effective in performance improvement of transactional persistent memory.

5.2.2. Multithread 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 multithread evaluation results for the hash table and RBTree workloads; the others have similar patterns and are omitted.

Figure 10 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 numbers of threads. As shown at the top left of Figure 10, 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 at the bottom left of Figure 10. 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 Figure 10 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. To conclude, concurrency control has no side effect on blurred persistence, and BPPM can gain benefits in both single-thread and multithread settings.

5.3. 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, and the transaction idle time. In this section, 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.

5.3.1. 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 8B, 64B, 256B, 1024B to 4096B.

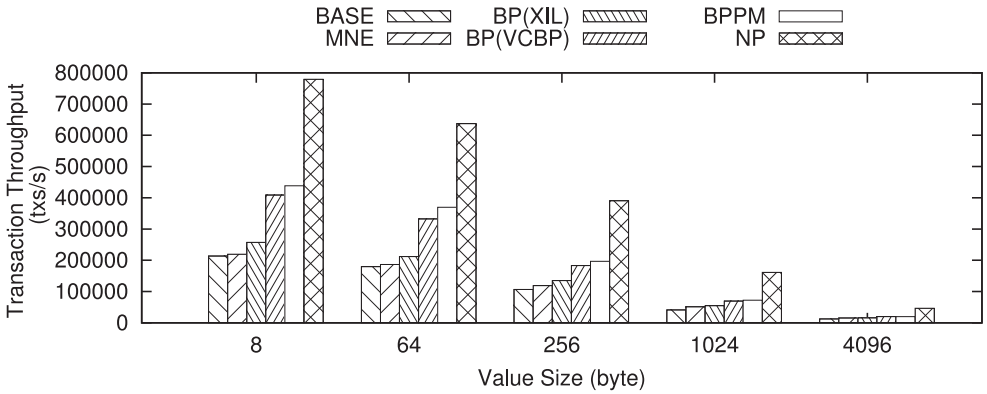


Fig. 11. Sensitivity to the value size.

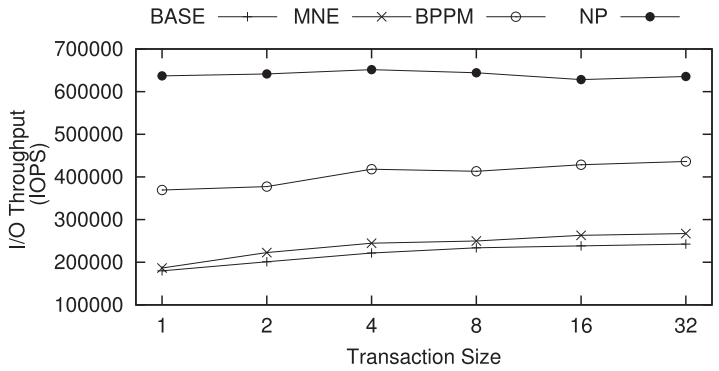


Fig. 12. Sensitivity to the transaction size.

Figure 11 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. Second, performance improvement in BPPM (in terms of transaction overhead that is reduced in BPPM) drops smoothly from 39.7% with an 8B value size to 23.0% with a 4096B value size. We conclude that BPPM gains more benefits in workloads with smaller value sizes.

5.3.2. Sensitivity to 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 12 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 NP system keeps almost constant while I/O throughputs of the others improve smoothly. The reason why performance in the BASE, MNE and BPPM systems improves is similar to

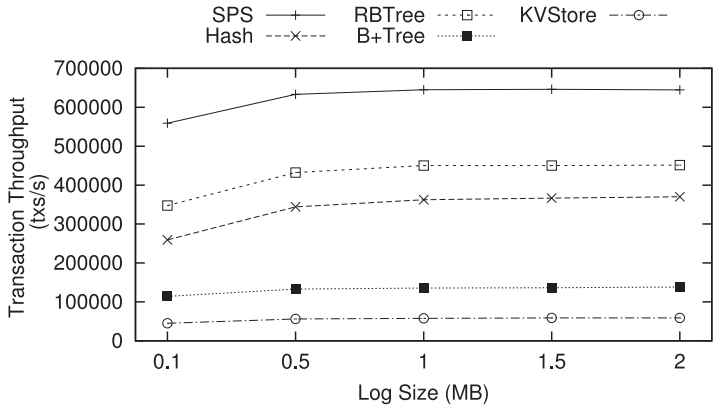


Fig. 13. Impact of log size on transaction throughput.

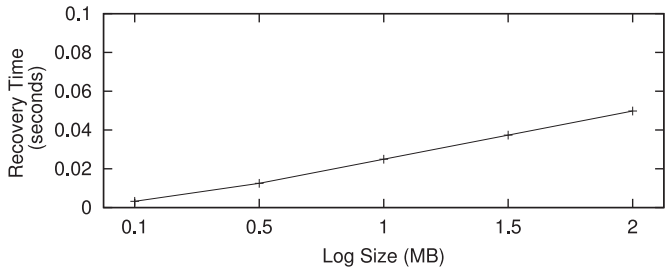


Fig. 14. Impact of log size on recovery time.

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 NP system. Therefore, performance in the NP system can be hardly affected by the transaction size.

5.3.3. Sensitivity to 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 log size, we vary the log size from 0.1MB to 2MB (log size is set to 1MB by default in other evaluations) to measure its implications on transaction throughputs.

Figure 13 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 14 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 close to each other. As shown in the figure, the recovery time increases almost linearly from 3.2μs with a log size of 0.1 megabytes to 49.8μs with a log size of 2MB. Even with the log size of 2MB, the recovery time is in the order of tens of microseconds, which can be regarded as constant recovery.

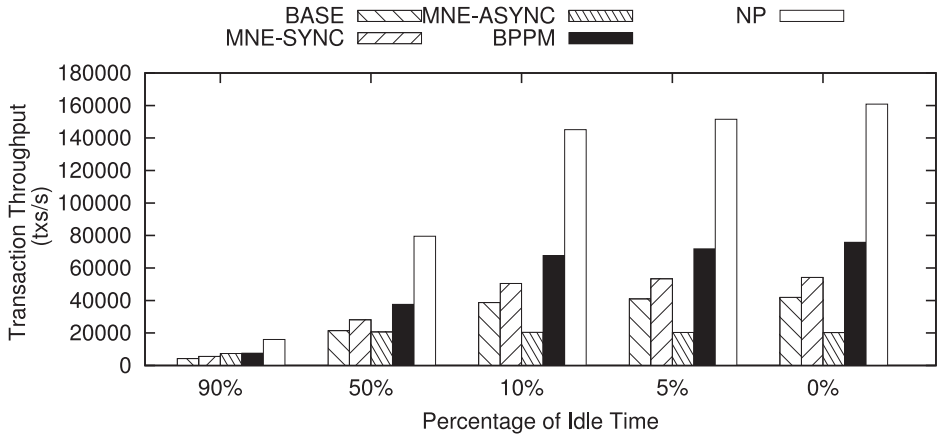


Fig. 15. Sensitivity to the transaction idle time.

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

5.3.4. Sensitivity to 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 1024B and vary the percentage of transaction idle time from 90% to 0% (no idle time) to evaluate the sensitivity. In addition to the BASE, BPPM, and NP systems, we also evaluate the MNE system with synchronous and asynchronous log truncation methods [Volos et al. 2011], which are denoted as *MNE-SYNC* and *MNE-ASYNC*, respectively, in Figure 15.

Figure 15 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. With lower idle time percentage, MNE-ASYNC has poorer performance than MNE-SYNC, which is consistent with the results reported in previous work [Volos et al. 2011]. This is because the background log truncation thread competes with the foreground transaction execution threads. Comparatively, BPPM has consistently better performance than MNE 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.

5.4. Adaptive BPPM (BPPM-A) Evaluation

In this evaluation, we first study the impact on different systems of main memory latencies, then evaluate the performance of BPPM-A under different workloads.

5.4.1. Sensitivity to Memory Latency. To study the impact from different NVM technologies that have different memory access latencies, we set the memory write latency to 35ns, 95ns, 150ns, 1000ns, and 2000ns 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. 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 16 shows transaction throughputs of the hash table workload with different memory latency settings. In this figure, we omit the performance for the NP system,

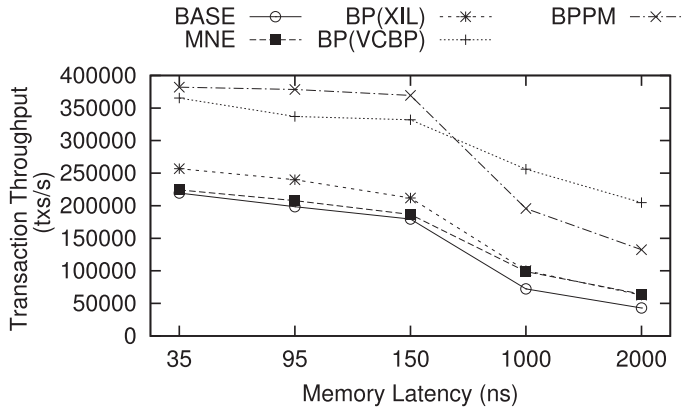


Fig. 16. Sensitivity to the memory latency.

Table II. Warm-Up Time (ms)

Memory Latency (ns)	SPS	Hash	RBTree	B+Tree	KVStore
35	39	49	35	112	216
95	53	49	56	112	202
150	52	52	50	110	210
1000	62	51	60	126	229
2000	62	72	74	128	245

because the memory-write latency has little effect on the NP system. As the NP system has no persistence operations, it is not sensitive to the memory latency. The figure shows transaction throughputs of other systems, for which 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 the BPPM system forces uncommitted data blocks to be written back to persistent main 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 main 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 main memory instead of fast volatile memory (e.g., DRAM), which 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 the VCBP technique used, can even have nearly five times of the performance of the baseline system.

5.4.2. Performance of Adaptive BPPM (BPPM-A). To evaluate the performance of BPPM-A, we measure the transaction throughputs of BP(VCBP), BPPM, and BPPM-A systems for all evaluated workloads. In BPPM-A, the warm-up time is set to the time of running 10,000 transactions.

Table II lists the warm-up time for each workload setting. The warm-up time is tens or hundreds of milliseconds. The overhead of the dynamic strategy in check mode switch is small. Even with the small set, BPPM-A chooses the right mode in

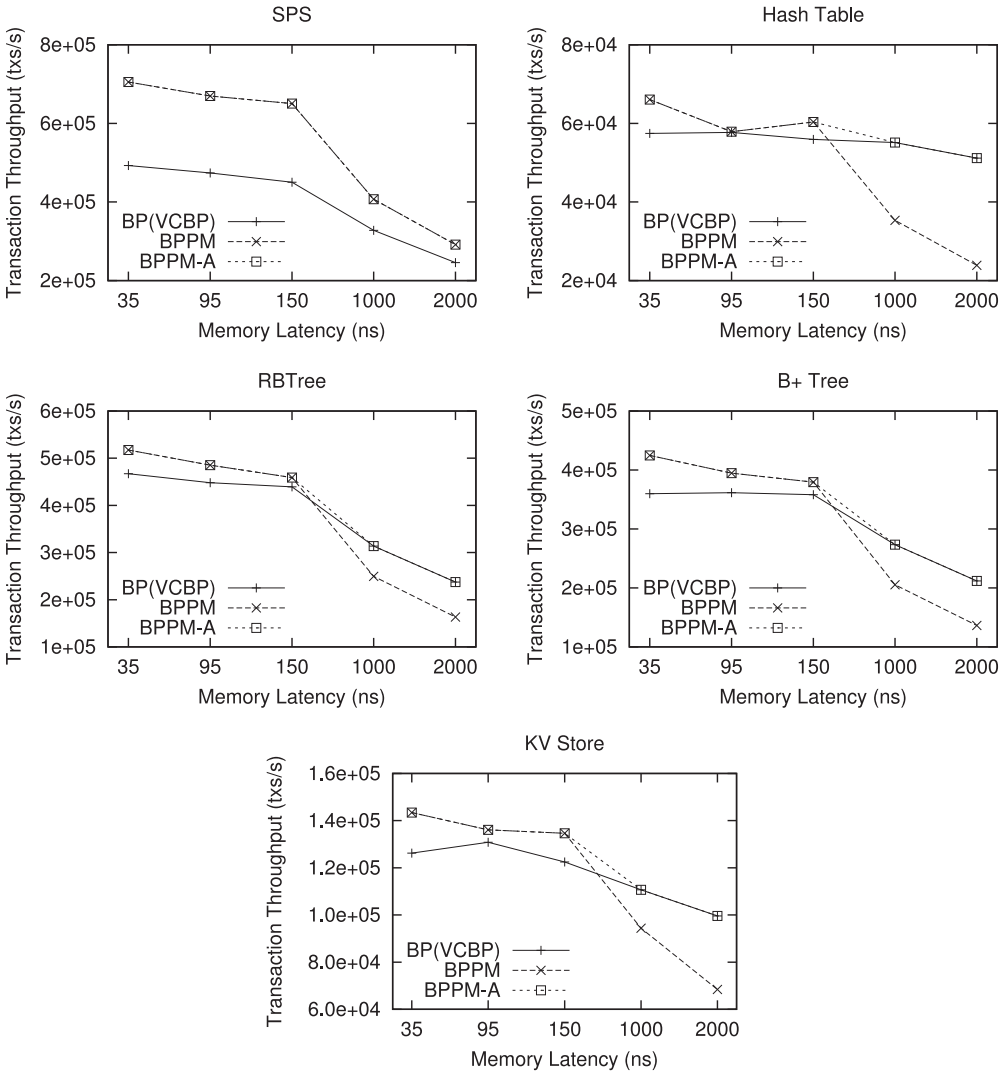


Fig. 17. Performance of adaptive BPPM (BPPM-A).

the warm-up phase. As shown in Figure 6, BPPM-A keeps the best performance in the three systems. In contrast, the static strategy is not flexible. For instance, if the threshold is chosen to be 200ns, the SPS workload cannot be run in the better mode (i.e., the BPPM mode). In conclusion, the dynamic strategy is more accurate in choosing the better mode in BPPM-A, and the overhead of the dynamic strategy is small and acceptable.

Figure 17 shows the transaction throughputs for BP(VCBP), BPPM, and BPPM-A for evaluated workloads. As shown in the figure, all workloads except the SPS workload have worse performance in BPPM than in BP(VCBP) when the memory latency is 1000ns or 2000ns. The reason is that the execution data in addition to the transaction data in XIL are forced to be written to persistent main memory, which incurs extra overhead (as discussed earlier). When the memory latency is high, even the SPS workload still has better performance in BPPM than in BP(VCBP), the performance gap

becomes much smaller. There is no single rule to choose the right mode. We conclude that BPPM-A is necessary and effective.

6. RELATED WORK

The transaction mechanism has been widely used to ensure storage consistency in both database management systems [Gray et al. 1981; Mohan et al. 1992; Ramakrishnan and Gehrke 2000] and file systems [Tweedie 1998; Seltzer et al. 2000]. The design of the transaction mechanism has evolved as storage media methods move from magnetic disks to emerging NVMs. Flash memory has the *no-overwrite* property, that is, a flash page cannot be overwritten 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 method, both the new and the old versions are kept in persistent flash memory. Data versioning in transactions is naturally supported. This enables efficient transaction protocols inside storage devices [Prabhakaran et al. 2008; Lu et al. 2013a, 2015a; Ouyang et al. 2011; Lu et al. 2014c, 2015b], which are designed directly on flash memory to leverage their no-overwrite property and reduce transaction overhead by removing the journal or log writes.

Emerging byte-addressable 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 [Coburn et al. 2013]. 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 [Lowell and Chen 1997; Satyanarayanan et al. 1994; Lee et al. 2013]. Third, when NVMs are used for data storage at the memory level (i.e., *persistent memory*), the transaction mechanism needs to be designed when the data are written back from the CPU cache to the persistent main 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 [Condit et al. 2009; Venkataraman et al. 2011; Volos et al. 2011; Coburn et al. 2011; Wu and Reddy 2011; Moraru et al. 2011; Narayanan and Hodson 2012; Guerra et al. 2012; Zhao et al. 2013; Dulloor et al. 2014; Lu et al. 2014a; Wang et al. 2014; Sun et al. 2015]. Our proposed BPPM is used in persistent memory, that is, the third category of the NVM usage.

In persistent memory, the high transaction overhead comes from not only the persistence overhead that writes through multiple levels of the CPU cache, but also the ordering overhead that requires I/Os to be performed in strict order. Existing approaches try to reduce the transaction overhead from both hardware and software efforts. Thus, we classify existing approaches into different categories, as shown in Figure 18.

Hardware Approaches in Reducing Persistence Overhead (top right of Figure 18). In persistent memory, data persistence is achieved by forcing data to be written back to memory through multiple levels (e.g., L1, L2, and last-level cache (LLC)) of the CPU cache. Making some or all of the levels nonvolatile can reduce persistence overhead. Kiln [Zhao et al. 2013] is a recently proposed protocol to use fast NVM as the LLC in the CPU cache to reduce persistence overhead. Whole System Persistence (WSP) [Narayanan and Hodson 2012] takes this approach to an extreme. It makes all levels of cache nonvolatile and uses backed battery to transfer data through data buses safely, even on power failures. In contrast, our proposed BPPM does not require

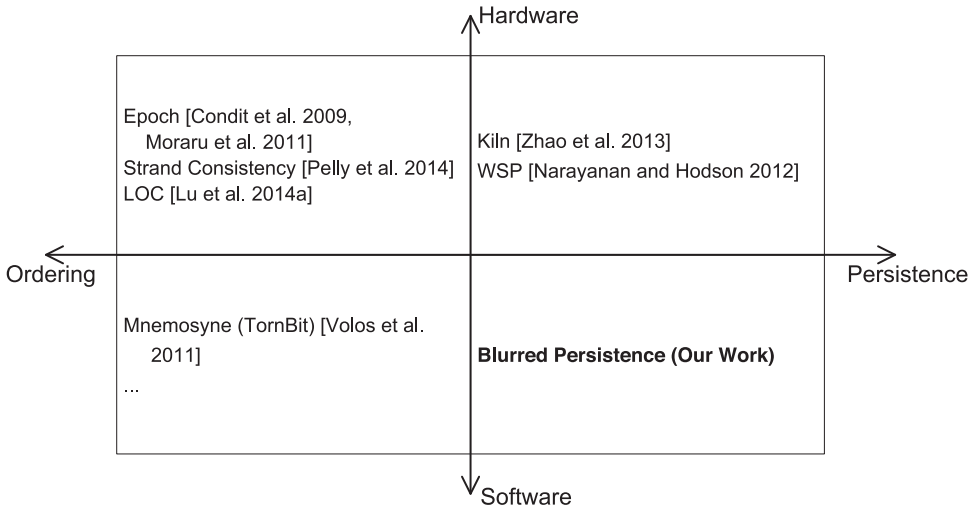


Fig. 18. Classifications of existing approaches in reducing transaction overhead in persistent memory.

new hardware; instead, it reduces persistence overhead by blurring the volatility-persistence boundary.

In some NVMs, such as PCM, write operations can be performed faster with lower retention and reliability requirements. Leveraging this property, DP^2 [Sun et al. 2015] 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 Ordering Overhead (top left of Figure 18). Relaxing the ordering requirements is another approach to reducing 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 [Condit et al. 2009] 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, which frees programs from costly waiting in ordering keeping. Similarly, CLC [Moraru et al. 2011] keeps the ordering in the CPU cache with status checking for programs.

Strand Persistency [Pelley et al. 2014] 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 [Lu et al. 2014a] introduces speculative techniques to the CPU cache hardware for I/O persistence. LOC allows reordering of I/Os to persistent main memory, but makes the commit sequence visible in program order. PTM [Wang et al. 2014] takes a similar approach to extend the CPU cache hardware for consistency issues.

BPPM differs from these techniques in two ways: (1) BPPM is a software approach, while the other techniques need hardware support; (2) BPPM relaxes the persistence requirement, eliminating unnecessary persistence operations, while the other techniques relax the ordering requirement to allow the reordering of persistence operations.

Software Approaches in Reducing the Ordering Overhead (bottom left of Figure 18). Ordering overhead in transactions has long been a design challenge in storage systems. New commit protocols, using checksums [Prabhakaran et al. 2005], backpointers [Chidambaram et al. 2012; Lu et al. 2014b], and counters [Lu et al. 2013b], 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 [Frost et al. 2007; Nightingale et al. 2006].

In persistent memory, NME [Volos et al. 2011] 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, which 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 NME uses asynchronous techniques to hide the write overhead in transactions, our proposed BPPM achieves the same goal by avoiding unnecessary persistence operations, that is, redundant persistence of duplicated copies that are introduced by strict isolation of the uncommitted data from the to-be-persisted data.

In conclusion, Blurred Persistence uses a software approach to reduce the persistence overhead. It falls into a new category in reducing transaction overhead of persistent memory, as shown at the bottom right of Figure 18.

7. 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 main 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 main 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 eliminated 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 the anonymous reviewers and our shepherd Raju Rangaswami for their comments and suggestions to improve the MSST version paper. We also thank Hu Wan, who is a graduate research assistant at Tsinghua University and a graduate student at Capital Normal University, for his assistance of experiments.

REFERENCES

AMD. 2011. AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions.

Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA.

Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. 2013. 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, New York, NY, 197–212.

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, 105–118.

Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, 133–146.

Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. ACM, New York, NY, Article 15, 15 pages.

Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. 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, New York, NY, 237–246.

Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. 2007. Generalized file system dependencies. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, 307–320.

Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The recovery manager of the system R database manager. *Computing Surveys* (1981).

Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software persistent memory. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*. USENIX, Boston, MA, 319–331.

Tim Harris, Jim Larus, and Ravi Rajwar. 2010. *Transactional Memory (Synthesis Lectures on Computer Architecture)* (2nd ed.). Morgan & Claypool, San Francisco, CA.

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, 289–300.

Intel. 2013. Intel architecture instruction set extensions programming reference, 319433-015.

Intel. 2014. Intel© C++ STM Compiler, Prototype Edition. Retrieved December 15, 2015 from https://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition.

Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. 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, 256–267.

FAL Lab. 2014. Tokyo Cabinet: A modern implementation of DBM. Retrieved December 15, 2015 http://fallabs.com/tokyocabinet/.

Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, 2–13.

Eunji Lee, Hyokyung Bahn, and Sam H. Noh. 2013. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA.

David E. Lowell and Peter M. Chen. 1997. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, 92–101.

Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. 2013a. 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, 115–122.

Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. 2015a. High-performance and lightweight transaction support in flash-based SSDs. *IEEE Transactions on Computers* 64, 10, 2819–2832.

Youyou Lu, Jiwu Shu, Jia Guo, and Peng Zhu. 2015b. Supporting system consistency with differential transactions in flash-based SSDs. *IEEE Transactions on Computers* (2015). DOI:http://dx.doi.org/10.1109/TC.2015.2419664 to appear.

Youyou Lu, Jiwu Shu, and Long Sun. 2015c. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st Conference on Massive Storage Systems and Technologies (MSST)*. IEEE, 1–13.

Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014a. Loose-ordering consistency for persistent memory. In *Proceedings of the IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE.

Youyou Lu, Jiwu Shu, and Wei Wang. 2014b. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 75–88.

Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013b. 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)*. USENIX, Berkeley, CA.

Youyou Lu, Jiwu Shu, and Peng Zhu. 2014c. 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.

Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. 2013. A case for efficient hardware/software cooperative management of storage and memory. In *Proceedings of 5th Workshop on Energy Efficient Design*.

C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*.

Iulian Moraru, David G. Andersen, Michael Kaminsky, Nathan Binkert, Niraj Tolia, Reinhard Munz, and Parthasarathy Ranganathan. 2011. *Persistent, Protected and Cached: Building Blocks for Main Memory Data Stores*. Technical Report CMU-PDL-11-114v2. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA.

Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, 401–410.

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Berkeley, CA, 1–14.

Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. 2011. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 301–311.

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 265–276.

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, 206–220.

Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. 2008. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX, Berkeley, CA, 147–160.

Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. 2011. Phase change memory: From devices to systems. *Synthesis Lectures on Computer Architecture* 6, 4, 1–134.

Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, 24–33.

Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems*. Osborne/McGraw-Hill, New York, NY.

M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. 1994. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems* 12, 1, 33–57.

Margo I. Seltzer, Gregory R. Ganger, Marshall K. McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. 2000. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of 2000 USENIX Annual Technical Conference*. USENIX, Berkeley, CA, 71–84.

Long Sun, Youyou Lu, and Jiwu Shu. 2015. DP2: Reducing transaction overhead with differential and dual persistency in persistent memory. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*. ACM, New York, NY.

Stephen C. Tweedie. 1998. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo*.

Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H. Campbell, and others. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 61–75.

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, 91–104.

Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen. 2014. Persistent transactional memory. *Computer Architecture Letters* (2014).

Wikipedia. (2015) NVDIMM. Retrieved December 15, 2015 from http://en.wikipedia.org/wiki/NVDIMM.

Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, New York, NY, Article 39, 11 pages.

Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. 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, New York, NY, 421–432.

Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, 14–23.

Received November 2015; revised November 2015; accepted November 2015