Loose-Ordering Consistency for Persistent Memory

Youyou Lu †, Jiwu Shu † §, Long Sun † and Onur Mutlu ‡
†Department of Computer Science and Technology, Tsinghua University, Beijing, China
§State Key Laboratory of Computer Architecture, Institute of Computing Technology,
Chinese Academy of Sciences, Beijing, China
‡Computer Architecture Laboratory, Carnegie Mellon University, Pittsburgh, PA, USA
luyy09@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn, sun-l12@mails.tsinghua.edu.cn, onur@cmu.edu

***Abstract*—Emerging non-volatile memory (NVM) technologies enable data persistence at the main memory level at access speeds close to DRAM. In such persistent memories, memory writes need to be performed in strict order to satisfy storage consistency requirements and enable correct recovery from system crashes. Unfortunately, adhering to a strict order for writes to persistent memory significantly degrades system performance as it requires flushing dirty data blocks from CPU caches and waiting for their completion at the main memory in the order specified by the program.**

This paper introduces a new mechanism, called Loose-Ordering Consistency (LOC), that satisfies the ordering requirements of persistent memory writes at significantly lower performance degradation than state-of-the-art mechanisms. LOC consists of two key techniques. First, *Eager Commit* reduces the commit overhead for writes within a transaction by eliminating the need to perform a persistent commit record write at the end of a transaction. We do so by ensuring that we can determine the status of all committed transactions during recovery by storing necessary metadata information statically with blocks of data written to memory. Second, *Speculative Persistence* relaxes the ordering of writes between transactions by allowing writes to be speculatively written to persistent memory. A speculative write is made visible to software only after its associated transaction commits. To enable this, our mechanism requires the tracking of committed transaction ID and support for multi-versioning in the CPU cache. Our evaluations show that LOC reduces the average performance overhead of strict write ordering from 66.9% to 34.9% on a variety of workloads.

I. Introduction

Emerging non-volatile memory (NVM) technologies, such as Phase Change Memory (PCM), Spin-Transfer Torque RAM (STT-RAM) and Resistive RAM (RRAM), provide DRAM-like byte-addressable access at DRAM-like latencies and disk-like data persistence. Since these technologies have low idle power, high storage density, and good scalability properties compared to DRAM [1, 2], they have been regarded as potential alternatives to replace or complement DRAM as the technology used to build main memory [3, 4, 5, 6, 7, 8, 9]. Perhaps even more importantly, the non-volatility property of these emerging technologies promises to enable memory-level storage (i.e., persistent memory), which can store data persistently at the main memory level at low latency [10, 11, 12, 13, 14, 15, 16].

Since memory writes in memory-level storage are persistent, they need to be performed atomically and in correct order to ensure *storage consistency*, i.e., consistent state transition for storage systems. Storage consistency ensures atomicity and durability of storage systems, so that the system is able to correctly recover from unexpected system crashes [17, 18, 19, 20, 21, 22, 23, 24], where volatile data gets lost. In order to provide correct recovery on a system crash, multiple related persistent writes are grouped into a *storage transaction* by the programmer. A storage transaction is atomic: either all its writes complete and update persistent memory or none. To accomplish this, both the old and new versions of the data associated with the location of a write are kept track of within the transaction. The writes within and across transactions are *persisted (i.e., written to persistent memory) in strict program order*, to ensure that correct recovery is possible in the presence of incomplete transactions. As such, any persistent memory protocol needs to support both *transaction atomicity* and *strict write ordering to persistent memory (i.e., persistence ordering)* in order to satisfy traditional storage consistency requirements.

Traditionally, disk-based storage systems have employed transaction-based recovery protocols, such as write-ahead logging [17] or shadow paging [18], to provide both transaction atomicity and persistence ordering. These protocols maintain 1) two copies/versions of each written data within a transaction, and 2) a strict write order to the storage device, which enables the atomic switch from the old version of data to the new version upon transaction commit. Unfortunately, traditional transaction-based recovery protocols, designed with disk-based storage systems in mind, are not suitable for memory-level storage due to their large performance overhead when applied to much faster persistent memory.

Transaction support in memory-level storage has two major challenges. First, the boundary of volatility and persistence in memory-level storage lies between the hardware-controlled CPU cache and the persistent memory. In contrast, in traditional disk-based storage systems, the boundary between volatility and persistence lies between the software-controlled main memory and disk storage. While data writeback from main memory is managed by software (i.e., the operating system) in traditional systems, enabling transactional protocols to effectively control the order of writes to persistent storage, data writeback from the CPU cache is managed by hardware in persistent memory systems, making it harder to control the order of writes to persistent memory at low performance overhead. This is because the CPU cache behavior is opaque to the system and application software. Therefore, in order to preserve persistence ordering from the CPU cache to persistent memory, software needs to explicitly include the relatively costly cache flush (e.g., *clflush*) and memory fence (e.g., *mfence*) instructions (at the end of each transaction) to force the ordering of cache writebacks [11, 13, 14, 15]. The average overhead of a clflush and mfence combined together is reported to be 250ns [14], which makes this approach costly, given that persistent memory access times are expected to be on the order of tens to hundreds of nanoseconds [3, 4, 7].

Second, existing systems reorder operations, including writes, at multiple levels, especially in the CPU and the cache hierarchy in order to maximize system performance. For example, writebacks from the cache are performed in an order that is usually completely different from the program-specified order of writes. Similarly, the memory controller can reorder writes to memory to optimize performance (e.g., by optimizing row buffer locality, bank-level parallelism and write-to-read turnaround delays [25]). Enforcing a strict order of writes to persistent memory to preserve storage consistency eliminates the reordering across not only writes/writebacks but also limits reordering possibilities across other operations, thereby significantly degrading the performance. This is because ensuring a strict order of writes requires 1) flushing dirty data blocks from each cache level to memory, 2) writing them back to main memory in the order specified by the transaction at transaction commit time, and 3) waiting for the completeness of all memory writes within the transaction before a single write for the next transaction can be performed. Doing so

Corresponding authors: Jiwu Shu and Onur Mutlu.

This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61327902), the National High Technology Research and Development Program of China (Grant No. 2012AA011003), Shanghai Key Laboratory of Scalable Computing and Systems, Huawei Technologies Co. Ltd., Intel Science and Technology Center for Cloud Computing, US National Science Foundation (CNS 1320531, CCF 1212962), and Tsinghua University Initiative Scientific Research Program.

978-1-4799-6492-5/14/$31.00 ©2014 IEEE

can greatly degrade system performance, as high as by 10 times for some memory-intensive workloads we evaluate, as we demonstrate in Section V-B).

Our goal in this paper is to design new mechanisms that reduce the performance overhead caused by strict ordering of writes in persistent memory. To achieve this, we identify different types of persistence ordering that degrade performance: intra-transaction ordering (i.e., strict ordering of writes inside a transaction) and inter-transaction ordering (i.e., strict ordering of writes between transactions). We observe that relaxing either of these types of ordering can be achieved without compromising storage consistency requirements by changing the persistent memory log organization and providing hardware support in the CPU cache. Based on this observation, we develop two complementary techniques that respectively reduce the performance overhead due to intra- and inter-transaction (tx) ordering requirements. We call the resulting mechanism *Loose-Ordering Consistency (LOC)* for persistent memory.

LOC consists of two new techniques. First, a new transaction commit protocol, *Eager Commit*, enables the commit of a transaction without the use of commit records, traditionally employed for storage systems to record the status of each transaction (which is needed for recovery purposes on system crash) [17, 19, 20, 26]. Doing so removes the need to perform a persistent commit record write at the end of a transaction and eliminates the intra-tx ordering requirement, improving performance. To achieve this, *Eager Commit* allocates the metadata associated with data in the memory log in a static manner - one metadata block is stored with every seven data blocks and the metadata is updated atomically along with data and written to persistent memory. This static log organization enables the system to determine the status of each transaction during recovery without requiring the use/query of a commit record and thus enables the system to recover from crashes effectively as it would know which transactions are fully committed by inspecting the metadata information. Hence, *Eager Commit* removes the recording of data persistence for an entire transaction from the critical path of transaction commit time and delays it until the recovery phase, when the inspection of metadata is truly necessary.

Second, *Speculative Persistence* relaxes the ordering of writes between transactions by allowing writes to be speculatively written to persistent memory. This allows data blocks from multiple transactions to be written to persistent memory, potentially out of the specified program order. A speculative write is made visible to software *only after* its associated transaction commits, and transactions commit in program order. To enable this, our mechanism requires the tracking of committed transaction ID and support for multi-versioning in the CPU cache. Hence, *Speculative Persistence* ensures that storage consistency requirements are met while ordering of persistent memory writes is relaxed, improving performance.

The major contributions of this paper are as follows:

- We identify two types of persistence ordering that lead to performance degradation in persistent memory: intra-transaction ordering and inter-transaction ordering.
- We introduce a new transaction commit protocol, *Eager Commit*, that eliminates the use of commit records (traditionally needed for correct recovery from system crash) and thereby reduces the overhead due to intra-transaction persistence ordering.
- We introduce a new technique, *Speculative Persistence*, that allows writes from different transactions to speculatively update persistent memory in any order while making them visible to software only in program order, thereby reducing the overhead of inter-transaction persistence ordering.
- We evaluate our proposals and their combination, *Loose-Ordering Consistency (LOC)*, with a variety of workloads ranging from basic data structures to graph and database workloads. Results show that LOC significantly reduces the average performance overhead due to persistence ordering from 66.9% to 34.9%.

II. Background and Motivation

A. Non-volatile Memory

Emerging byte-addressable non-volatile memory technologies, also called storage-class memory technologies, have performance characteristics close to that of DRAM. For example, one source [27] reports a read latency of 85ns and a write latency of 100-500ns for Phase Change Memory (PCM). Spin-Transfer Torque RAM (STT-RAM) has lower latency, e.g., less than 20ns for reads and writes [27]. Their DRAM-comparable performance and better-than-DRAM technology-scalability, which can enable high memory capacity at low cost, make these technologies promising alternatives to DRAM [3, 4, 5]. As such, many recent works examined the use of these technologies as part of main memory [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16], providing disk-like data persistence at DRAM-like latencies.

B. Storage Consistency

Storage Transactions. In database management systems, transaction management provides four properties: atomicity (A), consistency (C), isolation (I), and durability (D). To achieve the ACID properties, transaction management has to provide 1) concurrency control for the execution of multiple transactions, 2) transaction recovery in case of failure. Isolation of concurrent execution of multiple transactions is the subject of *concurrency control*, and the atomic and durable update of state by each transaction is the subject of *transaction recovery* [28]. The two concepts are respectively borrowed by *transactional memory* [29, 30] and *storage transactions* [19, 20, 21, 22, 23, 31]. In this paper, we focus on storage transactions, which provide atomicity and durability in the presence of system failures.

Transaction Recovery. Transaction recovery requires data blocks modified by one transaction to be *atomically persisted* to storage devices, such that the persistent data blocks can be used to recover the system to a consistent state *after* an unexpected system crash/failure. To enable this, existing transaction recovery protocols maintain 1) two copies/versions of each written data within a transaction, and 2) a strict write order to the storage device, which enables the atomic switch from the old version of data to the new version upon transaction commit. We briefly describe Write-Ahead Logging (WAL) [17], the state-of-the-art protocol which we use as our baseline.

Write-Ahead Logging (WAL) [17] is a commonly used protocol for transaction recovery. A transaction commit occurs in four phases to ensure correct recoverability of data, as illustrated in Figure 1. In Phase 1 (during transaction execution), WAL writes the new version of each updated data block to the log area in persistent memory, while the old version is kept safe in its home/original location. In Phase 2 (which starts right after the program issues a transaction commit request), WAL first waits until all the data blocks the transaction has updated are written into the log. After this, WAL writes a commit record to the log to keep the transaction status. At the end of Phase II, the new-version data and the commit record are persisted completely and WAL sends an acknowledgment to the program indicating that the transaction commit is done. In Phase 3, WAL copies the new version of each updated data block from the log to its home location to make it visible to accesses from the software (this is called in-place update of data). Finally, after in-place update completes, in Phase 4, WAL truncates the log such that the committed transaction is removed from the log. We call each of these phases an I/O phase.

Ordering. To achieve atomicity and durability, I/O phases are performed one by one, in strict order. This is done to ensure correct recovery in case the system fails/crashes during transaction commit. Updates to persistent memory across the I/O phases are performed in a strict order such that one phase cannot be started before the previous phase is complete. This is called *persistence ordering*. Note that this is different from the ordering of program instructions (loads and stores), which is enforced by the CPU. Persistence ordering is the ordering of *cache writebacks to persistent memory* such that the correct ordering of storage transactions is maintained. As shown in

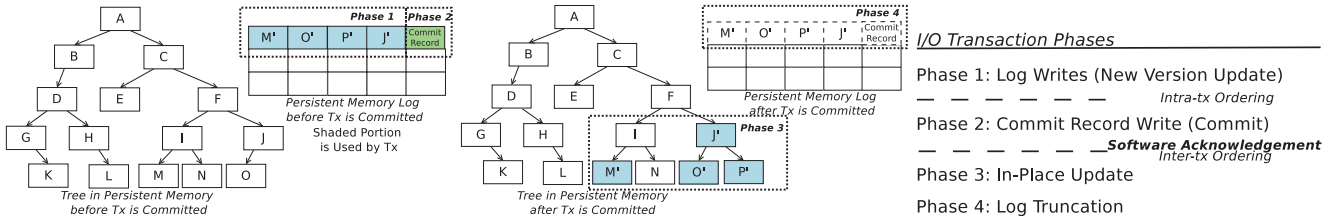


Fig. 1: I/O Phases and Ordering in Write-Ahead Logging (illustrated with a tree data structure in persistent memory).

Figure 1, there are two kinds of persistence ordering in transaction recovery.

Intra-transaction (Intra-tx) Ordering refers to the ordering required within a transaction. Before a transaction commits, WAL needs to ensure that the new version of each updated data block of the transaction is completely persisted. Only after that, WAL updates the commit record. Otherwise, if the commit record is updated before all data blocks updated by the transaction are persisted, the transaction recovery process after a system crash may incorrectly conclude that the transaction is committed, violating atomicity and consistency guarantees. Intra-tx ordering ensures that the new versions of data are completely and safely written to persistent memory when the commit record is found during the transaction recovery process.

Inter-transaction (Inter-tx) Ordering refers to the ordering required across transactions. The program needs to wait for the commit acknowledgment (shown as "Software Acknowledgment" in Figure 1) of a transaction in order to start the next transaction. Inter-tx ordering ensures that the transaction commit order is the same as the order specified by the program.

C. Mitigating the Ordering Overhead

As explained in Section I, in order to preserve persistence ordering from the CPU cache to persistent memory, software combines cache flush (e.g., *clflush*) and memory fence (e.g., *mfence*) instructions to force the ordering of cache writebacks [11, 13, 14, 15]. The average overhead of a clflush and mfence combined together is reported to be 250ns [14], which makes this approach costly, given that persistent memory access times are expected to be on the order of tens to hundreds of nanoseconds [3, 4, 7].[1] The two instructions flush dirty data blocks from the CPU cache to persistent memory and wait for the completeness of all memory writes, and incur high overhead in persistent memory [10, 14, 32, 33].

Several works tried to mitigate the ordering overhead in persistent memory with hardware support [10, 32, 33, 34, 35]. These can be classified into two approaches:

(1) *Making the CPU cache non-volatile*: This approach aims to reduce the time gap between volatility and persistence by employing a non-volatile cache. Kiln [32] uses a non-volatile last-level cache (NV-LLC), so that the path of data persistence becomes shorter and the overhead of the required ordering is smaller. Kiln also uses the NV-LLC as the log to eliminate the need to perform multiple writes in main memory. Whole-system persistence [35] takes this approach to the extreme by making all levels of the CPU cache non-volatile. The approach we develop in this paper, LOC, is complementary to the approach of employing NV caches.

(2) *Allowing asynchronous commit of transactions*: This approach allows the execution of a later transaction without waiting for the persistence of previous transactions. To ensure consistency, the program queries the hardware for the persistence status of transactions. BPFS [10] and CLC [34] use versions of this approach. They inform the CPU cache hardware of the ordering points within the program via the *epoch* command, and let the hardware keep the ordering asynchronously without waiting for data persistence within each epoch. Strand persistency [33] enables the reordering of the commit sequence of transactions for better concurrency (instead of requiring a strict order in which transactions have to be committed). This technique requires the software to inform the hardware which transactions can be reordered, increasing the burden on the programmer/system.

Although these *asynchronous commit* approaches allow data blocks to be written to the CPU cache without waiting for the persistence of previous transactions, data blocks are written to persistent memory in transactions one by one. Strict ordering is still required in persistent memory writes. In other words, *asynchronous commit* approaches change only the execution order (i.e., the order of CPU cache writes) but not persistence order (i.e., the order of persistent memory writes) as specified by the program[2]. In contrast, our proposal, LOC, allows the reordering of persistent memory writes of different transactions in a finer-grained manner and can be combined with the asynchronous commit approaches.

III. Loose-Ordering Consistency

Loose-Ordering Consistency (LOC) is designed to mitigate the performance degradation caused by strict ordering of writes by loosening the ordering without compromising consistency in persistent memory. It aims to reduce both intra-tx and inter-tx ordering overheads. LOC consists of two techniques:

1) *Eager Commit*, a commit protocol, that eliminates the use of commit records, thereby removing intra-tx ordering.
2) *Speculative Persistence* that allows writes from different transactions to speculatively update persistent memory in any order while making them visible to software only in program order, thereby relaxing inter-tx ordering.

This section describes both techniques in detail.

A. Eager Commit

Commit protocol in storage transactions is the consensus between normal execution and recovery on system failure. It is used to determine when to switch between the old and new versions of data that is updated by a transaction. In the commonly used WAL protocol (described in Section II-B), a commit record is used for each transaction to indicate this switch. The commit protocol in WAL makes sure that (1) the new version of data is persisted before writing the commit record, and (2) the old version of the data is overwritten only after the persistent update of the commit record. On a system crash/failure, the recovery logic checks the availability of the commit record for a transaction. If the commit record exists, the transaction is determined to be committed, and the new versions of the committed data blocks are copied from the log to their home locations in persistent memory; otherwise, the transaction is determined to be not committed (i.e., system might have crashed before the transaction commit is complete), and the log data associated with the transaction is discarded.

Unfortunately, it is the commit record itself that introduces the *intra-tx ordering requirement* (described in Section II-B) and therefore degrades performance heavily in persistent memory. *Eager Commit* eliminates the use of the commit record and thus removes the intra-tx ordering. The key idea is to *not* wait for the completeness of log writes and instead eagerly commit a transaction. The completion check of log writes is delayed until the recovery phase. The removal of

[1] Recent research argues that these commands, which are used for cache coherence, do not correctly flush cache data to persistent memory, and thus proposes to ensure ordering in CPU hardware [10, 12].

[2] Even with strand persistency [33], the persistence order is fixed once transaction concurrency has been specified in the program.

completion check from the critical commit path removes the intra-tx ordering and thus reduces commit latency for each transaction. *Eager Commit* enables a delayed completion check at recovery time using a static log organization with a count-based commit protocol, which we describe next. This static log organization enables 1) the system to determine the status of each transaction during recovery without requiring the use/query of a commit record, 2) enables updates of different transactions to the log to be interleaved in the log.

Log Organization. *Eager Commit* organizes the memory log space in a static manner, as opposed to appending all updated data blocks and the commit record at the end of the log for a transaction as done in WAL. It divides the memory log area into block groups, as shown in Figure 2. Each block group consists of eight data blocks, seven for the log data and one for the metadata associated with the seven data blocks. The size of the block group is 64 bytes, which can be transmitted to memory in a single burst [36]. Thus, data and metadata blocks in each block group are written atomically. During recovery after a system crash, the metadata of the block group can be read to determine the status of the data blocks of the block group.

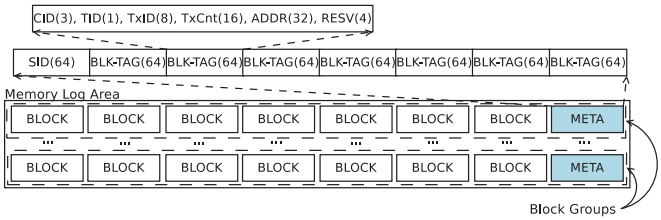


Fig. 2: Memory Log Organization: Every eight blocks form a block group, where one is used for metadata and the others for data.

In a block group, the metadata block stores the sequence ID (*SID*), which is the unique number in the memory log area to represent a block group, and the metadata (*BLK-TAG*) of the other blocks. *BLK-TAG* records the CPU core ID (*CID*), the hardware thread ID (*TID*), the transactional identifier (*TxID*), the transactional counter (*TxCnt*) and the home location address (*ADDR*) of the data in the block. The first three IDs are used to identify the transaction. Therefore, block groups from different transactions can be written to the log in an interleaved manner. On recovery, each data block is identified as belonging to some transaction using the three IDs. Afterwards, the commit protocol uses the pairs $<TxID, TxCnt>$ to determine the transaction status, as described below.

Commit Protocol. In the memory log area, each data block has associated transactional metadata, $<TxID, TxCnt>$. *Eager Commit* uses the pair $<TxID, TxCnt>$ to determine the committed/not-committed status of each transaction. For each transaction, the last data block has its associated *TxCnt* value set to the total number of data blocks in its transaction, and all the others have *TxCnt* set to zero. During recovery, the number of data blocks logged for a transaction (those that have the same *TxID*) is counted and this count is compared with the non-zero *TxCnt* stored with one of the data blocks. If the count matches the non-zero *TxCnt* (indicating that all of the transaction's data blocks are already written to the log), the transaction is deemed to be committed and the recovery process copies its updated blocks from the log to the home locations of the blocks in persistent memory. Otherwise, the transaction is deemed to be not committed and its entries in the log are discarded. This count-based commit protocol is borrowed from [20], where it is described in more detail.

Thus, *Eager Commit* removes the intra-tx ordering by using a static log organization and a count-based commit protocol that can enable the determination of transaction status upon recovery without requiring a commit record.

B. Speculative Persistence

Inter-tx ordering guarantees that the commit sequence of transactions in the storage system is the same as the *commit issue order of transactions by the program* (i.e., the order in which transaction commit commands are issued). To maintain this order, all blocks in one transaction must be persisted to the memory log before any block of a later transaction is persisted. To ensure this, a cache conflict in the CPU cache that causes a block of a later transaction to be evicted must force the eviction of all data blocks of itself and previous transactions. Thus, inter-tx ordering not only causes significant serialization of persistent memory requests but it also results in inefficient utilization of the CPU cache and higher memory traffic, thereby degrading system performance.

Speculative Persistence relaxes inter-tx ordering by allowing blocks from different transactions to be written to the persistent memory log speculatively, out of the software-specified transaction commit issue order. However, the written blocks become visible to software only in the software-specified order. As such, the high-level idea is somewhat similar to out-of-order execution in processors: persistent memory writes are completed out-of the program-specified transaction order (within a window) but they are made visible to software in program transaction commit order. We call this property *"out-of-order persistence, in-order commit"*.

With *Speculative Persistence*, a transaction starts persisting its data blocks without waiting for the completion of the persistence of previous transactions' data blocks. Instead, there is a *speculation window*, in which all transactions are persisted out-of-order. The size of the *speculation window* is called *speculation degree (SD)*. *Speculation degree* defines the maximum number of transactions that are allowed to persist log blocks out-of-order. As such, the inter-tx ordering is relaxed. Relaxed inter-tx ordering brings two benefits. First, cache conflict of one data block does not force eviction of all blocks of its and all previous transactions. This improves the cache utilization. Second, writes from multiple transactions are coalesced when written back to memory, which leads to lower memory traffic.

Figure 3 illustrates transaction persistence in a *speculation window* with a *speculation degree* of four. Within the *speculation window*, data blocks from the four transactions can be persisted in any order. For instance, blocks in T3 can be persisted before blocks in T2. For a data block that has multiple versions across transactions, only the latest version needs to be persisted. All blocks, A, B, C, D, in T1 do not need to be written to memory because their later versions, block A in T2 and blocks B, C, D in T3, will overwrite them. Therefore, inter-tx ordering is relaxed within the *speculation window*.

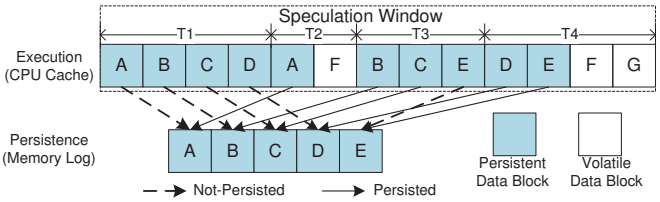


Fig. 3: Illustration of Speculative Persistence.

Figure 3 also illustrates that *Speculative Persistence* preserves the "out-of-order persistence, in-order commit" property: transaction T1 is reported to be committed while T3 is not, because T2 has not finished its updates to the persistent memory log. To preserve this property, *Speculative Persistence* has to carefully deal with 1) overlapping writes from different transactions (to the same block) to ensure that any write to any block is recoverable, and 2) commit dependencies between transactions to ensure that transactions are committed in program order. To enable the former, our mechanism supports multi-versioning in the CPU cache. To enable the latter, our proposal not only leverages multi-versioning in the CPU cache but also keeps track of the committed transaction ID based on the commit issue order of transactions by the program. Both of these, we describe next.

Multiple Versions in the CPU Cache. In *Speculative Persistence*, multiple versions of a data block are maintained in the volatile CPU

cache, similarly to the versioning cache [37]. Otherwise, if only a single copy (the latest copy) is kept but the transaction that last wrote to the block aborts, all previously committed transactions that were supposed to write to the block would also *have to be aborted*. With multiple versions of a data block present in the cache, one version can be removed only when one of its succeeding versions has been committed (i.e., the software has committed a later-in-program-order transaction that writes to the data block). This is because the committed version in a future transaction that is committed later in program order is guaranteed to overwrite any previous version regardless of whether the transaction that is supposed to write to the previous version is aborted or committed.

There are two issues with keeping multiple versions in the cache: 1) *version overflow*, 2) increased cache pressure. First, version overflow refers to the case that the associativity of the cache is not enough to keep all active versions of a cache block (or different cache blocks) in the corresponding cache set. When this happens, our design evicts the oldest version to the memory log to make space for the new version. This eviction can reduce the benefit of *Speculative Persistence*: probability of merging of writes from different transactions reduces and an old version of the block may be written to persistent memory unnecessarily. However, this eviction does not affect the correctness of the commit protocol. Second, since multiple versions of a block are kept in the cache set, the pressure on the cache set increases compared to a conventional cache, which may lead to higher miss rates. Note that, although *Speculative Persistence* keeps multiple cache versions of a block, only the latest version is persisted to the memory log when versions do not overflow. As such, write coalescing is enabled across transactions within a *speculation window*.

Commit Dependencies Between Transactions. Write coalescing for a block across different transactions causes new transaction dependencies that need to be resolved carefully at transaction commit time. This happens due to two reasons: 1) an aborted transaction may have overwritten a block in its preceding transactions, 2) an aborted transaction may have a block that is overwritten by succeeding transactions that have completed the update of their logs with the new version. To maintain the *out-of-order persistence, in-order commit* property of *Speculative Persistence*, we have to deal with the two problems when a transaction aborts: 1) how to rescue the preceding transactions that have overlapped writes with the aborted transaction?, and 2) how to abort the succeeding transactions that have completed the write of their logs for an overlapping write with the aborted transaction?

The two problems are solved by tracking the *commit issue order* of the transactions within each *speculation window* along with leveraging the multi-versioning support in the CPU cache. To solve the first problem, when an abort happens, preceding transactions that have overlapped writes with the aborted transaction write their versions of the blocks written by the aborted transaction from the CPU cache to the persistent memory log. To solve the second problem, we simply abort the transactions that come later in the commit issue order than the aborted transactions. In the recovery phase, a transaction is determined to be committed *only if* it is checked to be committed using the count-based commit protocol that checks *TxCnt* (as described in Section III-A) *and* its preceding transactions in its *speculation window* are committed.

Modifications to the Commit Protocol. In *Speculative Persistence*, overlapped writes in the same *speculation window* are merged, as described above. As a result, a transaction that has overlapped writes with succeeding transactions does not write the overlapped data blocks to the persistent memory log. This requires a modification to the commit protocol we described in Section III-A because, without modification of the commit protocol, the transaction might be mistaken as a not-committed transaction as the actual number of data blocks in the memory log area that are stored for it is different from the non-zero *TxCnt*. To differentiate between the two kinds of transactions (those with overlapped writes and non-overlapped writes), we add a new field of metadata, *Transaction (Tx) Dependency Pair*, in the memory log to represent the dependency between transactions with overlapped writes. *Tx Dependency Pair* $<T_x, T_y, n>$ represents that transaction T_x has n overlapped writes with its succeeding transaction T_y. Transaction T_x is determined to be committed if and only if T_y is committed and the actual number of log blocks of T_x plus n equals its non-zero *TxCnt*. As such, *Tx Dependency Pairs* help the commit status identification for transactions with coalesced writes.

C. Recovery from System Failure

Upon a system crash/failure, LOC scans the memory log area to recover the system to a consistent state. It first checks the start and end addresses of the valid logs, and then reads and processes the logs in the unit of *speculation window*. It processes each *speculation window* one by one, in program order. Since strict ordering is required between *speculation windows*, transactions across *speculation windows* have no dependencies. Each *speculation window* can thus be recovered independently.

First, the *META* block of each data block group is read in each *speculation window*,. LOC counts the number of *BLK-TAG*s for each $<CID, TID, TxID>$; this is the number of logged blocks of this transaction. If the number matches the non-zero *TxCnt* in any *BLK-TAG* of this transaction, the transaction is marked as committed.

Second, *Tx Dependency Pairs* from the memory log area are read. For each pair $<T_x, T_y, n>$, LOC adds the value n to T_x if T_y is committed. These pairs are checked in the reverse sequence, from tail to head. After this step, transactions that have overlapped writes are marked as committed using the commit protocol.

Third, the first not-committed transaction is found. All transactions after it are marked as not-committed. This guarantees the *in-order commit* property. After this, LOC finishes the committed transactions by writing the data blocks of these transactions to the home locations of the data blocks. Recovery completes after discarding all data blocks of the not-committed transactions from the log, and the system returns to a consistent state.

IV. Implementation and Hardware Overhead

We now describe the architecture implementation details and discuss the hardware overhead of LOC. Figure 4 shows the overview of LOC design. CPU issues load and store instructions to perform memory I/O operations. From the program's point of view, the volatile CPU cache and the persistent main memory are not differentiated; all stores to memory within a storage transaction are deemed to be persistent memory updates. In order to keep the storage system that resides in persistent memory consistent, both the I/O interface and the CPU cache hardware are extended to make sure data in volatile cache are persisted to persistent memory atomically.

A. Interface

The I/O interface, which lies between the CPU core and the CPU cache hardware, is extended with transactional commands: TxBegin, TxCommit, TxAbort and TxFlush. *TxBegin*, *TxCommit* and *TxAbort*

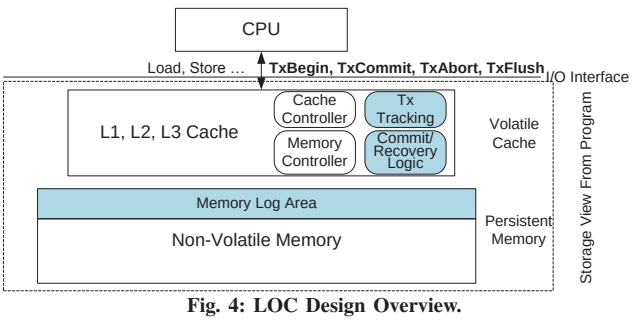
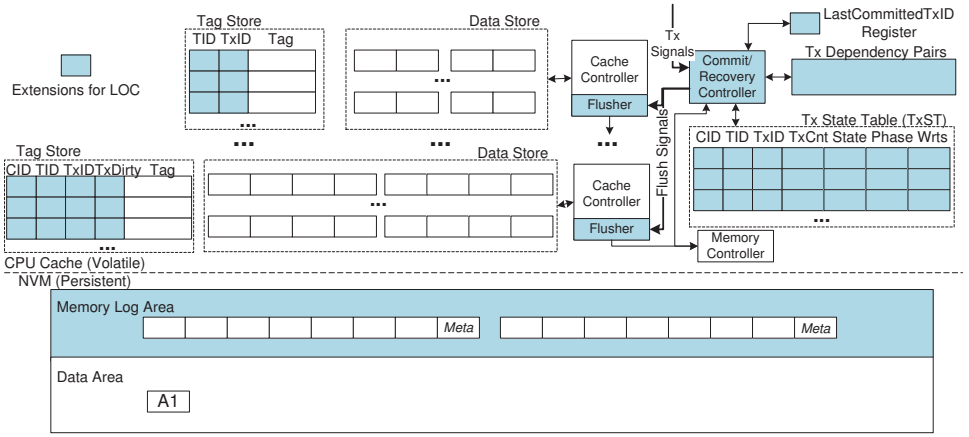


Fig. 4: LOC Design Overview.



(a) LOC Hardware Extensions

(Bits)	L1	L2	L3	TxST
CID	0	0	3	3
TID	1	1	1	1
TxID	8	8	8	8
TxDirty	0	0	1	0
TxCnt	0	0	0	16
State	0	0	0	2
Phase	0	0	0	2
Wrts	0	0	0	16
Total	9	9	12	48

(b) Storage Overhead of Hardware Extensions (extra bits per cache block or TxST entry)

Bit	State
0	invalid
1	active
2	committed
3	aborted

(c) Tx States

Bit	Phase
0	log write
1	in-place write
2	complete

(d) Tx Phases

Fig. 5: LOC Hardware Extensions and Their Storage Overhead.

are respectively used to start, commit and abort a transaction. *TxFlush* is used to explicitly write data to persistent memory from the CPU cache. In LOC, durability and atomicity are decoupled, similarly to the approaches of [26, 10, 34]. *TxCommit* and *TxFlush* are combined to provide both atomicity and durability in LOC.

B. Components

LOC adds three new components to the system: *Tx Tracking Component*, *Commit/Recovery Logic*, and *Memory Log Area*. Figure 5 shows where these components reside.

Tx Tracking Component has two parts: *Tx Dirty Block Table* and *Tx State Table*. *Tx Dirty Block Table* tracks the dirty blocks for each transaction at each level of the CPU cache hierarchy. It is supported with extra bits added to the tag store of each cache. As shown in Figure 5(a), each tag in all cache levels is extended with the hardware thread ID (*TID*) and the transaction ID (*TxID*). In the LLC, two other fields, the CPU core ID (*CID*) and the transaction dirty flag (*TxDirty*), are also added. *TxDirty* indicates whether the block has been written to the persistent memory log, and the original dirty flag indicates whether the block has been written to its home location. Transaction durability is achieved when log writes are persistent, i.e., *TxDirty* is unset. After that, home-location writes can be performed using the original dirty flag as in a conventional CPU cache. The storage overhead of each cache is illustrated in Figure 5(b). Only 9 bits (or 12 bits) are added for each 64B block in each level of cache (or LLC).

Tx State Table (TxST) tracks the status of active transactions, as shown in Figure 5. Each *TxST* entry has the *CID, TID, TxID, TxCnt, State, Phase and Wrts* fields. *State* denotes the transaction state, as shown in Figure 5(c). State transitions are induced by transactional commands. For instance, *TxBegin* changes the state from invalid to active; *TxCommit* (*TxAbort*) changes the state from active to committed (aborted). *Phase* denotes the current status of the write-back of the transaction: the transaction could be in the *log writing* phase, *in-place writing* phase updating home locations or could have completed the entire write-back, as shown on Figure 5(d). *Wrts* denotes the number of blocks written back in each phase, and is used to keep track of the completeness of *log writing* or *in-place writing* to determine the status of each transaction. The storage overhead is shown in Figure 5(b). Each *TxST* entry has 48 bits. For 128 transactions allowed in the system, the total size of *TxST* is 768 bytes.

Commit/Recovery Logic (CRL) receives transactional commands and manages the status of each transaction in the *Tx State Table*. CRL stalls new transactions until the current *speculation window* completes. For each *speculation window*, CRL tracks different versions of each data block and adds a 32KB volatile buffer to store its *Tx Dependency Pairs*. When a *speculation window* completes, the buffer is written back to the memory log area. In addition, CRL maintains a *LastCommittedTxID* register to keep the ID of the last committed transaction, which tells software that all transactions with smaller IDs are committed.

Memory Log Area is a contiguous physical memory space to log the writes from transactions. At the beginning of the memory log area, there is a log head, which records the start and end addresses of the valid data logs. The main body of memory log area consists of the log data block groups (as shown in Figure 2) and the metadata of *Tx Dependency Pairs* (introduced in Section III-B). In our evaluations, 32MB memory is allocated for the *Memory Log Area* as this was empirically found to be enough for the supported 128 transactions, but this space can be dynamically expanded.

C. Operations

A transaction writes data in three phases: execution, logging and checkpointing. In *execution* phase, data are written to the CPU cache. In this phase, transactional semantics are passed to the CPU cache with the extended transactional interface. Transactional writes are buffered in the CPU cache until the transaction commits. When a transaction commits, it enters the *logging* phase, in which data are persisted to the memory log area (i.e., log write). Only after all data are completely persisted to the log, a transaction can enter the *checkpointing* phase, in which data are written to their home locations in persistent memory (i.e., in-place write). Rather than keeping two copies (log write and in-place write) of each data block in the CPU cache, LOC stores only a single copy with an additional dirty flag (*TxDirty*). The *Tx Tracking Component* updates the *TxDirty* flag of each transactional write during different phases. The *Tx Tracking Component* allows the in-place writes only after all log writes in the transaction have completed. In the *logging* phase, when a transaction has all its data blocks persisted, the *Commit/Recovery Logic* checks the statuses of its previous transactions. If all its previous transactions have been committed, it also updates the *LastCommittedTxID* register to inform the software of the last committed transaction.

V. Evaluation

In this section, we first compare LOC with previous transaction protocols. Then, we analyze the performance gains of *Eager Commit* and *Speculative Persistence*. We also study sensitivity to memory latency.

A. Experimental Setup

We evaluate different transaction protocols using a full-system simulator, GEM5 [38]. GEM5 is configured using the syscall emulation (SE) mode. Benchmarks can directly run on the full system simulator without modification or recompilation. In the evaluation, GEM5 uses the *Timing Simple CPU* mode and the *Ruby memory system*. The CPU is 1 GHz, and the CPU cache and memory have the parameters shown in Table I. We revise both the cache and memory

controllers in GEM5 to simulate LOC, as shown in Figure 5. We faithfully model all overheads associated with LOC. In our evaluation, the *Speculation Degree* of LOC is set to 16 by default.

TABLE I: Simulator Configuration.

L1 Cache	32KB, 2-way associative, 64B block size, LRU replacement, block access latency = 1 cycle
L2 Cache	256KB, 8-way associative, 64B block size LRU replacement, block access latency = 8 cycles
LLC	1MB, 16-way associative, 64B block size LRU replacement, block access latency = 21 cycles
Memory	8 banks, memory access latency = 168 cycles

Workloads. Table II lists the workloads we evaluate. B+ tree is a widely used data structure in both file systems and database management systems. We implement a B+ tree, in which each 4KB node contains 200 key(8B)-value(4B) pairs. Each transaction consists of multiple key-value insert or delete operations. Similarly, we use hash table, red-black tree and random array swap data structures, also used in literature [13]. Our graph processing workload inserts and deletes edges in a large graph [39]. We also evaluate a database workload [40] on SQLite 3.7.17 [41].

TABLE II: Workloads.

Workloads	Description
B+ Tree	Insert/delete nodes in a B+ tree
Hash [13]	Insert/delete entries in a hash table
RBTree [13]	Insert/delete nodes in a red-black tree
SPS [13]	Random swaps of array entries
SDG [39]	Insert/delete edges in a large graph
SQLite [40]	Database benchmark on SQLite

B. Overall Performance

We measure and compare transaction throughput of five different transaction protocols: S-WAL, H-WAL, LOC-WAL, Kiln and LOC-Kiln. S-WAL is a software WAL protocol that manages logging and ordering in software [17]. H-WAL is a hardware WAL protocol that manages logging in hardware. Different from S-WAL, which writes two copies respectively for log and in-place writes in the CPU cache, H-WAL keeps only a single copy in the CPU cache and lets the hardware manage the log and in-place writes. H-WAL does not change the ordering behavior of S-WAL. LOC-WAL is our proposed protocol in this paper. Kiln is a recent protocol that uses non-volatile last-level cache to reduce the persistence overhead [32], as we described in Section II. Since Kiln's optimization is orthogonal to our LOC mechanism, we combine the two and also evaluate this combined version, called LOC-Kiln. LOC-Kiln achieves the best of both Kiln and LOC by only flushing L1 and L2 caches (as in Kiln) and performing loose ordering (as in LOC).

Figure 6 shows the normalized transaction throughput of the five protocols. The results are normalized to the transaction throughput of the baseline, which runs benchmarks without any transaction support and thus without the associated overheads of transactions. Note that this baseline does not provide consistency upon system failure and is intended to show the overhead of providing such consistency via different mechanisms. Transaction throughput is calculated by dividing the total number of committed transactions with the total runtime of each benchmark. We make two key observations.

(1) LOC significantly improves the performance of WAL, including both S-WAL and H-WAL. Normalized transaction throughput increases from 0.316 in S-WAL and 0.331 in H-WAL to 0.651 in LOC-WAL. In other words, LOC-WAL reduces ordering overhead from 68.4% in S-WAL and 66.9% in H-WAL to 34.9%. This is because S-WAL manages logging in software. The log writes and home-location writes have different memory addresses, and thus are independently cached in CPU cache. Keeping two copies in CPU cache hurts cache efficiency, which H-WAL removes, but this does not greatly reduce the overhead of WAL. LOC greatly reduces the overhead of WAL by removing intra-tx ordering using *Eager Commit* and loosening inter-tx ordering using *Speculative Persistence*. The loosened ordering improves cache efficiency and increases probability of write coalescing in the CPU cache.

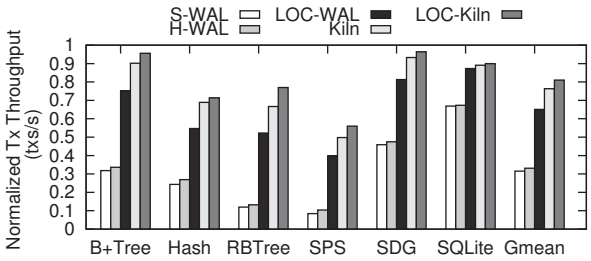


Fig. 6: Performance Comparison of Consistency Protocols.

(2) LOC and Kiln can be combined favorably. Doing so improves normalized transaction throughput to 0.811 on average, i.e., the ordering overhead is reduced to 18.9%. Kiln shortens the persistence path by employing an NV last-level cache. LOC mitigates performance degradation via a complementary technique, i.e., loosening the persistence ordering overhead that still exists in an NV cache.

We conclude that LOC effectively mitigates performance degradation from persistence ordering by relaxing both intra- and inter-transaction ordering.

C. Effect of the Eager Commit Protocol

We compare the transaction throughput of H-WAL and EC-WAL. EC-WAL is the LOC mechanism for WAL with only *Eager Commit* but without *Speculative Persistence*. Figure 7 plots the normalized transaction throughput of the two techniques. EC-WAL outperforms H-WAL by 6.4% on average. This is because the completeness check in *Eager Commit* is removed from the critical path of transaction commit. The elimination of intra-tx ordering leads to fewer cache flushes and improves cache efficiency.

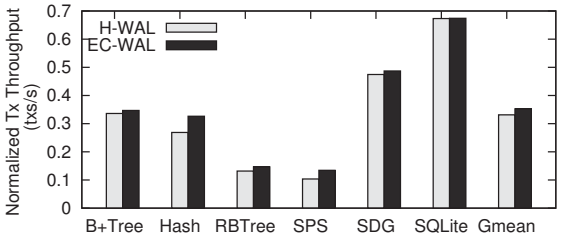


Fig. 7: Effect of Eager Commit on Transaction Throughput.

D. Effect of Speculative Persistence

To evaluate performance gains from *Speculative Persistence*, we vary the *speculation degree (SD)* from 1 to 32 (SD was set to 16 in previous evaluations). Figure 8 shows the normalized transaction throughput of LOC-WAL with different SD values. On average, the normalized transaction throughput of LOC-WAL increases from 0.353 to 0.689 with 95.5% improvement, going from SD=1 to SD=32. This benefit comes from two aspects of *Speculative Persistence*. First, *Speculative Persistence* allows out-of-order persistence of different transactions. A cache block without a cache conflict is not forced to be written back to persistent memory within a *speculation window* (as explained in Section III-B), reducing memory traffic and improving cache efficiency. Second, *Speculative Persistence* enables write coalescing across transactions within the *speculation window*, reducing memory traffic. Both of these effects increase as the *speculation degree* increases, leading to larger performance benefits with larger *speculation degrees*.

E. Sensitivity to Memory Latency

We evaluate LOC performance with different memory latencies to approximate the effect of different types of non-volatile memories. We vary memory latency from 35, 95, 168 and 1000 nanoseconds (our default evaluations so far were with a 168-nanosecond latency). We

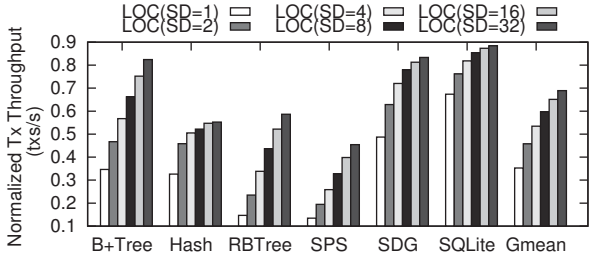


Fig. 8: Effect of Speculative Persistence on Transaction Throughput.

measure the transaction throughput of both H-WAL and LOC at each latency. Figure 9 shows the performance improvement of LOC over H-WAL at different memory latencies. In the figure, the black part of each stacked bar shows the normalized transaction throughput of H-WAL, and the gray part shows the performance improvement of LOC over H-WAL. Two major observations are in order. First, performance of H-WAL reduces as memory latency increases. This shows that higher memory latency in NVMs leads to higher persistence ordering overheads. Second, LOC's performance improvement increases as memory latency increases. This is because LOC is able to reduce the persistence overhead, which increases with memory latency. We conclude that persistence ordering overhead is becoming a more serious issue with higher-latency NVMs, which LOC can effectively mitigate.

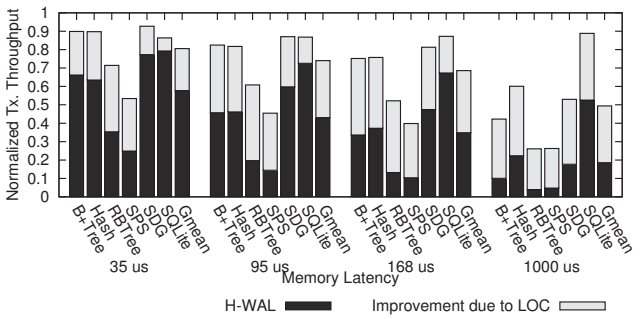


Fig. 9: Sensitivity to Memory Latency.

VI. Conclusion

Persistent memory provides disk-like data persistence at DRAM-like latencies, but requires memory writes to be written to persistent memory in a strict program order to maintain storage consistency. Enforcing such strict persistence ordering requires flushing dirty blocks from all levels of the volatile CPU caches and waiting for their completion at the persistent memory, which dramatically degrades system performance. To mitigate this performance overhead, we introduced *Loose-Ordering Consistency (LOC)*, which relaxes the persistence ordering requirement without compromising storage consistency. LOC's two key mechanisms, *Eager Commit* and *Speculative Persistence*, in combination, relax write ordering requirements both within a transaction and across multiple transactions. Our evaluations show that LOC can greatly improve system performance by reducing the ordering overhead across a wide variety of workloads. LOC also combines favorably with non-volatile CPU caches, providing performance benefits on top of systems that employ non-volatile last-level caches. We conclude that LOC can provide a high-performance consistency substrate for future persistent memory systems.

References

[1] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase change memory architecture and the quest for scalability," *CACM*, 2010.

[2] O. Mutlu, "Memory scaling: A systems architecture perspective," in *MEMCON*, 2013.

[3] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009.

[4] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009.

[5] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA*, 2009.

[6] B. C. Lee, P. Zhou, J. Yang, Y. Zhang *et al.*, "Phase-change technology and the future of main memory," *IEEE Micro*, 2010.

[7] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.

[8] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: a hybrid PRAM and DRAM main memory system," in *DAC*, 2009.

[9] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *ICCD*, 2012.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek *et al.*, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.

[11] X. Wu and A. L. N. Reddy, "SCMFS: a file system for storage class memory," in *SC*, 2011.

[12] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz *et al.*, "System software for persistent memory," in *EuroSys*, 2014.

[13] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp *et al.*, "NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS*, 2011.

[14] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," in *ASPLOS*, 2011.

[15] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *FAST*, 2011.

[16] J. Meza, Y. Luo, S. Khan, J. Zhao *et al.*, "A case for efficient hardware/software cooperative management of storage and memory," in *WEED*, 2013.

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

[18] J. Gray, P. McJones, M. Blasgen, B. Lindsay *et al.*, "The recovery manager of the System R database manager," *ACM Computing Surveys*, 1981.

[19] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *OSDI*, 2008.

[20] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *ICCD*, 2013.

[21] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From ARIES to MARS: Transaction support for next-generation solid-state drives," in *SOSP*, 2013.

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

[23] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *FAST*, 2013.

[24] Y. Lu, J. Shu, and W. Wang, "ReconFS: A reconstructable file system on flash storage," in *FAST*, 2014.

[25] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems," HPS Research Group, UT Austin, Tech. Rep., 2010.

[26] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *SOSP*, 2013.

[27] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture*, 2011.

[28] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. Osborne/McGraw-Hill, 2000.

[29] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993.

[30] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, 2010.

[31] Y. Lu, J. Shu, and P. Zhu, "TxCache: Transactional cache using byte-addressable non-volatile memories in SSDs," in *NVMSA*, 2014.

[32] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *MICRO*, 2013.

[33] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA*, 2014.

[34] I. Moraru, D. G. Andersen, M. Kaminsky, N. Binkert *et al.*, "Persistent, protected and cached: Building blocks for main memory data stores," PDL, CMU, Tech. Rep., 2011.

[35] D. Narayanan and O. Hodson, "Whole-system persistence," in *ASPLOS*, 2012.

[36] Intel, "Intel® 64 and IA-32 architectures software developers manual," 2014.

[37] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi, "Speculative versioning cache," in *HPCA*, 1998.

[38] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt *et al.*, "The Gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[39] "Graph: adjacency list," http://www.boost.org/doc/libs/1_52_0, 2013.

[40] "LevelDB benchmarks," http://leveldb.googlecode.com/svn/trunk/doc/benchmark.html.

[41] "Sqlite," http://www.sqlite.org.