MorLog: Morphable Hardware Logging for Atomic Persistence in Non-Volatile Main Memory

Xueliang Wei, Dan Feng*, Wei Tong*, Jingning Liu, Liuqing Ye
Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System,
Engineering Research Center of Data Storage Systems and Technology (School of Computer Science and Technology,
Huazhong University of Science and Technology), Ministry of Education of China
{xueliang_wei, dfeng, tongwei, jnliu, liuqingye}@hust.edu.cn

***Abstract*—Byte-addressable non-volatile memory (NVM) is emerging as an alternative for main memory. Non-volatile main memory (NVMM) systems are required to support atomic persistence and deal with the high overhead of programming NVM cells. To this end, recent studies propose hardware logging and data encoding designs for NVMM systems. However, prior hardware logging designs incur either extra ordering constraints or redundant log data. Moreover, existing data encoding designs are unaware of the characteristics of log data, resulting in writing unnecessary log bits.**

In this paper, we propose a morphable hardware logging design (MorLog) that only logs the data necessary for recovery and dynamically selects encoding methods with least write overhead. We observe that (1) only the oldest undo and the newest redo data in each transaction are necessary for recovery, and (2) the log data for clean bits are clean. The first motivates our morphable logging mechanism. This mechanism logs both undo and redo data for the first update to the data in a transaction, and then logs only redo data. Undo data are eagerly written to NVMM to ensure atomicity, while redo data are buffered in a volatile log buffer and L1 caches to write only the newest redo data to NVMM. The second motivates our selective log data encoding mechanism. This mechanism simultaneously encodes log data with different methods, and writes the encoded log data with the least write cost to NVMM. We devise a differential log data compression method to exploit the characteristics of log data. This method directly discards clean bits from log data and compresses remained dirty bits. Our evaluation shows that MorLog improves performance by 72.5%, reduces NVMM write traffic by 41.1%, and decreases NVMM write energy by 49.9% compared with the state-of-the-art design.

***Index Terms*—Non-volatile Memory, Persistence, Hardware Logging**

I. Introduction

Emerging byte-addressable non-volatile memory (NVM) technologies, such as phase change memory (PCM) [1], [31], [51], [59], resistive RAM (RRAM) [13], and 3D Xpoint [17], blur the boundary between memory and storage. NVM provides both DRAM-like access latency and disk-like data persistence. Due to these advantages, NVM is considered to be used as main memory, either to augment or replace DRAM. The non-volatile main memory (NVMM) allows applications to store important persistent data in memory through the fast (load/store) processor interface, without serializing data to the file system and executing expensive system calls.

Merely placing NVM on the memory bus will not suffice to reap its full potential. Designing NVMM systems faces many challenges. One of the most important challenges is supporting atomic persistence. The atomic persistence ensures that a group of data is persisted to NVMM in an all or nothing manner in the presence of system failures, such as power loss and system crashes. Without this guarantee, a system failure may corrupt data in NVMM. Another challenge is reducing NVMM write overhead. NVM suffers from high write latency and energy, especially for multi-level/triple-level cell (MLC/TLC) NVM that stores more than one data bit in a single cell [1], [45], [51]. Programming MLCs/TLCs requires precise control to adjust the cell resistance to one of many predetermined ranges. For this reason, an iterative program-and-verify technique [42], [45], [51] is usually used to program MLC/TLC NVM, which may increase write latency and energy by 10 times compared with single-level cell (SLC) NVM [42], [51].

Problem. To support atomic persistence, many studies propose to update data in NVMM through durable transactions with hardware logging [12], [22]–[24], [43], [53]. Hardware logging overlaps log operations with transaction execution, as long as log data are persisted before the data updated by transactions. However, existing hardware logging designs incur either extra ordering constraints or redundant log data, resulting in suboptimal performance and energy. Prior `undo` designs [24], [53] wait for persisting all the updated data when the transaction commits to guarantee persistence. Prior `redo` designs [12], [22], [23] forbid updating in-place data in NVMM until all the log data of a transaction are persisted. Prior `undo+redo` designs [43] relax the ordering constraints by logging both undo and redo data (i.e., the old and new values of updated data), but still conservatively force log data to NVMM before updated data for each write, which causes both unnecessary ordering constraints and log writes.

To reduce NVMM write overhead, data encoding techniques are proposed to transform the data written to NVMM, either to minimize the number of bit flips (i.e., ”1” → ”0” or ”0” → ”1”) [9], [11], [20], [21], [37], [44], [51], [52], [60] or using only the desirable states of MLCs/TLCs [32], [42], [45], [56], [61]. However, existing methods are suboptimal in encoding log data since they are unaware of the characteristics of log data. For example, data-comparison write (DCW) [62] is widely

[*] Corresponding author (Dan Feng and Wei Tong).

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00057

used in current encoding methods to avoid writing clean bits (i.e., ”0” → ”0” or ”1” → ”1”). If a persistent variable is updated but its value is not changed, the in-place data are not written since all the bits are clean. But, the log data for this update are still written to NVMM, because they may not have the same value as the old data in the location to be written.

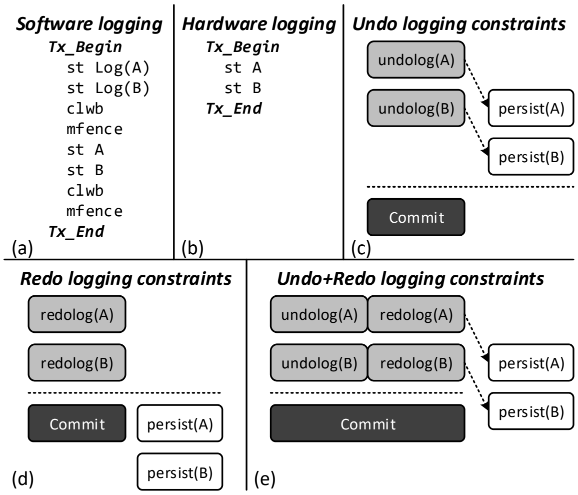
Our Approach. Our goal is to (1) mitigate logging overhead by avoiding unnecessary ordering constraints and log writes, and (2) reduce write overhead by exploiting the characteristics of log data. To mitigate logging overhead, we observe that logging both *undo and redo data* relaxes ordering constraints. Besides, for the updates to the same address in a transaction, only the *oldest undo* data and the *newest redo* data are necessary for recovery. To reduce write overhead, we observe that *log data for clean bits* are *clean* and do not need to be written to NVMM.

Based on the observations, we propose MorLog, a morphable hardware logging design that only logs necessary data and reduces the amount of clean log data in NVMM. MorLog consists of two mechanisms. First, the *morphable logging* mechanism creates undo+redo log entries for the first updates to the data in a transaction. The undo+redo entries are eagerly written to NVMM to ensure that they are persisted before the updated data. As in-memory data can be recovered with undo data, there are no ordering constraints between redo data and updated data. Thus, morphable logging lazily writes back redo data to coalesce the redo data for the same addresses. We observe that (1) 83.1% of data are updated more than once in a transaction, and (2) 44.8% of the write distance (i.e., the number of writes between two writes to the same address) is larger than 32. To coalesce redo data for these writes, morphable logging uses a redo buffer and L1 caches to buffer redo data. As more redo data are buffered, transaction commit may have longer latency to flush the buffered redo data. To reduce commit latency, we devise a *delay-persistence commit* protocol that allows transactions to instantly commit after execution.

Second, the *selective log data encoding* (SLDE) mechanism dynamically selects the encoding method with least write cost (e.g., latency or energy). We observe that 70.5% of bits among updated data are clean. To efficiently encode log data with many clean bits, we design a *differential log data compression* (DLDC) method. DLDC identifies dirty bits of log data by comparing the old and the new values of updated data. If all the bits are clean, DLDC directly discards the log data. Otherwise, DLDC first discards clean bits from log data, and then compresses dirty log data that follow predetermined data patterns. When encoding data, SLDE performs existing methods and DLDC in parallel. Only the encoded log data with the least write cost are written to NVMM.

In summary, our main contributions are:

- We propose MorLog, a hardware logging design that (1) avoids writing unnecessary log data, and (2) dynamically selects the encoding method for log data. To the best of our knowledge, MorLog is the first design that investigates the combination of hardware logging and data encoding.



Software logging

```
Tx_Begin
    st Log(A)
    st Log(B)
    clwb
    mfence
    st A
    st B
    clwb
    mfence
Tx_End
```

Hardware logging

```
Tx_Begin
    st A
    st B
Tx_End
```

Fig. 1. Example codes of (a) software and (b) hardware logging. Ordering constraints of hardware (c) undo, (d) redo, and (e) undo+redo logging.

- We devise a morphable logging mechanism to log only the oldest undo and the newest redo data for each transaction. Besides, we design a delay-persistence commit protocol that enables instant transaction commits.
- We devise a selective log data encoding mechanism that dynamically selects the encoding method for log data. Moreover, we design a differential log data compression method for encoding log data with many clean bits.
- Our evaluation shows that MorLog improves performance by 72.5%, reduces NVMM write traffic by 41.1%, and decreases NVMM write energy by 49.9% compared with the state-of-the-art [43].

II. Background and Motivation

A. Hardware Logging

Unexpected system failures may leave writes partially completed, corrupting persistent data structures in NVMM. Many studies propose to update data in NVMM through durable transactions with the write-ahead logging (WAL) technique [6], [7], [10], [25], [28], [34], [38], [54], [55]. The basic principle of WAL is to write data to a log region in NVMM before updating in-place data, so that in-place data can be recovered with log data after a system failure.

Why hardware logging. Log operations can be performed in software or hardware. Figure 1(a) shows that software logging approaches use `clwb` and `mfence` instructions [18] to guarantee the ordering constraints between the log data (e.g., `Log(A)`) and the updated data (e.g., `A`). As a result, log operations are performed in the critical path of transaction execution, causing performance degradation up to 70% [24]. Figure 1(b) shows that hardware logging approaches need only the annotations of the transaction boundaries in the codes. There is no explicit instruction for log operations. The hardware logs data for each write in a transaction and ensures the ordering constraints in the background. Thus, hardware logging significantly reduces logging overhead by overlapping log operations and transaction execution.

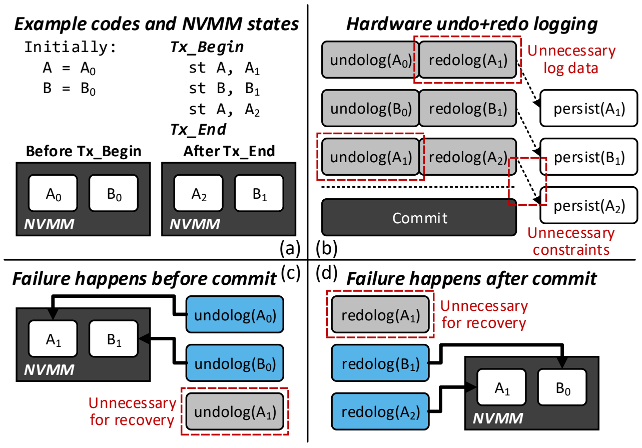


Fig. 2. (a) Example codes of a transaction and the correct data states in NVMM. (b) The log data and the ordering constraints of hardware undo+redo logging. (c) Recover the data in NVMM from the system failure that happens before commit. (d) Recover from the system failure that happens after commit.

Ordering constraints in hardware logging. Hardware logging approaches with different log types have different ordering constraints. As shown in Figure 1(c), hardware undo logging allows the in-place data in NVMM (e.g., `A`) to be updated after the corresponding undo data (e.g., `undolog(A)`) are persisted. If a system failure happens, we can undo the update to the in-place data with the undo data. However, to ensure persistence, transaction commit needs to wait for persisting the updated data. Otherwise, we cannot recover the latest transaction state after a system failure, because the updates made by the transaction may be lost.

Figure 1(d) shows that hardware redo logging can commit the transaction without persisting the updated data. After a system failure, we can recover the latest transaction state by updating the in-place data with the redo data. However, to ensure atomicity, the in-place data cannot be updated until all the redo data of the transaction are persisted. A mechanism is required to carefully manage the updates to the in-place data, which increases the complexity of the NVMM system design.

Hardware undo+redo logging combines the benefits of both undo and redo logging [43], because both undo and redo data are logged. As illustrated in Figure 1(e), the in-place data can be updated after the corresponding log data are persisted. Besides, transaction commit does not need to wait for updating in-place data. However, logging both undo and redo data increases the number of NVMM writes. It is critical to reduce the amount of undo and redo data.

B. Minimal Necessary Log Data for Recovery

As discussed in Section II-A, logging both undo and redo data has great potential for reducing logging overhead. To log undo and redo data, existing hardware logging designs create an undo+redo log entry for each write in transactions [43]. Each undo+redo log entry is persisted before the corresponding updated data. Figure 2(a) and (b) show an example of the hardware logging design. The data in NVMM are A_0 and B_0 before the transaction begins (i.e., `Tx_Begin`), while the data are updated to A_2 and B_1 after commit (i.e., `Tx_End`). After a system failure happens, we need to recover the data to one of the states shown in Figure 2(a). If a failure happens before commit, as shown in Figure 2(c), the in-place data are rolled back with the undo data undolog(A_0) and undolog(B_0). On the contrary, as illustrated in Figure 2(d), if a failure happens after commit, the in-place data are rolled forward with the redo data redolog(A_2) and redolog(B_1). Therefore, the intermediate log data undolog(A_1) and redolog(A_1) are unnecessary for recovery. Furthermore, the ordering constraints between the unnecessary undo data and the corresponding updated data are also unnecessary.

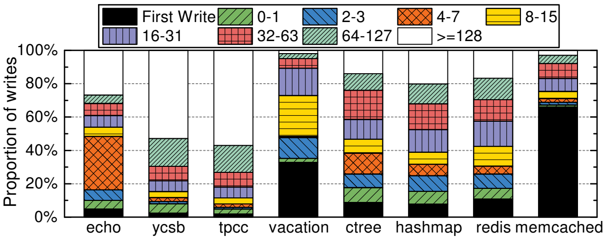


Fig. 3. Distribution of write distance for writes in transactions. The write distance refers to the number of writes between two writes with the same address. The `First Write` indicates the first write to an address.

CONSEQUENCE 1. *Logging both undo and redo data can relax the ordering constraints, but only the oldest undo data and the newest redo data in each transaction are necessary for recovery. Besides, the ordering constraints of unnecessary log data are also unnecessary.*

Figure 2 shows that unnecessary log data are written to NVMM if in-place data are updated more than once in a transaction. To reduce unnecessary log data, existing undo+redo designs add a log buffer (a volatile FIFO) in the memory controller [43]. The log data for the same in-place data are coalesced in the log buffer, so that the intermediate log data are discarded. However, these designs limit the log buffer size. If the log buffer has `N` entries, log data will be evicted `N` cycles after they are added to the log buffer. To ensure that undo data are persisted before the corresponding updated data, `N` is conservatively set smaller than the minimum latency of traversing the cache hierarchy. Therefore, if there are more than `N` writes between the two writes that update the same address in a transaction, the log data of the two writes cannot be coalesced in the log buffer.

We use *write distance* to refer to the number of writes between the two writes that update the same address. To quantify the distance of writes in transactions, we run a set of WHISPER [39] applications on a Linux server (kernel version 4.15.0-43) with an Intel Xeon E5620 2.4GHz CPU and 12GB DRAM (of which we use 4GB as NVMM), and use the PIN [36] tool to monitor the writes in transactions. Figure 3 illustrates the distribution of write distance. The `First Write` indicates the first write to an address. The results show that the distance of 44.8% of writes is larger than 31, on average. Therefore, many unnecessary log data cannot be discarded in the log buffer with limited size. This motivates our morphable logging mechanism (§ III).

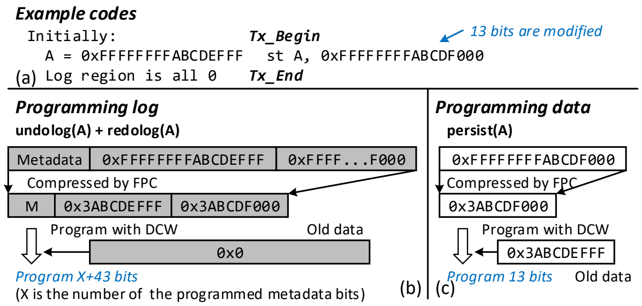


Fig. 4. (a) Example codes of a transaction. (b) Programming log data with 64-bit FPC [45] and DCW [62]. (c) Programming updated data.

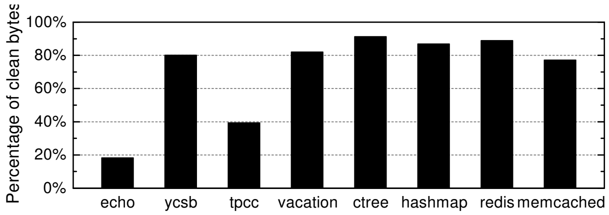


Fig. 5. Percentage of clean bytes among the data updated by transactions.

C. Data Encoding for NVMM

Data encoding techniques [9], [11], [20], [21], [32], [37], [42], [44], [45], [51], [52], [56], [60], [61] are proposed to alleviate the high write latency and energy problems of NVMM. However, existing encoding methods are suboptimal in encoding log data. Figure 4 shows an example of encoding the data written to NVMM. Note that we do not show the compression tags for simplicity. For the transaction in Figure 4(a), the undo+redo log entry of A is created and written to NVMM. Figure 4(b) shows that the log entry is first compressed by the 64-bit frequent pattern compression (FPC) [45] method. Assume that log metadata and log data (i.e., undo or redo data) are compressed separately. This can be realized by packing the metadata from different log entries into one write with the log packing optimization [22]. After compression, the log entry is written to NVMM with DCW [62]. DCW compares the old data and the new data, and programs only the flipped bits. As a result, X+43 bits are programmed, where X is the number of bit flips in the metadata. However, Figure 4(c) shows that only 13 bits are programmed to update A. Thus, the log entry contains many clean bits of A. These clean log bits are unnecessary for recovery since the corresponding bits of A are unprogrammed.

CONSEQUENCE 2. *The log data for clean updated data are also clean. The clean log data are unnecessary for recovery, and should be discarded to improve write latency and energy.*

We quantify the amount of clean updated data by comparing the old and the new value for each write in transactions. Figure 5 shows that 70.5% of bytes among the data updated by transactions are clean, on average. Logging such a large amount of clean data imposes significant write overhead. This motivates our selective log data encoding mechanism (§ IV).

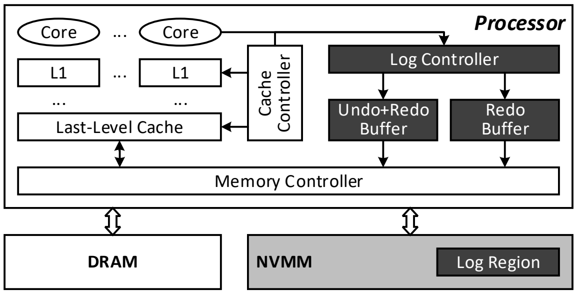


Fig. 6. Architecture overview of MorLog.

III. Morphable Logging

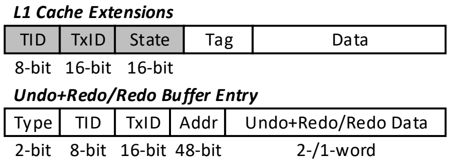
To mitigate logging and NVMM write overhead, we propose two novelties in MorLog: *morphable logging* and *selective log data encoding* (SLDE). We now describe the design and implementation details of morphable logging.

A. Assumptions and Architecture Overview

Failure model. We assume that both DRAM and NVMM are placed on the memory bus, and mapped to a single physical address space. DRAM is used to store the data that do not require persistence, while the user-defined critical data are stored in NVMM (similar to [22], [43], [65]). MorLog ensures the atomic persistence of updates to the data in NVMM. To reduce persist latency, we adopt the ADR [16] optimization that allows the data in the memory controller to be flushed to NVMM on a power loss.

Transaction-based atomic persistence. We assume that atomic persistence is based on durable transactions with hardware logging (similar to [12], [22]–[24], [43], [53]). Programmers annotate the transaction boundaries through Tx_Begin and Tx_End interfaces [22]. We rely on software mechanisms, such as fine-grained locking [6], to ensure isolation between conflict transactions. Moreover, nested transactions are not supported and left for future work since they are orthogonal to atomic persistence [22]. Our current design guarantees the outer most atomic persistence for nested transactions.

Overview. Figure 6 shows the architecture overview of MorLog. We adopt two log buffers (volatile FIFOs), an undo+redo buffer and a redo buffer. When data are updated in a transaction for the first time, MorLog creates an undo+redo log entry for this update at 64-bit word granularity, and sends the entry to the undo+redo buffer. Before the entry is written back to NVMM, MorLog still creates undo+redo log entries for the subsequent updates to the same data, so that these entries can be coalesced in the undo+redo buffer. Otherwise, MorLog sends redo log entries to the redo buffer for the subsequent updates. As the undo data have been persisted, the updated data can be written to NVMM before the redo log entries. In this case, MorLog directly discards the corresponding entries from the redo buffer to avoid writing unnecessary redo data. Note that, to avoid a cache coherence issue, both undo+redo and redo buffers directly send log entries to the memory controller by bypassing the caches. This allows log operations to be performed without affecting the data in the caches.



L1 Cache Extensions

TID	TxID	State	Tag	Data
8-bit	16-bit	16-bit		

Undo+Redo/Redo Buffer Entry

Type	TID	TxID	Addr	Undo+Redo/Redo Data
2-bit	8-bit	16-bit	48-bit	2-/1-word

Fig. 7. Hardware extensions and data structures for morphable logging.

The log region is used to store log entries. MorLog organizes the log region as a single-consumer, single-producer Lamport circular structure [30], so that the log can be simultaneously appended and truncated without locking. To maintain the log, we employ two 64-bit registers to store log head and tail pointers (similar to [43]). In the case that the write set of an in-flight transaction exceeds the log region size, overflow may happen. The overflow can be prevented by (1) allocating a large-enough log region or (2) allocating a temporary region when the current one is filled by an in-flight transaction [43].

Figure 6 shows the architecture with only one processor, but the design can also be extended to support multi-processor systems. Specifically, in a multi-processor system, the new components (e.g., the log buffers and the log controller) are added in each processor to perform log operations. Moreover, a log region can be allocated in the local NVMM of each processor to reduce the latency of log writes.

B. Eager Undo Lazy Redo Writeback

Maintaining log data. Morphable logging adopts an *eager undo lazy redo* writeback policy to maintain the buffered log entries. Before going to this policy, we first assume that there is only an undo+redo buffer. As the distance is large for many writes (§II-B), we need a large undo+redo buffer to coalesce log data. Ideally, if the undo+redo buffer is large enough to buffer all the undo and redo data of a transaction, we can easily find the necessary log data, and only write them to NVMM. However, a large buffer not only causes substantial hardware overhead, but also violates the ordering constraints between undo data and updated data.

To solve these problems, the eager undo lazy redo writeback policy utilizes an undo+redo buffer, a redo buffer, and the L1 caches to coalesce log data. When an L1 cache line is updated in a transaction for the first time, an undo+redo entry is created and sent to the undo+redo buffer. The undo data are gotten from the L1 cache line, while the redo data are gotten from the current update operation itself. Due to the ordering constraints between undo data and updated data, the undo+redo buffer *eagerly* writes the entry to NVMM after `N` cycles, where `N` is smaller than the minimum latency of traversing the cache hierarchy. Before the undo+redo entry is evicted, we still create undo+redo entries for the subsequent updates to the same L1 cache line in the same transaction. This allows log data for the write with a small distance to be coalesced in the undo+redo buffer. After the undo+redo entry is persisted, we store the redo data of the subsequent updates in place in the L1 cache line. The redo entry is created when the L1 cache line is evicted. The redo buffer *lazily* writes the redo entry to NVMM. Before the redo entry is evicted, if the corresponding cache line is evicted by the LLC or updated again by the same transaction, the entry is directly discarded from the redo buffer. When the transaction commits, the log data in the two buffers and the L1 caches are persisted. However, waiting for persisting log data may lead to long commit latency. We provide an optional commit protocol that enables instant transaction commit in Section III-C.

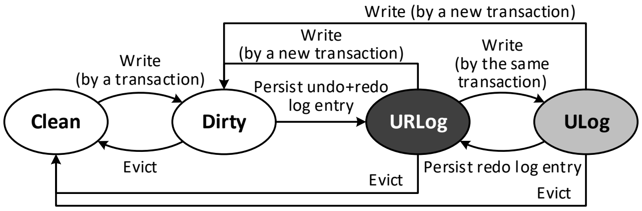

Write (by a new transaction)
Write (by a transaction)
Write (by a new transaction)
Write (by the same transaction)
Clean
Dirty
Persist undo+redo log entry
URLog
ULog
Evict
Evict
Persist redo log entry
Evict

Fig. 8. State transition of L1 cache lines.

Hardware extensions. Figure 7 shows the structures of the log entry and the extensions of the L1 caches. All the buffer entries contain a 2-bit entry type flag (`Type`), an 8-bit thread ID (`TID`), a 16-bit transaction ID (`TxID`), a 48-bit physical address of the in-place data (`Addr`), and a 1-word redo data. The undo+redo entry also includes a 1-word undo data. For each log entry in NVMM, we add a 1-bit torn bit to indicate whether all the entries of a transaction are written [55]. The value of torn bits are the same in one pass of the log region, and reversed in the next pass. If the torn bits of log entries are the same as the commit record, all the entries are in the log region. In each L1 cache line, we add an 8-bit `TID`, a 16-bit `TxID`, and a 16-bit log state flag (`State`). Each 2 bits of `State` indicates the state of one 64-bit word in the cache line. A word can be in one of the four states. The `Clean` state means that the word has not been updated by a transaction. The `Dirty` state means that the word has been updated by an in-flight transaction. The `URLog` state means that the undo+redo entry of the word has been persisted. The `ULog` state means that the oldest undo data have been persisted but the newest redo data are not. Figure 8 summarizes the state transition triggered by various events. Note that the state changes from `URLog` to `ULog` only when the word is updated again by the same transaction that causes the `Clean` → `Dirty` transition.

C. Reducing Commit Latency

Instant transaction commit. We provide an optional *delay-persistence commit* protocol to reduce commit latency. This protocol makes the tradeoff between performance and persistence. Only the atomicity of transactions is guaranteed after commit, while the persistence is not. Our design can naturally guarantee atomicity. This is because the eager undo lazy redo writeback policy ensures that undo data are persisted before updating in-place data. Furthermore, undo data are persisted in the order that the corresponding update operations are issued, since the undo+redo buffer is a FIFO. No matter when a system failure happens, in-place data can be rolled back with undo data. Therefore, this protocol allows a transaction to instantly commit after its execution is completed.

TABLE I
SUMMARY OF MAJOR HARDWARE OVERHEAD OF MORPHABLE LOGGING.

Component	Type	Size
Log head and tail registers	flip-flops	16 bytes
L1 cache extensions	SRAM	40 bits per cache line
Undo+Redo buffer	SRAM	404 bytes
Redo buffer	SRAM	552 bytes
Ulog counters (Optional)	flip-flops	20 bytes

Reducing data loss. The prior designs also support instant transaction commit [43]. Differently, we need to carefully determine when to persist the redo data buffered in the L1 cache. If we eagerly send the redo data to the redo buffer during commit, the transaction commit cannot be completed instantly. On the contrary, if we lazily write the redo data to NVMM through cache evictions, the updates made by many transactions may be lost after system failures.

To enable instant commit and reduce data loss, our commit protocol creates the redo entries when the L1 cache lines in ULog state are updated by a new transaction or evicted from the L1 cache. To determine whether all the L1 redo data of a transaction are persisted, we maintain a 10-bit *ulog counter* per hardware thread. The ulog counter indicates the number of L1 cache lines in ULog state for the in-flight transaction on the hardware thread. When the transaction commits, we log the ulog counter by adding it in the commit record. Note that the transaction commit does not need to wait for logging the counter. If the counter is equal to the number of the redo entries that are created after commit, all the redo data of the transaction are in the log region.

D. Hardware Overhead

Table I summarizes the hardware overhead of morphable logging in the processor. These values may be different in other system configurations. The size of the log head or tail register is 8 bytes in a 64-bit machine, while it is 4 bytes in a 32-bit machine. The size of the L1 cache extensions can be reduced by using a smaller TID and TxID. But, doing so limits the number of transactions that can be logged in NVMM. The undo+redo and redo buffer sizes depend on the total number of buffer entries. We adopt a 16-entry undo+redo buffer and a 32-entry redo buffer to make a balance between performance and hardware cost. The ulog counters are only used in the delay-persistence commit protocol, thus are unnecessary for the system without this protocol.

E. Recovery

If the delay-persistence commit is disabled, transaction commit ensures both atomicity and persistence. After a system failure, we recover the persistent data in NVMM by following the same steps as the prior undo+redo hardware logging [43]. That is, the recovery routine first scans the log region from the head to the tail, and then uses the commit record and the torn bits to determine whether a transaction has committed. Then, the updates made by committed transactions are redone with the redo data, while the updates made by non-committed transactions are undone with the undo data. To undo and redo the updates, the corresponding log data are copied to the home locations (indicated by Addr) in NVMM. After that, log entries are deleted by updating the log head pointer.

If the delay-persistence commit is enabled, transaction commit only ensures atomicity. We use the ulog counter in the commit record to determine the persistence of a transaction. If the ulog counter is equal to the number of redo entries created after commit, the transaction has been persisted before the system failure occurs. The recovery routine scans the log region from the head to the tail, and copies the redo data to the home locations for the persisted transactions. When the recovery routine encounters a non-persisted transaction, all the transactions that commit after this one are also considered non-persisted, since transactions must be persisted in the commit order to ensure correctness. For the non-persisted transactions, the undo data are copied to the home locations.

F. Discussion

Log types. Logs in NVMM can be centralized [22], [24] or distributed (e.g., per-thread) [15], [57]. Centralized logs simplify the maintenance of inter-thread transaction dependencies, while distributed logs provide better scalability in large systems. Our design works with either log type. With centralized logs, each log entry needs to maintain a TID, but the commit order of transactions from different threads can be easily determined by the locations of commit records. With distributed logs, the TID is no longer necessary in each log entry. Instead, a timestamp is added in each commit record to indicate the commit order. During recovery, we need to use the timestamps to ensure that transactions are persisted in the commit order if the delay-persistence commit is enabled. As we show in the experiments, our design with centralized logs effectively improves performance across various workloads. We leave the evaluation of the design with distributed logs as our future work.

Log management. The log entries of a committed transaction cannot be deleted until all the updated data are persisted. We provide two options to delete log entries. First, we adopt the force-write-back [43] mechanism. This mechanism adds an additional flag bit in each cache line. The cache controller periodically scans cache lines. For each dirty cache line, if the flag bit is unset, it is set by the cache controller. Otherwise, the cache controller writes back the cache line, but does not invalidate it (similar to clwb [18]). When the scan is completed, we update the log head to delete the log entries of the transactions that commit before the last two scans. Second, we use a variation of the transaction table [22]. Each table entry consists of a TID, a TxID, addresses of the first and last log entries, and a counter. The counter indicates the number of cache lines that contain the updated data. If the counter is equal to 0, the corresponding log entries can be deleted from the log region. The first option is simpler and has less hardware cost, while the second one provides more flexibility.

ADR-based optimizations. Asynchronous DRAM Refresh (ADR) supported by Intel [16] allows the memory controller to be considered part of the persistence domain, which provides many optimization opportunities for hardware logging. The ADR-based optimizations can be used in MorLog to further reduce log writes. For example, a log pending queue (LPQ) [53] is added in the memory controller to buffer log writes. The log writes of committed transactions can be directly discarded from the LPQ without writing to NVMM when the corresponding updated data are persisted. Note that MorLog with the ADR-based optimizations still benefits from morphable logging. As the LPQ has only a limited number of entries, many log writes are still sent to NVMM during transaction execution. Morphable logging reduces the number of log writes to NVMM by avoiding logging unnecessary log data.

Dealing with non-temporal stores. The persistent data may be updated by non-temporal stores (e.g., `movntq` [18]) that bypass the caches. Non-temporal stores can form whole cache lines by themselves. In this case, creating undo+redo entries needs to wait for reading undo data from NVMM. Therefore, we only log redo data for the non-temporal stores. To ensure atomicity, we utilize a small region of DRAM to cache the non-temporal stores, and write them to NVMM after the transaction commits. The DRAM cache is managed in a similar way to the prior design [22].

Applications with extreme locality. In some special cases, applications may have extremely low or high temporal locality. For example, each transaction of an application either never updates the same data more than once, or only updates one cache line many times. In this case, log data cannot be coalesced in the redo buffer and the L1 caches, or the undo+redo buffer is enough to coalesce log data. Thus, we can use only the undo+redo buffer for these applications to provide both short commit latency and persistence. We leave the exploration of how to dynamically adjust the place for buffering log data as future work.

IV. Selective Log Data Encoding

A. Differential Log Data Compression

Prior encoding methods for NVMM do not distinguish log data and updated data, resulting in storing many clean log data. To encode only dirty log data, we design a *differential log data compression* (DLDC) method for SLDE.

Getting dirtiness of log data. DLDC introduces a *dirty flag* in each log buffer entry to indicate which bytes in the undo or redo data are dirty. As the L1 caches also buffer redo data, we add a dirty flag for each 64-bit word in each L1 cache line. We maintain the dirty flag by comparing the old and the new value of each write in a transaction. The old value is gotten from the L1 cache line, while the new value is gotten from the in-flight write itself. Reading the old value does not cause extra latency, since it can be overlapped with tag matching of the in-flight write [33]. The flag bit is set if the corresponding bytes are dirty. In the case that all the flag bits are unset, the log data are completely clean. The two log buffers directly discard the log entries in which the log data are completely clean. We use *silent log writes* to refer to the writes for these log entries. Silent log writes are unnecessary for recovery as discussed in Section II-C. Note that we consider all the log data are dirty for non-temporal stores to avoid waiting for reading old values from NVMM.

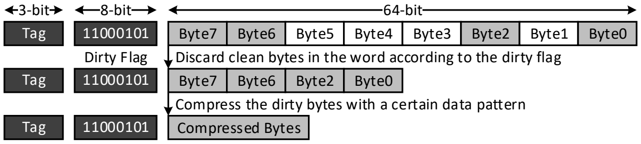


Fig. 9. Encoding process of DLDC.

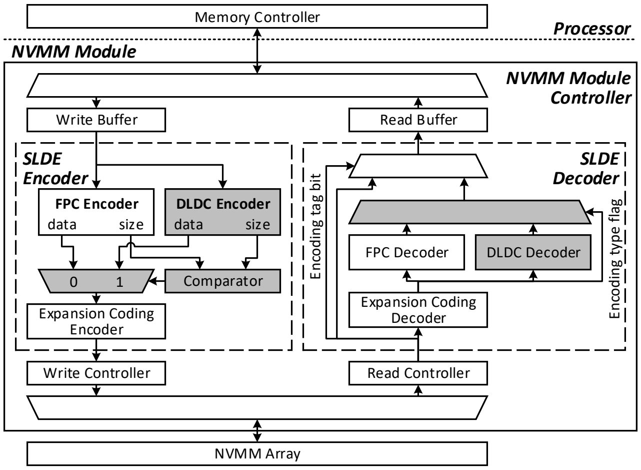


Fig. 10. Example of SLDE codec architecture that uses CRADE [61] as an alternative encoding method.

Encoding log data. DLDC encodes the undo or redo data in two steps. Figure 9 shows an example of the encoding process. In this example, each flag bit is associated with one byte of log data. DLDC first discards the clean bytes according to the dirty flag, and then compresses the remained dirty bytes if they match a certain data pattern. The compressed bytes are stored along with a 3-bit pattern tag that indicates the data pattern. Table II lists the data patterns. We quantify the percentage of dirty log data compressed by each pattern by monitoring the execution of WHISPER [39] applications with the PIN [36] tool. The results averaged over all the applications are shown in the last column of Table II. Cumulatively, DLDC can compress about 42.5% of dirty log data.

Log data are read and decoded during recovery. DLDC decodes log data by decompressing the compressed bytes according to the pattern tags. After that, the dirty log data are copied to the home location to recover the in-place data. The dirty flags indicate which bytes of the in-place data need to be written with the dirty log data.

B. SLDE Codec Architecture

Due to the extra dirty flags, DLDC may increase the size of log entries when most bytes are dirty. Thus, SLDE adopts an alternative method to encode the data that are not suitable for DLDC. Figure 10 shows an example of SLDE codec architecture that uses CRADE [61] as the alternative encoding method. CRADE first compresses the data with FPC [45], and then expands the compressed data with the best-performing

TABLE II

DATA PATTERNS THAT CAN BE COMPRESSED BY DLDC. THE 3-BIT PATTERN TAG IS INDICATED IN RED. N IS THE DATA SIZE BEFORE COMPRESSION. THE LAST COLUMN SHOWS THE PERCENTAGE OF DIRTY LOG DATA THAT CAN BE COMPRESSED WITH THE GIVEN PATTERN.

Tag	Data Pattern	Example	Compressed Example	Size Reduction	Percentage
000	All zero	`0x00000000`	`0x0`	$(N-3)$ bits	9.3%
001	2-bit sign-extended in each byte	`0x01F20101`	`0x165`	$(3N/4-3)$ bits	4.5%
010	4-bit sign-extended in each byte	`0x03F905FE`	`0x2395E`	$(N/2-3)$ bits	5.9%
011	1-byte sign-extended	`0xFFFFFF80`	`0x380`	$(N-11)$ bits	4.4%
100	2-byte sign-extended	`0x00007FFF`	`0x47FFF`	$(N-19)$ bits	1.4%
101	4-byte sign-extended	`0xFF80000000`	`0x580000000`	$(N-35)$ bits	3.8%
110	4-bit padded with a zero 4-bit in each byte	`0x10203040`	`0x61234`	$(N/2-3)$ bits	10.4%
111	1 least significant byte is zero	`0x1234567800`	`0x712345678`	5 bits	2.8%

IDM [42] according to the compression ratio. The compression ratio is equal to p/q, where p is the size of data before compression, and q is the size of compressed data including the corresponding tags. The SLDE encoder is on the write path of the NVM module controller between the memory bus and the NVMM array, while the decoder is on the read path.

Write. When the NVMM module receives a write from the memory controller, SLDE encodes the data as follows. First, the data are compressed by the FPC encoder and the DLDC encoder in parallel. DLDC compresses only log data, and directly outputs other data (e.g., updated data or log metadata). We utilize the write address and a metadata bit to distinguish between different data types. If the write address is within the log region, the data are either log metadata or log data, since metadata and data are separated after log packing [22]. The data are the log metadata if the metadata bit is set, the log data otherwise. Second, the sizes of data compressed by FPC and DLDC are compared to determine which compression method is better. The data from the DLDC encoder are used only if the data are log data and their size is smaller. Third, the compressed data are expanded according to the compression ratio. To distinguish between the log data encoded with different methods, we add a 3- and 2-bit encoding type flag in each undo+redo log entry and each redo log entry, respectively. Note that the undo and redo data of one entry are not compressed with DLDC at same time to avoid losing undo data.

Read. When the NVMM module receives a read, SLDE decodes the data before forwarding them to the memory controller. If the encoding tag bit stored along with the data shows that the data are not encoded, SLDE directly sends the original data to the read buffer. Otherwise, SLDE either chooses the appropriate method to decode the log data according to the encoding type flag, or uses FPC to decompress the data with other types.

C. Hardware Overhead

Capacity overhead. A dirty flag is added in each log buffer entry and L1 cache line. The size of each dirty flag is determined by the number of bytes associated with one flag bit. Assume that n is the size of undo or redo data, while m is the number of bytes associated with one flag bit. The size of the dirty flag is n/m. Therefore, for the undo+redo entry, the redo entry, and the L1 cache line, the capacity overheads of the dirty flag are $\frac{4}{101m}$, $\frac{4}{69m}$, and $\frac{1}{8m}$, respectively. Furthermore, a metadata bit is stored along with each 64-byte block in the log region, while an encoding type flag is added in each log entry. The overhead of these flag bits is $\leq 1/512 + \max(3/202, 2/138) = 1.7\%$.

Latency and logic overhead. SLDE incurs extra latency and logic overhead over the existing encoding method that is used as the alternative. To estimate the overhead, we implement the SLDE encoder and decoder in Verilog hardware description language. Then, we synthesize the logic in Synopsys Design Compiler and scale the results down to 22nm technology node. The extra logic overhead of SLDE is $\approx 4.2K$ gates, which is less than 0.1% per NVMM module. The extra encoding and decoding latencies are smaller than 1ns, which is negligible compared with the NVMM access latency. In addition, the extra energy consumptions of encoding and decoding are $1.4pJ$ and $1.3pJ$, respectively. As the averaged write energy of a TLC RRAM cell is $16.0pJ$ [42], [45], [61], the extra energy consumption of SLDE is less than 0.1% of the write energy of a 64-byte data block.

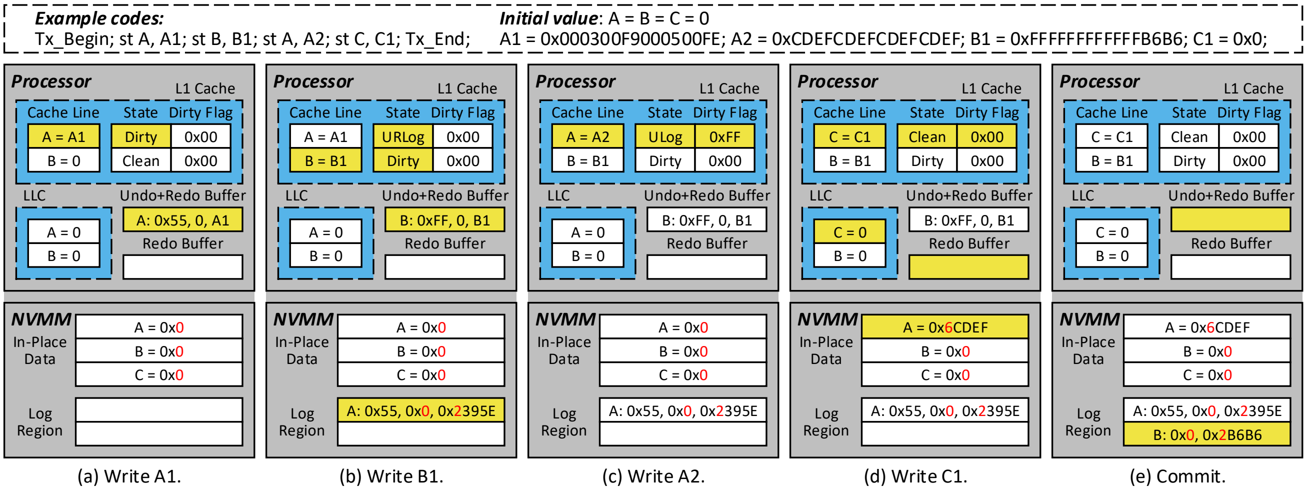
D. System with Secure NVMM

Some systems encrypt the data in NVMM to enhance security [4], [63], [66]. All good encryption algorithms have the diffusion property (also called Avalanche Effect) [3], [64]. That is, a change of even one bit in the plaintext causes a larger number of bits in the ciphertext to be changed. Therefore, encryption increases the percentage of dirty bits written to NVMM. To solve this problem, some new techniques [3], [64], [67] have been proposed to keep writes clean during encryption. For example, DEUCE [64] re-encrypts only the dirty words when writing data to NVMM, so that the other words remain clean after encryption. With these techniques, SLDE can still effectively improve performance in the system with secure NVMM by avoiding logging clean bits.

V. PUTTING IT TOGETHER

Figure 11 shows how morphable logging and SLDE work together in MorLog. In this example, each dirty flag bit indicates the dirtiness of one data byte. For simplicity, we assume that each cache line contains only one 8-byte word. The transaction execution with 64-byte cache lines has a similar flow, since each word of a cache line is also associated with a 2-bit log state flag and an 8-bit dirty flag.

Write `A1`. The transaction writes `A1` to the cache line *A*. As the old and the new value are different, MorLog changes the



(a) Write A1. (b) Write B1. (c) Write A2. (d) Write C1. (e) Commit.

Fig. 11. Example of the execution flow of a transaction.

log state to Dirty, and creates an undo+redo entry for the updated word.

Write B1. The transaction writes B1 to the cache line *B*. MorLog evicts the existing undo+redo entry to NVMM since the buffer is full. After the evicted entry is persisted, the log state of the cache line *A* is changed to URLog. When writing the evicted entry to NVMM, the undo and redo data are encoded with SLDE. The undo data (i.e., 0) are compressed by FPC, while the redo data (i.e., A1) are compressed by DLDC. As DLDC is used for encoding, the dirty flag (i.e., 0x55) is also stored in the log entry.

Write A2. The transaction writes A2 to the cache line *A*. MorLog changes the log state to ULog, and buffers the redo data in the cache line *A* in place. To store the information about which bytes of the redo data are dirty, the dirty flag is set by comparing the values of A1 and A2.

Write C1. The transaction writes C1 to the cache line *C*, but the state is still Clean since the value is not changed. When evicting the cache line *A* from the L1 cache, MorLog creates a redo entry for the word in ULog state. Before the entry is evicted from the redo buffer, the cache line *A* is evicted from the LLC. In this case, MorLog directly discards the corresponding entry from the redo buffer. Furthermore, MorLog chooses FPC to compress the data when writing the cache line *A* to NVMM, because they are not log data.

Commit. When the transaction commits, MorLog persists all the corresponding log data to ensure persistence. In the case that the system can tolerate some data loss, MorLog can avoid persisting log data in the critical path by using the delay-persistence commit protocol (§III-C).

VI. Evaluation

A. Methodology

We implement MorLog on the Gem5 [5] simulator to evaluate its efficiency. Gem5 is configured to model a multi-core processor. The undo+redo and redo buffer sizes are 16 and 32 by default, respectively. We also evaluate the performance variation on different buffer sizes in Section VI-E. We perform the force-write-back mechanism [43] every three million cycles to write back updated data from the caches, in line with the prior studies [43]. To simulate the NVMM system, we model a TLC RRAM based main memory system by integrating Gem5 with NVMain [49], a cycle-accurate memory simulator for NVMs. The timing and energy parameters of the TLC RRAM are the same as the previous researches [42], [45], [61]. We extend NVMain to implement SLDE. Each dirty flag bit is associated with one log data byte. The detailed configuration of the simulated system is listed in Table III.

TABLE III
Simulator Configuration.

Component	Configuration	
Cores	8 cores, 3GHz, in-order	
L1 I/D Cache	Private 32KB, 8-way, 64B lines, 4 cycles	
L2 Cache	Private 256KB, 8-way, 64B lines, 12 cycles	
L3 Cache	Shared 8MB, 16-way, 64B lines, 28 cycles	
Main Memory	8GB TLC RRAM, 4 channels, 1 ranks, 8 banks, FRFCFS-WQF, 64-entry queue, 80% drain watermark	
Latency and energy of TLC RRAM [42], [45], [61]		
Read	25ns latency	
Write	Latency (ns) 000: 15.2 100: 150 001: 46.8 101: 101 010: 98.3 110: 52.7 011: 143 111: 12.1	Energy per cell (pJ) 000: 2 100: 35.6 001: 6.7 101: 19.6 010: 19.3 110: 8.5 011: 35.1 111: 1.5

TABLE IV
Benchmarks used in our experiments.

Benchmarks		Description
Micro-Benchmarks [23], [24], [43]	BTree	Insert/delete nodes in a b-tree
	Hash	Insert/delete entries in a hash table
	Queue	Insert/delete entries in a queue
	RBTree	Insert/delete nodes in a red-black tree
	SDG	Insert/delete edges in a scalable graph
	SPS	Swap two random entries in an array
Macro-Benchmarks [22], [39]	Echo	A scalable key-value store
	YCSB	20%/80% of read/update
	TPCC	New order transactions

Workloads. We evaluate MorLog with the benchmarks listed in Table IV. The micro-benchmarks are similar to the benchmarks used in previous studies [23], [24], [43]. The macro-benchmarks are from WHISPER [39] benchmarks, but are modified to allocate memory by using pmalloc/pfree interfaces instead of mmap interfaces (similar to [22]). In our simulation, all the workloads execute 100K transactions. The micro-benchmarks are run with eight threads, while the macro-benchmarks are run with four threads. We evaluate all the benchmarks except `TPCC` with both the small (64B) and the large (4KB) dataset (e.g., tree nodes). For `TPCC`, we adopt the new order transaction workload.

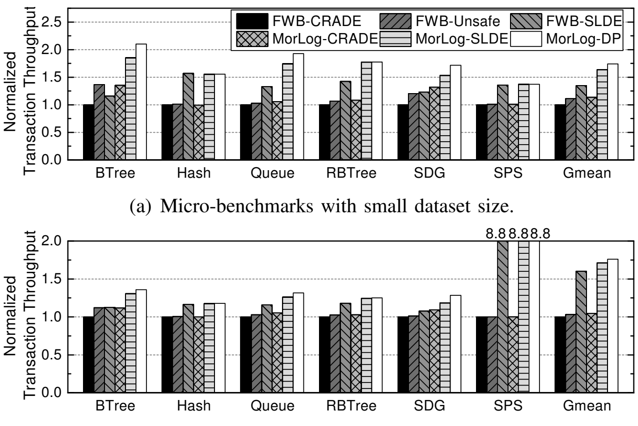
Evaluated designs. We evaluate the following designs:

- **FWB-CRADE**: The prior undo+redo hardware logging design, FWB [43], works with the existing coding mechanism, CRADE [61]. FWB also performs force-write-back every three million cycles. The log buffer size is the same as the undo+redo buffer.
- **FWB-Unsafe**: FWB [43] works with CRADE [61]. The log buffer size is the same as the total size of the undo+redo and redo buffers. We use this design to show the performance improvement by simply increasing the log buffer size. Note that this design cannot guarantee atomic persistence due to the large log buffer.
- **FWB-SLDE**: FWB [43] works with our coding mechanism, SLDE. Each log buffer entry is added with an extra dirty flag. The dirty flag is set by comparing the undo and the redo data when the log entry is created.
- **MorLog-CRADE**: Our morphable logging design works with CRADE [61]. The delay-persistence commit protocol is disabled to ensure persistence.
- **MorLog-SLDE**: Our morphable logging design works with SLDE. The delay-persistence commit protocol is disabled.
- **MorLog-DP**: Our design with all the optimizations discussed above (§III and §IV).

B. Performance Comparison

Transaction throughput. Figure 12 shows the transaction throughput of the evaluated designs on the micro-benchmarks, normalized to `FWB-CRADE`. `MorLog-CRADE` outperforms `FWB-Unsafe` in some benchmarks (e.g. `SDG`), since the writes of intermediate redo data can be avoided by leveraging the L1 caches to buffer redo data. As the redo data in the L1 caches need to be persisted during commit, `MorLog-CRADE` slightly degrades the transaction throughput by up to 1.9% in some benchmarks, such as `BTree` and `Hash`. `MorLog-SLDE` outperforms `MorLog-CRADE` by 44.7% and 63.4% on average for the small and the large dataset size, respectively. This is because SLDE avoids silent log writes and encodes log data by using the methods with less write cost. `MorLog-DP` improves the transaction throughput by up to 13.3% compared with `MorLog-SLDE`, since transaction commit does not need to wait for persisting log data.

MorLog shows more performance improvement for the small dataset size in all the micro-benchmarks except `SPS`. With the large dataset size, the log buffers and the L1 caches are filled with the log data from several dataset updates, which reduces the possibilities of coalescing redo data. Furthermore, as the dataset size increases, more dirty data of the dataset are logged, thus less data can be compressed by DLDC. For `SPS` with the large dataset size, MorLog significantly improves performance by avoiding writing clean log data, since the array entries are initialized with the same value.



(a) Micro-benchmarks with small dataset size.

(b) Micro-benchmarks with large dataset size.

Fig. 12. Transaction throughput normalized to `FWB-CRADE`.

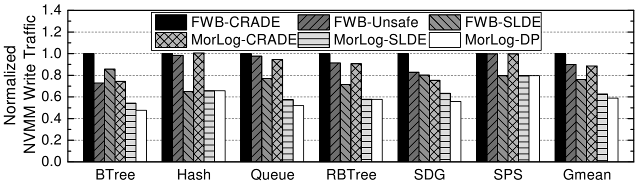


Fig. 13. NVMM write traffic normalized to `FWB-CRADE` for benchmarks with small dataset size.

NVMM write traffic. Figure 13 shows the number of NVMM writes on the micro-benchmarks with the small dataset size, normalized to `FWB-CRADE`. The results for the large dataset size show a similar trend, but we do not show them due to space constraints. Both morphable logging and SLDE can help to reduce NVMM writes. The number of NVMM writes is reduced by up to 25.6% in `MorLog-CRADE` compared with `FWB-CRADE`, while the number is further reduced by up to 39.3% in `MorLog-SLDE`. Furthermore, by comparing `MorLog-SLDE` and `MorLog-DP`, we observe that the delay-persistence commit protocol reduces NVMM writes by 11.9%. This is because redo entries are not forced to NVMM during commit. These entries can be directly discarded from the redo buffer after the corresponding updated data are persisted.

Energy consumption. Table V shows the NVMM write energy reduction compared with `FWB-CRADE`. The results are averaged over all the micro-benchmarks. SLDE provides substantial energy reduction by avoiding silent log writes and compressing log data with DLDC. The compressed log data of DLDC may be smaller than those of FPC, so that a more energy-efficient expansion coding can be used.

C. Endurance

Reducing the number of writes also improves the lifetime of NVMM. To show how MorLog affects the lifetime, we disable

TABLE V

NVMM WRITE ENERGY REDUCTION COMPARED WITH FWB-CRADE.

Dataset Size	FWB-Unsafe	FWB-SLDE	MorLog-CRADE	MorLog-SLDE	MorLog-DP
Small	0.6%	39.5%	2.1%	43.7%	45.9%
Large	1.6%	30.3%	4.3%	34.6%	36.0%

TABLE VI

LOG BIT REDUCTION COMPARED WITH FWB-CRADE WHEN THE EXPANSION CODING IS DISABLED.

Dataset Size	FWB-Unsafe	FWB-SLDE	MorLog-CRADE	MorLog-SLDE	MorLog-DP
Small	10.4%	41.6%	16.0%	57.1%	59.5%
Large	4.2%	33.7%	9.9%	43.5%	45.8%

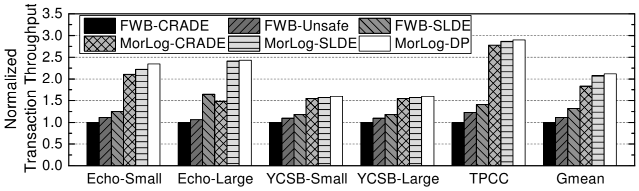


Fig. 14. Transaction throughput on the macro-benchmarks.

the expansion coding during simulation, since it may increase the number of bits written to NVMM. Table VI lists the log bit reduction compared with FWB-CRADE, averaged over all the micro-benchmarks. MorLog-DP reduces the number of log bits by 59.5% and 45.8% for the small and the large dataset size, respectively. Therefore, MorLog can improve the lifetime of NVMM.

D. Macro-Benchmark Throughput

Figure 14 shows the transaction throughput on the macro-benchmarks, normalized to FWB-CRADE. MorLog continues to provide performance improvement for the macro-benchmarks. Specifically, MorLog-CRADE outperforms FWB-CRADE by 83.8% on average. MorLog-SLDE improves the transaction throughput by 12.8% over MorLog-CRADE, while MorLog-DP further provides 2.1% of improvement. Note that morphable logging performs better for the macro-benchmarks. This is because the macro-benchmarks have better temporal locality than the micro-benchmarks that access data structures with random keys.

E. Sensitivity Study

Log buffer size. Figure 15 shows the transaction throughput and the number of NVMM writes on different buffer sizes for the echo benchmark. We use Redo002 to represent MorLog-SLDE with a 2-entry redo buffer, and similarly label other designs. The results are normalized to Redo002 with a 1-entry undo+redo buffer. As the undo+redo buffer size increases, the number of NVMM writes decreases, while the transaction throughput first increases and then drops due to the long commit latency. We make a tradeoff between performance and hardware overhead, and adopt a 16-entry undo+redo buffer. As Redo032 performs best when the undo+redo buffer size is 16, we use a 32-entry redo buffer in our experiments.

Number of threads. Figure 16 shows the transaction throughput on different thread numbers. The results are averaged over the micro-benchmarks and normalized to FWB-CRADE. MorLog continues to outperform FWB as the thread number increases. But, the transaction throughput drops for the large dataset size when the thread number is larger than four. This is because log entries may frequently be evicted from the two log buffers due to large dataset size and thread numbers, which decreases the chance of coalescing log data in the log buffers.

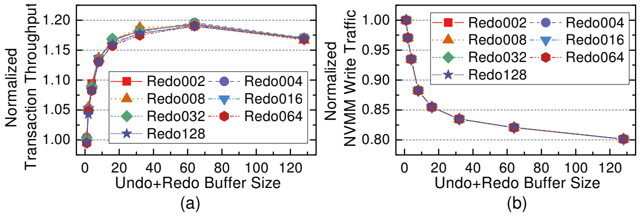


Fig. 15. Comparison of (a) transaction throughput and (b) NVMM write traffic on different log buffer sizes.

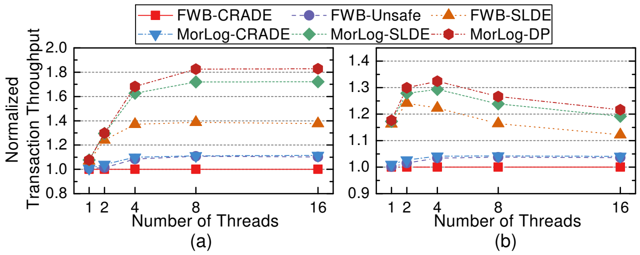


Fig. 16. Comparison of transaction throughput on different thread numbers for benchmarks with (a) small and (b) large dataset size.

NVMM latencies. We also measure the transaction throughput while NVMM write latency varies from 1 to 32 times of the latency listed in Table III. The results normalized to FWB-CRADE are similar to those shown in Figure 12. Specifically, as the write latency increases, the change in the normalized results is less than 1.9%. Therefore, NVMM write latency has negligible effect on the efficiency of MorLog.

VII. RELATED WORK

Atomic persistence through logging. WAL is widely used to support atomic persistence for NVMM in many existing designs [6], [7], [10], [55]. These designs enforce ordering constraints in software. The ordering constraints can be reasoned about through memory persistency models [27], [47]. To reduce the ordering overhead, recent studies propose to commit transaction in batch [48], persist updated data in bulk [35], and delay lock releases after updating in-place data [28]. Besides, SoftWrAP [14] and DudeTM [34] decouple persist operations from transaction execution by using a DRAM cache. Different from prior studies that adopt software logging, MorLog focuses on the hardware logging approach.

Hardware logging designs with various log types are proposed to mitigate logging overhead. The prior undo designs [24], [53] require to synchronously persist updated data during commit. The prior redo designs [12], [22], [23] need to enforce the ordering constraints between redo data and updated data. To this end, WrAP [12] uses redo data to update in-place data, thus causing extra NVMM reads. ReDU [22] buffers the evicted data in a DRAM cache, which consumes DRAM capacity. DHTM [23] aborts the transaction if its updated data are evicted from the LLC, which limits the transaction size. The prior undo+redo designs [43] and MorLog combine the benefits of both undo and redo logging. However, the undo+redo designs increase the amount of log data since undo+redo entries are created for each update, while MorLog reduces log data through morphable logging and SLDE.

Except for WAL, some prior studies propose new logging designs. Write-Behind Logging [2] allows in-place data to be updated before persisting log data, but can be used only in a multi-version system. JUSTDO logging [19] continues execution from the instruction interrupted by the system failures during recovery, but requires that all the updates in a failure-atomic region only access NVMM.

Other approaches for atomic persistence. Atomic persistence can also be ensured through multi-versioning. Kamino-Tx [38] maintains a main and a backup version of data. The two versions are stored in different regions of NVMM. The main version is updated during transaction execution, while the backup version is indirectly updated with the main version of data after commit. Direct updates to the backup version can be enabled by holding the main version in a non-volatile LLC [65] or a non-volatile buffer alongside the cache hierarchy [26], [29], [39]. In the case that the transaction size exceeds the size of the LLC or the buffer, the transaction execution falls back to a low-performance mode.

Some existing NVMM system designs [40], [41], [50], [58] guarantee atomic persistence through software-transparent checkpointing. These designs back up system states during checkpointing, and recover by rolling back to the latest checkpoint. These designs can recover the whole memory state, but need to persist the memory data in both DRAM and NVMM when creating checkpoints.

Data encoding techniques. Data encoding techniques are proposed to alleviate the high overhead of programming NVM cells. The data flip methods [9], [37] flip the original data when the number of dirty bits exceeds a threshold. The coset coding methods [20], [51], [52] maps the original data into a set of vectors, and selects the vector according to the write cost (e.g., the number of bit flips or energy consumption). The write cost can also be reduced by mapping most frequent data patterns to most efficient states of MLCs/TLCs [8], [56], or using only the desirable states [42]. Most of the above methods require extra encoding tag bits. Some researches [21], [44], [45], [60], [61] reduce the storage overhead of tag bits by combining these methods with compression [11], [46]. All the existing data encoding methods mentioned above do not distinguish between log data and updated data, which causes many clean log data to be logged. In contrast, MorLog can write only dirty log data to NVMM with DLDC.

VIII. Conclusion

We propose MorLog that logs only data necessary for recovery and dynamically selects the encoding method for log data. MorLog consists of two mechanisms: morphable logging and SLDE. Morphable logging creates an undo+redo log entry for the first update to the data in a transaction. The undo+redo entry is eagerly written to NVMM. After that, only redo data are logged for the subsequent updates to the data. Redo data are coalesced in the redo buffer and the L1 caches. To reduce commit latency, we provide an optional delay-persistence commit protocol. SLDE adopts DLDC to encode log data with many clean bytes. DLDC first discards clean log data and then compresses remained dirty log data. When encoding data, SLDE dynamically chooses between DLDC and the existing encoding method according to the write cost of encoded log data. Our evaluation shows that MorLog significantly improves transaction throughput, reduces NVMM write traffic, and decreases NVMM write energy.

Acknowledgment

This work is supported by the National Natural Science Foundation of China No.61821003, No.61832007, No.61772222, No.U1705261, Major Project of National Science and Technology of China No.2017ZX01032-101, and the Fundamental Research Funds for the Central Universities No.2019kfyXMBZ037.

References

[1] M. Arjomand, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Boosting Access Parallelism to PCM-based Main Memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 695–706.

[2] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind Logging," *Proc. VLDB Endow.*, vol. 10, no. 4, pp. 337–348, Nov. 2016.

[3] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 263–276.

[4] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-NVM: Persistency for Integrity-protected and Encrypted Non-volatile Memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 104–115.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[6] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging Locks for Non-volatile Memory Consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 433–452.

[7] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "REWIND: Recovery write-ahead system for in-memory non-volatile data-structures," *Proc. VLDB Endow.*, vol. 8, no. 5, pp. 497–508, 2015.

[8] Y. Chen, W. Wong, H. Li, and C. Koh, "Processor caches built using multi-level spin-transfer torque RAM cells," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2011, pp. 73–78.

[9] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 347–357.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 105–118.

[11] D. B. Dgien, P. M. Palangappa, N. A. Hunter, J. Li, and K. Mohanram, "Compression architecture for bit-write reduction in non-volatile memory technologies," in *2014 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, 2014, pp. 51–56.

[12] K. Doshi, E. Giles, and P. Varman, “Atomic persistence for SCM with a non-intrusive backend controller,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 77–89.

[13] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush, “A 16gb ReRAM with 200mb/s write and 1gb/s read in 27nm technology,” in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 338–339.

[14] E. R. Giles, K. Doshi, and P. Varman, “SoftWrAP: A lightweight framework for transactional support of storage class memory,” in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, May 2015, pp. 1–14.

[15] J. Huang, K. Schwan, and M. K. Qureshi, “NVRAM-aware Logging in Transaction Systems,” *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 389–400, Dec. 2014.

[16] Intel, “Intel Xeon Processor D Product Family Technical Overview.” [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview#_Toc419802876

[17] Intel, “Intel and Micron Produce Breakthrough Memory Technology | Intel Newsroom,” 2015. [Online]. Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/

[18] Intel, “Intel® 64 and IA-32 Architectures Software Developer Manuals,” 2018. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[19] J. Izraelevitz, T. Kelly, and A. Kolli, “Failure-Atomic Persistent Memory Updates via JUSTDO Logging,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016, pp. 427–442.

[20] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin, “Coset coding to extend the lifetime of memory,” in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 222–233.

[21] M. Jalili and H. Sarbazi-Azad, “Captopril: Reducing the pressure of bit flips on hot locations in non-volatile main memories,” in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 1116–1119.

[22] J. Jeong, C. H. Park, J. Huh, and S. Maeng, “Efficient Hardware-Assisted Logging with Asynchronous and Direct-Update for Persistent Memory,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 520–532.

[23] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, “DHTM: Durable Hardware Transactional Memory,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 452–465.

[24] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, “ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 361–372.

[25] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, “NVWAL: Exploiting NVRAM in Write-Ahead Logging,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016, pp. 385–398.

[26] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, “Delegated persist ordering,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[27] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, “Language-level Persistency,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 481–493.

[28] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, “High-Performance Transactions for Persistent Memories,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 399–411.

[29] C.-H. Lai, J. Zhao, and C.-L. Yang, “Leave the Cache Hierarchy Operation As It Is: A New Persistent Memory Accelerating Approach,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 5:1–5:6.

[30] L. Lamport, “Proving the Correctness of Multiprocess Programs,” *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977.

[31] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting Phase Change Memory As a Scalable Dram Alternative,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 2–13.

[32] H. G. Lee, S. Baek, J. Kim, and C. Nicopoulos, “A Compression-Based Hybrid MLC/SLC Management Technique for Phase-Change Memory Systems,” in *2012 IEEE Computer Society Annual Symposium on VLSI*, Aug. 2012, pp. 386–391.

[33] Y. Lee, S. Kim, S. Hong, and J. Lee, “Skinflint DRAM system: Minimizing DRAM chip writes for low power,” in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 25–34.

[34] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, “DudeTM: Building Durable Transactions with Decoupling for Persistent Memory,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 329–343.

[35] Y. Lu, J. Shu, and L. Sun, “Blurred persistence in transactional persistent memory,” in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, May 2015, pp. 1–13.

[36] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[37] R. Maddah, S. M. Seyedzadeh, and R. Melhem, “CAFO: Cost aware flip optimization for asymmetric memories,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 320–330.

[38] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, “Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx,” in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 499–512.

[39] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, “An Analysis of Persistent Memory Use with WHISPER,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 135–148.

[40] D. Narayanan and O. Hodson, “Whole-system Persistence,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012, pp. 401–410.

[41] T. Nguyen and D. Wentzlaff, “PiCL: A Software-Transparent, Persistent Cache Log for Nonvolatile Main Memory,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 507–519.

[42] D. Niu, Q. Zou, C. Xu, and Y. Xie, “Low power multi-level-cell resistive memory design with incomplete data mapping,” in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 131–137.

[43] M. A. Ogleari, E. L. Miller, and J. Zhao, “Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 336–349.

[44] P. M. Palangappa and K. Mohanram, “Flip-Mirror-Rotate: An Architecture for Bit-write Reduction and Wear Leveling in Non-volatile Memories,” in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, 2015, pp. 221–224.

[45] ——, “CompEx: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVM,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 90–101.

[46] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, “Base-delta-immediate compression: Practical data compression for on-chip caches,” in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 377–388.

[47] S. Pelley, P. M. Chen, and T. F. Wenisch, “Memory persistency,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. IEEE Press, 2014, pp. 265–276.

[48] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, “Storage Management in the NVRAM Era,” *Proc. VLDB Endow.*, vol. 7, no. 2, pp. 121–132, Oct. 2013.

[49] M. Poremba, T. Zhang, and Y. Xie, "NVMain 2.0: A User-Friendly Memory Simulator to Model (Non-)Volatile Memory Systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, 2015.

[50] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 672–685.

[51] S. Seyedzadeh, A. Jones, and R. Melhem, "Enabling Fine-Grain Restricted Coset Coding Through Word-Level Compression for PCM," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 350–361.

[52] S. M. Seyedzadeh, R. Maddah, A. Jones, and R. Melhem, "PRES: Pseudo-Random Encoding Scheme to increase the bit flip reduction in the memory," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[53] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 178–190.

[54] S. Shin, J. Tuck, and Y. Solihin, "Hiding the Long Latency of Persist Barriers Using Speculative Execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 175–186.

[55] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 91–104.

[56] J. Wang, X. Dong, G. Sun, D. Niu, and Y. Xie, "Energy-efficient multi-level cell phase-change memory system with data encoding," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, Oct. 2011, pp. 175–182.

[57] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 865–876, Jun. 2014.

[58] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye, "NICO: Reducing Software-Transparent Crash Consistency Cost for Persistent Memory," *IEEE Transactions on Computers*, pp. 1–12, 2018.

[59] F. Xia, D. Jiang, J. Xiong, M. Chen, L. Zhang, and N. Sun, "DWC: Dynamic Write Consolidation for Phase Change Memory Systems," in *Proceedings of the 28th ACM International Conference on Supercomputing*, 2014, pp. 211–220.

[60] J. Xu, D. Feng, Y. Hua, W. Tong, J. Liu, C. Li, G. Xu, and Y. Chen, "Adaptive Granularity Encoding For Energy-Efficient Non-Volatile Main Memory," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[61] J. Xu, D. Feng, Y. Hua, W. Tong, J. Liu, C. Li, and W. Zhou, "Improving Performance of TLC RRAM with Compression-Ratio-Aware Data Encoding," in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 573–580.

[62] B. Yang, J. Lee, J. Kim, J. Cho, S. Lee, and B. Yu, "A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme," in *2007 IEEE International Symposium on Circuits and Systems*, May 2007, pp. 3014–3017.

[63] M. Ye, C. Hughes, and A. Awad, "Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 403–415.

[64] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-Efficient Encryption for Non-Volatile Memories," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 33–44.

[65] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 421–432.

[66] K. A. Zubair and A. Awad, "Anubis: Ultra-low Overhead and Recovery Time for Secure Non-volatile Memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 157–168.

[67] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the Performance and Endurance of Encrypted Non-Volatile Main Memory through Deduplicating Writes," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 442–454.