HOOP: Efficient Hardware-Assisted Out-of-Place Update for Non-Volatile Memory

Miao Cai*
Computer Science
Nanjing University

Chance C. Coats
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Jian Huang
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

***Abstract*—Byte-addressable non-volatile memory (NVM) is a promising technology that provides near-DRAM performance with scalable memory capacity. However, it requires atomic data durability to ensure memory persistency. Therefore, many techniques, including logging and shadow paging, have been proposed. However, most of them either introduce extra write traffic to NVM or suffer from significant performance overhead on the critical path of program execution, or even both.**

In this paper, we propose a transparent and efficient hardware-assisted out-of-place update (HOOP) mechanism that supports atomic data durability, without incurring much extra writes and performance overhead. The key idea is to write the updated data to a new place in NVM, while retaining the old data until the updated data becomes durable. To support this, we develop a lightweight indirection layer in the memory controller to enable efficient address translation and adaptive garbage collection for NVM. We evaluate HOOP with a variety of popular data structures and data-intensive applications, including key-value stores and databases. Our evaluation shows that HOOP achieves low critical-path latency with small write amplification, which is close to that of a native system without persistence support. Compared with state-of-the-art crash-consistency techniques, it improves application performance by up to 1.7×, while reducing the write amplification by up to 2.1×. HOOP also demonstrates scalable data recovery capability on multi-core systems.

***Index Terms*—Non-volatile memory, out-of-place update, logging, memory persistency**

I. INTRODUCTION

Emerging non-volatile memory (NVM) like PCM [28], [44], [55], STT-RAM [25], [42], ReRAM [48], and 3D XPoint [2] offers promising properties, including byte-addressability, non-volatility, and scalable capacity. Unlike DRAM-based systems, applications running on NVM require memory persistency to ensure crash safety [19], [26], [41], [50], [54], which means a set of data updates must behave in an atomic, consistent, and durable manner with respect to system failures and crashes.

Ensuring memory persistency with commodity out-of-order processors and hardware-controlled cache hierarchies, however, is challenging and costly due to unpredictable cache evictions. Prior researches have developed various crash-consistency techniques for NVM, such as logging [34], shadow paging [10], and their optimized versions (see details in Table I and § II-B). However, they either introduce extra write traffic to NVM, or suffer from significant performance overhead on the critical path of program execution, or even both.

*Work done while visiting the Systems Platform Research Group at UIUC.

Specifically, although logging provides strong atomic durability against system crashes, it introduces significant overheads. First, both undo logging and redo logging must make a data copy before performing the in-place update. Persisting these data copies incurs extra writes to NVM on the critical path of program execution [33], [38]. This not only decreases application performance, but also hurts NVM lifetime [6], [30], [43], [44]. Second, enforcing the correct persistence ordering between log and data updates requires cache flushes and memory fences [1], [29], which further causes significant performance overheads [17], [23], [24], [40], [47].

To address the aforementioned problems, researchers recently proposed asynchronous in-place updates, such as DudeTM [29] and Redu [23], in which the systems maintain an explicit main copy of data to perform in-place updates, and then asynchronously apply these changes to the data copy, or asynchronously persist the undo logs to NVM [37]. Unfortunately, it does not mitigate the problem of incurring additional write traffic, due to the background data synchronization. Kiln [54] alleviates this drawback by using a non-volatile on-chip cache to buffer data updates. However, it requires hardware modification to the CPU architecture and its cache coherence protocol.

An alternative technique, shadow paging, incurs both additional data writes to NVM and performance overhead on the critical path, due to its copy-on-write (CoW) mechanism [10]. Ni et al. [38], [39] optimized shadow paging by enabling data copies at cache-line granularity. However, it requires TLB modifications to support the cache-line remapping. Another approach is the software-based log-structured memory [17], which reduces the persistency overhead by appending all updates to logs. However, it requires multiple memory accesses to identify the data location for each read, which incurs significant critical-path latency.

In this paper, we propose a transparent hardware-assisted out-of-place (OOP) update approach, named HOOP. The key idea of HOOP is to store the updated data outside of their original locations in dedicated memory regions in NVM, and then apply these updates lazily through an efficient garbage collection scheme. HOOP reduces data persistence overheads in three aspects. First, it eliminates the extra writes caused by the logging mechanisms, as the old data copies already exist in NVM and logging is not required. Second, the out-of-place update does not assume any persistence ordering for store

TABLE I: Comparison of various crash-consistency techniques for NVM. Compared with existing works, HOOP provides a transparent hardware solution that significantly reduces the write traffic to NVM, while achieving low persistence overhead.

Approach	Subtype	Representative Project	Read Latency	On the Critical Path	Require Flush & Fence	Write Traffic
Logging	Undo	DCT [27]	Low	Yes	No	High
		ATOM [24]	Low	Yes	No	Medium
		Proteus [47]	Low	Yes	No	Medium
		PiCL [37]	High	No	No	High
	Redo	Mnemosyne [49]	High	Yes	Yes	High
		LOC [32]	High	Yes	No	High
		BPPM [31]	Low	Yes	Yes	Medium
		SoftWrAP [14]	High	Yes	Yes	High
		WrAP [13]	High	Yes	No	High
		DudeTM [29]	Low	No	No	High
		ReDU [23]	High	Yes	No	Medium
	Undo+Redo	FWB [40]	High	Yes	No	High
Shadow paging	Page	BPFS [10]	Low	Yes	Yes	High
	Cache line	SSP [39]	Low	Yes	Yes	Low
Log-structured NVM		LSNVMM [17]	High	No	No	Medium
HOOP			**Low**	**No**	**No**	**Low**

operations, which allows them to execute in a conventional out-of-order manner. Third, persisting the new updates in new locations does not affect the old data version, which inherently supports the atomic data durability.

Since the update is written to a new place in NVM, we develop a lightweight indirection layer in the memory controller to handle the physical address remapping. HOOP enables high-performance and low-cost out-of-place update with four major components. First, we organize the dedicated memory regions for storing data updates in a log-structure manner, and apply data packing to the out-of-place updates. This makes HOOP best utilize the memory bandwidth of NVM as well as reduce the write traffic to NVM. Second, to reduce the memory space cost caused by the out-of-place updates, HOOP develops an efficient garbage collection (GC) algorithm to adaptively restore the out-of-place updates back to their home locations. To further reduce the data movement overhead during GC, we exploit a data coalescing scheme that combines the updates to the same cache lines. Therefore, HOOP only need to restore multiple data updates once, which further reduces the additional write traffic. Third, HOOP maintains a hash-based address-mapping table in HOOP for physical-to-physical address translation, and ensures that load operations always read the updated data from NVM with trivial address translation overhead. Since the entries in the address-mapping table will be cleaned when the corresponding out-of-place updates are periodically written back to their home addresses, the mapping table size is small. Fourth, HOOP enables fast data recovery by leveraging the thread parallelism available in multi-core computing systems.

As HOOP is developed in the memory controller, it is transparent to upper-level systems software. No non-volatile cache or TLB modifications for address translation are required. Unlike software-based logging approaches that suffer from long critical-path latency for read operations, HOOP provides an efficient hardware solution with low performance overhead and write traffic, as shown in Table I. Overall, we make the following contributions in this paper:

- We present a hardware out-of-place update scheme to ensure the crash-consistency for NVM, which alleviates extra write traffic and avoids critical-path latency overhead in NVM.
- We propose a lightweight persistence indirection layer in the memory controller with minimal hardware cost, which makes out-of-place updates transparent to software systems.
- We present an efficient and adaptive GC scheme, which will apply the recent data updates from the out-of-place update memory regions to their original locations for memory space saving and write traffic reduction.

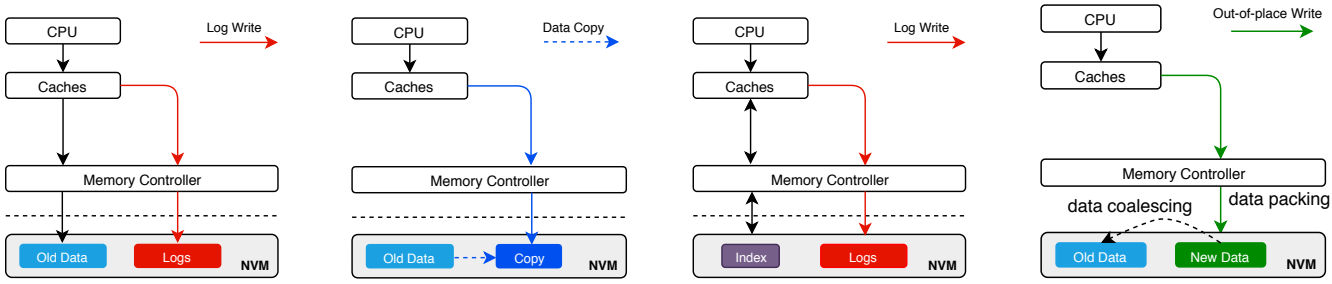
We implement HOOP in a Pin-based many-core simulator, McSimA+ [5], with the combination of an NVM simulator. We evaluate HOOP against four representative and well-optimized crash-consistency approaches, including undo logging [24], redo logging [13], optimized shadow paging [38], and log-structured NVM [17]. We use a set of microbenchmarks running against these popular data structures like hashmaps, B-trees [23], [24], [40], [47], and real-world data-intensive application workloads like Yahoo Cloud Service Benchmark (YCSB) and transactional databases [36]. Experimental results demonstrate that HOOP significantly outperforms state-of-the-art approaches by up to 1.7× in terms of transaction throughput, and reduces the write traffic to NVM by up to 2.1×, while ensuring the same atomic durability as existing crash-consistency techniques. HOOP also scales the data recovery as we increase the number of threads on a multi-core system.

We organize the rest of the paper as follows. We discuss the background and motivation in § II. We describe the design and implementation of HOOP in § III. We evaluate HOOP in § IV, present its related work in § V, and conclude it in § VI.

II. BACKGROUND AND MOTIVATION

A. Atomic Durability for NVM

NVM requires atomic durability to ensure crash consistency. Atomicity refers to a group of data updates happening in an all-or-nothing manner in case the program crashes, while durability requires these data updates are eventually persisted in the persistent storage. In modern memory hierarchies that consist of volatile CPU caches and persistent memory, unpredictable cache-line evictions make it challenging to achieve atomic durability, because they could cause only a subset of modified data to become durable before the system experiences an unexpected crash or application failure [8], [52], [54].



(a) Logging (b) Shadow Paging (c) Log-structured NVM (d) HOOP

Fig. 1: Illustration of different crash-consistency techniques. (a) Logging technique requires that both logs and data must be persisted, which incurs double writes; (b) Shadow paging maintains two copies of data, it suffers from copy-on-write overhead; (c) Log-structured NVM alleviates the double writes, but it suffers from significant overhead of index lookup; (d) Our hardware-assisted out-of-place update reduces the write amplification significantly, while providing efficient data accesses.

Enforcing atomic durability on current computer architectures is non-trivial. Commodity hardware provides instructions for atomic data update, but this hardware-supported atomicity only supports small granularities (8 bytes for 64-bit CPUs). To ensure these atomic updates become durable, applications must flush cache lines to persist data to NVM. For a group of data updates having a larger size, they have to rely on other crash-consistency techniques, such as write-ahead logging and shadow paging, to achieve atomic durability. Although these crash-consistency mechanisms support strong atomic durability, applying them to NVM is costly. We will discuss their details in the following section § II-B.

B. Crash-consistency Techniques

We categorize the crash-consistency techniques for NVM into three major types: logging [34], shadow paging [10], and log-structured NVM [17]. We summarize their recent representative work in Table I.

Logging on NVM: Write ahead logging (WAL) is widely-used for NVM. Its core idea is to preserve a copy before applying the change to the original data (see Figure 1a). The logging operations result in doubled write traffic and worsen the wear issue with NVM [17], [38]. To reduce the logging overhead, hardware-assisted logging schemes such as bulk persistence [23], [24], [31], [40] have been proposed. However, they can only partially mitigate the extra write traffic (see Figure 8 in our evaluation).

Beyond increasing the write traffic, logging also incurs lengthy critical-path latency [29], [41]. This issue is especially serious for undo logging, since it requires a strict persist ordering between log entries and data writes. Redo logging provides more flexibility, as it allows asynchronous log truncation and data checkpointing [13], [14], [23], [31], [49], which contributes to a shorter critical-path latency. However, it still generates doubled write traffic eventually.

Decoupling logging from data updates with asynchronous in-place update is another approach, as proposed in Soft-WrAP [14] and DudeTM [29]. It decouples the execution of durable transactions and logging, therefore, the number of memory barrier operations can be reduced. However, it needs to track the updated data versions, and the software-based address translation inevitably introduces additional overhead to the critical-path latency. And this approach still cannot reduce the write traffic to NVM.

Despite each of these logging approaches applying various optimizations, logging is still expensive, due to a simple reason: they are restricted by their intrinsic extra log write for each update, regardless of whether the update takes place synchronously or asynchronously.

Shadow Paging: Shadow paging can eliminate expensive cache flushes and memory fence instructions, however its write amplification is still a severe issue. With shadow paging, an entire page has to be copied, even though only a small portion of data is modified (see Figure 1b). Recent work proposed a fine-grained copy-on-write technique [38], [39] to reduce the write amplification. In this approach, one virtual cache line is mapped to two physical cache lines, and it ensures data atomicity at cache-line granularity. However, it requires frequent TLB updates to track the committed cached lines in NVM, which would sacrifice the performance benefits obtained from the cache-line copy-on-write optimization.

Log-structured NVM: Inspired by log-structured file systems [46], Hu et al. [17] proposed a software-based log-structured NVM, called LSNVMM, in which all the writes are appended into a log. Such an approach alleviates the double writes caused by the undo/redo logging. However, it incurs significant software overhead for read operations due to the complicated data indexing (see Figure 1c) and garbage collection. Although the index can be cached in DRAM, it still requires multiple memory accesses to obtain the data location. For instance, LSNVMM requires $\mathcal{O}(\log N)$ memory accesses for each data read, due to the address lookup in an index tree, where N is the number of log entries. This significantly increases the read latency of NVM.

C. Why Hardware-Assisted Out-of-Place Update

As discussed, the proposed crash-consistency approaches, such as logging, shadow paging, and log-structured memory, either increase the write amplification of NVM, or cause long critical-path data access latency, or even both.

In this paper, we propose a new approach: hardware-assisted out-of-place update, in which the memory controller writes the new data to a different memory location in a log-structured manner, and asynchronously applies the data update to its home address periodically. It alleviates the extra write traffic caused by the logging, and avoids the data copying on the critical path as discussed in shadow paging. Our proposed approach maintains a small physical-to-physical address mapping table in the memory controller for address translation, and adaptively writes the data updates into their home addresses. Therefore, it incurs minimal indirection and GC overhead. We further leverage data packing and coalescing to reduce the write traffic to NVM during the GC.

Our proposed hardware out-of-place update ensures the atomic data durability by default, as it always maintains the old data version in NVM, while persisting the updates in new memory locations in a log-structured manner. It also does not assume any persistence ordering for store operations as it is implemented in the memory controller, which significantly reduces the performance overhead caused by memory barriers.

III. Design and Implementation

A. Design Goals and Challenges

To perform hardware-assisted out-of-place updates efficiently, we aim to achieve the following three goals: (1) we will guarantee crash-consistency, while minimizing critical path latency and write traffic to NVM; (2) we aim to make trivial hardware modifications, thus minimizing the cost of our solution, while simplifying the upper-level software programming; (3) we will develop a scalable and fast data recovery scheme by exploiting the multi-core computing resources.

To achieve hardware-assisted out-of-place update, a straightforward approach is to persist updated cache lines along with necessary metadata to NVM in an out-of-place manner. However, we have to overcome the following challenges. First, persisting the data and metadata eagerly at a cache-line granularity will introduce extra write traffic as well as negatively affect system performance. Second, the indexing for out-of-place updates could introduce additional performance overhead to data accesses. Third, the required GC operations for free space will introduce additional write traffic to NVM as well as performance overhead. Inspired by the flash translation layer for flash-based solid-state drives [4], [15], [18], we propose an optimized and lightweight indirection layer in the memory controller to address these aforementioned challenges.

B. System Overview

To support memory persistency, transactional mechanisms have been developed as the standard approach because of their programming simplicity [23], [24], [29], [45], [49], [54]. Instead of inventing a new programming interface for NVM, Hoop provides two transaction-like interfaces (i.e., *Tx_begin* and *Tx_end*) to programs, and enables programmers to handle the concurrency control of transactions for flexibility, as proposed in these prior studies. These interfaces demarcate the beginning and end of a transaction which requires data atomic durability. Hoop only needs programmers to specify a failure-atomic region using *Tx_begin* and *Tx_end*, without requiring them to manually wrapping all read and write operations, and adding `clwb` and `mfence` instructions.

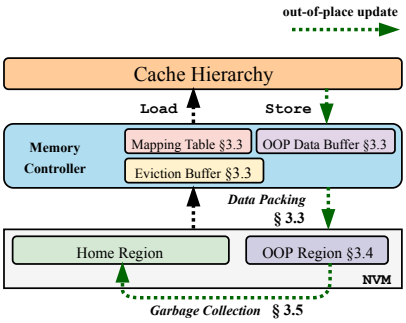


Fig. 2: Hardware-assisted out-of-place update with Hoop. Hoop performs out-of-place writes and reduces write traffic with *data packing and coalescing*. To reduce the storage overhead, Hoop adaptively migrates data in the out-of-place (OOP) region back to the home region with optimized GC.

We present the architectural overview of Hoop in Figure 2. During transaction execution, data is brought into the cache hierarchy with `load` and `store` operations. They will access the indirection layer to find the most recent version of the desired cache line (see §III-C). For the updated cache lines in a transaction, they are buffered in the *OOP data buffer* in Hoop. Each entry of this buffer can hold multiple data updates as well as the associated metadata. And persistence optimizations such as data packing are applied to improve the transaction performance, when flushing the updated cache lines to the OOP region (see §III-D). With the out-of-place writes, Hoop is crash-safe by ensuring committed transactions are persisted in the OOP region before any changes are made to the original addresses (i.e., the home region). As OOP region will be filled with updated data and metadata, Hoop performs periodic GC to migrate the most recent data versions to the home region, and uses *data coalescing* to minimize the write traffic to NVM (see §III-E). Upon power failures or system crashes, Hoop will leverage thread parallelism to scan the OOP region and instantly recover the data to a consistent state (see §III-F).

C. Indirection Layer in the Memory Controller

To provide crash-safety, Hoop must ensure that all updates from a transaction have been written to NVM before any of the modified cache lines in the transaction could be evicted and written to the home region. To guarantee this ordering, Hoop writes cache lines which are modified by transactions into the OOP region instead of the home region.

OOP Data Buffer. To improve the performance of out-of-place updates, Hoop reserves an OOP data buffer in the memory controller (see Figure 2). Each core has a dedicated OOP buffer entry (1KB per core) to avoid access contention during concurrent transaction execution. It stores the updated cache lines and associated metadata (i.e., home-region address), and facilitates the data packing for further traffic reduction. Specifically, Hoop tracks data updates at a word granularity

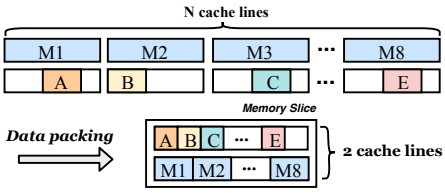


Fig. 3: Data packing in HOOP.

instead of a cache line granularity during data persistence, motivated by the prior studies showing that many application workloads update data at a fine granularity [9], [53]. HOOP applies data packing to reduce the write traffic during out-of-place updates. As shown in Figure 3, data residing in several independent cache lines are compacted into one single cache line. Similarly, HOOP also performs metadata packing for further traffic reduction. We show that metadata which are associated with eight data updates are also packed into a single cache line in Figure 3.

HOOP packs up to eight pieces of data and their metadata into a single unit, named a *memory slice* (see the details in § III-D). As for multiple updates in the same cache line happened in a transaction, HOOP will pack them in the same memory slice. We use a 40-bit address offset preserved in the metadata to address the home region (1TB). As NVM could have a larger capacity, the metadata size will also increase. To overcome this challenge, HOOP needs to only reduce the number of cache lines being packed (N in Figure 3). For a home region whose size is 1 PB (2^{50}), HOOP can pack seven units of cache lines (56 bytes) and their metadata in a memory slice, which still occupies two cache lines.

Persistence Ordering: HOOP flushes the updated data and metadata to the OOP region in two scenarios. First, if the HOOP has packed eight cache lines in the OOP data buffer during transaction execution, it will flush the packed memory slice into the OOP region. Second, if the transaction executes the *Tx_end* instruction, HOOP will automatically flush the remaining data along their metadata to the OOP region.

HOOP maintains the persistence ordering in the memory controller, which does not require programmers to explicitly execute cache-line flushes and memory barriers. We depict the transaction execution of different approaches in Figure 4. Undo logging requires strict ordering for each data update, incurring a substantial number of persistence operations during transaction execution. Redo logging mitigates this issue and only requires two flush operations per transaction: one for the redo logs and another for the data updates. Both schemes have to perform extra writes to NVM. The optimized shadow paging scheme can avoid additional data copy overheads. However, as shadow paging can only guarantee data atomicity. To ensure data persistence, it has to eagerly flush the updated cache lines, causing severe persistency overhead. HOOP uses the OOP data buffer to store the data updated by a transaction, and flushes the data in the unit of memory slice. Upon *Tx_end* instruction, HOOP persists the last memory slice to the OOP region.

Address Mapping. To track these cache lines for future accesses, HOOP uses a small hash table maintained in the

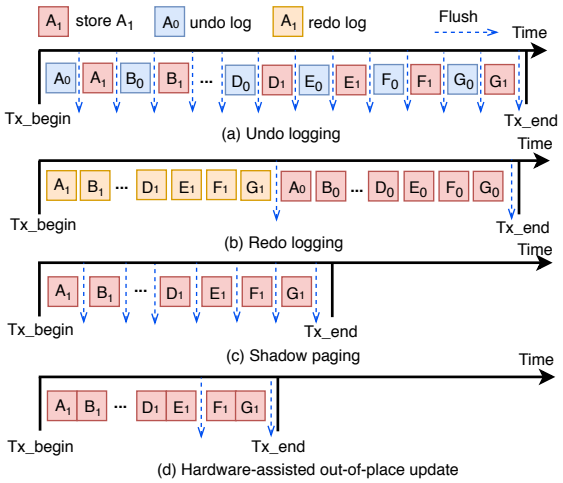


Fig. 4: Transaction execution of different approaches. Both undo and redo logging deliver lengthy transaction execution times due to log writes. Shadow paging has to copy additional data before performing in-place updates. HOOP achieves fast transaction execution with out-of-place updates.

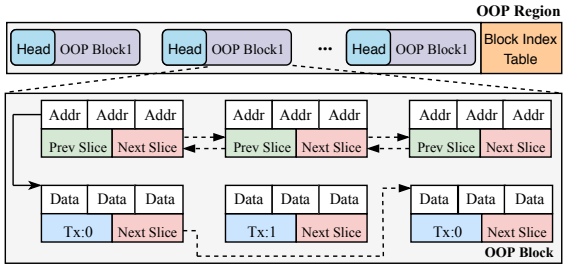
memory controller to map from home region addresses to the OOP region addresses (physical-to-physical address mapping). Compared to the software-based approaches, HOOP performs the address translation transparently in hardware and avoids expensive software overheads and TLB shootdown [17], [38].

Whenever a cache line is evicted from the LLC within a transaction, the cache line is written into the OOP region. HOOP adds an entry to the mapping table for tracking its location. In the mapping table, each entry contains its home-region address as well as the OOP-region address.

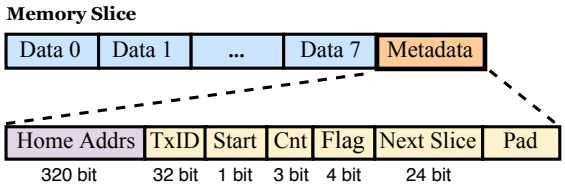
HOOP removes entries in the mapping table under two conditions. First, the most recent data versions have been migrated from the OOP region to the home region during the GC (see details in §III-E). Second, upon an LLC miss, HOOP will check the mapping table to determine whether the cache line should be read from the home region or OOP region. If its address is present in the table, it will be read from the OOP region. HOOP will remove this entry, since the most recent version is located within the cache hierarchy, and the existing cache coherence mechanisms will ensure this data will be read by any other requesting cores.

The mapping table in HOOP is essentially used to track the cache lines in the OOP region. It is shared by all cores. Its size is a function of the maximum number of evicted and flushed cache lines between two consecutive GC operations in NVM. In this paper, we use 256 KB per core as the size of the mapping table (2 MB in total) by default. And our evaluation (see §IV) shows this size provides reasonable performance guarantee for data-intensive applications.

Eviction Buffer. Along with the mapping table, HOOP has an *eviction buffer* to store cache lines (and their home-region addresses) that were written back to NVM during GC. By buffering the evicted cache lines, HOOP ensures that when a mapping table entry is being removed during the GC, a new mapping to the most recent version of the corresponding cache



(a) OOP region organization



(b) Data memory slice layout

Fig. 5: Layout of the OOP region. HOOP organizes the OOP region in a log-structured manner. Each OOP block consists of memory slices with a fixed size. There are two types of memory slices: data memory slice and address memory slice.

Algorithm 1 Garbage Collection in HOOP.

Definitions: Home region: Mem_{home}; OOP region: Mem_{oop}; OOP block: Blk_{oop}; Memory slice bitmap: $Bitmap$; Mapping Table: MT;

for All Blk_{oop} is BLK_FULL in Mem_{oop} **do**
  Read all address memory slices S_{addr}.
  Create a hash map H to hold the data during GC.
  Start from the latest start address $Addr$ in S_{addr}.
  for each start address $Addr$ in *reverse order* **do**
    Read all slices of the committed Tx from Mem_{oop}.
    for all memory slices in the Tx **do**
      Read the home addresses $Addr_{home}$ and $Data$.
      Check if $Addr_{home}$ hits in H.
      if hash entry $elem$ exists **then**
        continue.
      else
        Add $Addr_{home}$, $Data$, and $txID$ to the H.
      end if
    end for
  end for
end for
for All $data$ in H **do**
  Write the $data$ to $addr$ in the Mem_{home}.
  if $addr$ is in MT **then**
    Remove $addr$ entry from MT
  else
    continue.
  end if
end for
Update the header in Blk_{oop}.
Clear the corresponding entry in block index table.

line is still maintained. Therefore, the misses in the LLC will not read stale data. Upon an LLC miss, if the address of the missed cache line is not present in the mapping table, HOOP will first check the eviction buffer. If the missed cache line is not in the eviction buffer, HOOP will read the data from the home region. As HOOP migrates data from the OOP region to the home region at a small granularity during the GC, the required eviction buffer size is small (128 KB by default).

D. OOP Region Organization

HOOP organizes the OOP region in a log-structured manner to minimize fragmentation and enable sequential writes for high throughput. The OOP region is consisted of multiple OOP blocks (2MB per block). The OOP region has a block index table (direct mapping table) to store the index number and start address of each OOP block. This block index table will be cached in memory controller for fast data lookup.

OOP Block: We present the layout of an OOP block in Figure 5a. Each OOP block has an OOP header storing the block metadata. The header consists of (1) an 8-bit OOP block index number; (2) a 34-bit address pointing to the next OOP block; (3) a 2-bit flag denoting the block state (`BLK_FULL`, `BLK_GC`, `BLK_UNUSED`, `BLK_INUSE`). The remainder of an OOP block is composed of memory slices with a fixed size of 128-bytes. The fixed-size memory slices place an upper bound on the worst-case fragmentation which can occur within an OOP block, and HOOP can easily manage OOP blocks with a memory slice bitmap. Further, the 128-byte size of a memory slice means that HOOP is capable of flushing the memory slices to the OOP region using two consecutive memory bursts [22].

Memory Slice: We classify memory slices into two categories: *data memory slices* and *address memory slices*. As shown in Figure 5a, a large transaction can be composed of multiple data memory slices which are linked together. The start address of these linked memory slices is stored in an *address memory slice*. Address memory slices allow GC to quickly identify committed transactions in the OOP region.

We show the internal layout of a data memory slice in Figure 5b. With a total size of 128-byte, each memory slice can hold eight 8-byte words of data updated during a transaction, as well as metadata which is 64-byte in length. Each metadata block contains the reverse mappings (home addresses) of modified data to be used during GC and recovery processes. It also contains an address offset (24-bit) to find the next data memory slice, a transaction ID (32-bit) assigned by the memory controller at the start of a transaction, a bit used to identify the first memory slice in this transaction, a count of the updated words (3-bit) in that slice, and a flag (4-bit) used to identify the state of each slice for GC and recovery.

HOOP can achieve uniform aging of all cache lines within an OOP block. In particular, HOOP persists transaction data in the unit of memory slice. HOOP allocates OOP blocks and memory slices in a round-robin manner. Consequently, all OOP blocks achieve uniform wear.

E. Garbage Collection and Data Coalescing

HOOP performs GC in background. It migrates data within the *OOP region* to their original locations in the *home region*. In GC, we have to overcome two challenges. First, as all updated data are preserved in the OOP region, migrating these old data versions sequentially will cause large write traffic. Second, GC should be crash-safe against system failures.

To overcome the first challenge, HOOP scans the committed transactions in *reverse time order* and applies *data coalescing* to minimize the data migration overhead. It performs GC periodically. We depict the GC workflow in Algorithm 1. First,

HOOP reads address memory slices that have been committed in the OOP region (line 4). And then, HOOP creates a hash map H to store the home addresses and their modified data (line 5). According to the start addresses preserved in the address memory slice, HOOP reads each committed transaction from the OOP block in reverse time order (line 7). For each tuple $<<$*home-region addr, TxID*$>$, *Data*$>$ in the committed transaction, HOOP combines all data with the same address to avoid writing to the same home location multiple times (lines 9-17). Therefore, HOOP only maintains the entries for the latest out-of-place updates in the hash map. Since HOOP conducts GC at OOP block granularity (maximum of 128K addresses), the hash table H uses only 1MB buffer space.

Once HOOP finishes scanning all committed transactions in an OOP block, the data held in the hash map will be migrated to the home region (line 21). During migration, the corresponding cache line address is checked in the mapping table (line 22). If the address hits in the mapping table, the entry will be removed, since its most recent data version has been persisted in the home region (line 23). After restoring all data back to their home-region locations, HOOP updates the OOP block header by setting its state to `BLK_UNUSED` (line 28), and clearing its entry in the block index table (line 29).

As for the second challenge, HOOP ensures crash-safety during GC, because the OOP region is always in a consistent state. When a system crash happens while reading the memory slices (lines 4-10), writing the hash table (line 15), or migrating data (line 21), HOOP can simply replay all committed transactions in the OOP region with data recovery (see details in §III-F), and recover the system to a consistent state.

F. Data Recovery with Committed Transactions

Upon system crashes, HOOP will utilize the out-of-place updated data preserved in the OOP region to recover the system to a consistent state. During recovery, it leverages the operating system to create multiple recovery threads. The recovery thread reads the *block index table* to locate OOP blocks. Each recovery thread would map the memory of these OOP blocks into its address space with *kmap*. Then, all committed address memory slices are read from the OOP region to get the start address of memory slice of committed transactions. Once HOOP collects these addresses, it sorts them in the committed order and distributes these addresses to recovery threads in a round-robin fashion.

Each recovery thread will process its own working set independently. Specifically, each thread scans the committed transactions in the OOP region in a reverse order. The thread reads the data memory slices belonging to the transaction and adds the tuple $<<$*home-region address*, *TxID*$>$, *Data*$>$ into a local hash-map set. It preserves only the value with the largest commit ID (i.e., the latest updates). Once all transactions have been completely processed by the recovery threads, a master thread will aggregate the local hash sets into a global one, preserving only the latest version for each home address by checking the committed transaction ID. Finally, the master thread splits the global hash map and leverages other

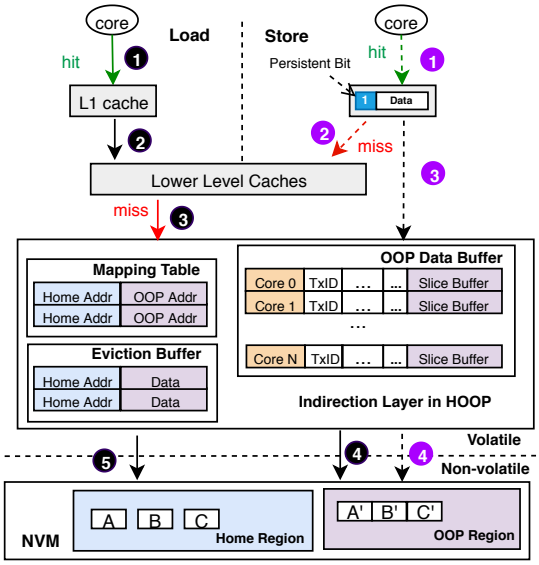


Fig. 6: The load and store procedure in HOOP.

recovery threads to write the data to their home locations in parallel, and to ensure the data durability with cache flush instructions. After that, every recovery thread will unmap the memory mapped space with *kunmap*. The mapping table, eviction buffer, and OOP region are cleared, programs can read the latest data at the home region. Similar to the GC, the data recovery is crash safe as well. HOOP will maintain the committed transactions in the OOP region until the data recovery is finished. When system crashes or failures happen during the recovery, HOOP can restart the recovery procedure.

G. Put It All Together

In this section, we demonstrate how HOOP handles `load` and `store` operations during transaction execution, as shown in Figure 6. The *Tx_begin* instruction sets the transaction state bit for the processor. `store` operations send this transaction state bit to the L1 cache along with the data. The *Tx_end* instruction will clear the transaction state bit and acts like a barrier to ensure durability of the committed transaction. HOOP allows the upper-level programs to handle the transaction concurrency control for flexibility. For instance, applications can use locking or optimistic concurrency control to resolve inter-transaction data dependencies. In this paper, we use the locking mechanism for simplicity. HOOP adds one bit per cache line to indicate whether a cache line has been modified by a transaction or not. This allows HOOP to track the state of these cache lines in the memory hierarchy.

Load Operation: A `load` instruction will read the transaction status bit from the status register in the processor core, thereby specifying the access is to failure-atomic region. This `load` instruction is then added to the load queue while it awaits address generation and disambiguation. Once this load is sent to the L1 cache (❶), a compulsory miss will most likely occur and the cache controller will generate a request to the lower level caches (❷). If there is a cache miss in the

TABLE II: System configuration.

Processor	2.5 GHZ, out-of-order, x86, 16 cores
L1 I/D Cache	32KB, 4-way
L2 Cache	256KB, 8-way, inclusive
LLC	2 MB, 16-way, inclusive
tRCD-tCL-tBL-tWR-tRAS-tRP-tRC-tRRD-tRTP-tWTR-tFAW	
10-10-8-10-24-10-34-4-5-5-20(ns)	
NVM	Read/Write = 50ns/150ns, Capacity: 512 GB Row buffer read/write: 0.93/1.02 pJ/bit Array read/write: 2.47/16.82 pJ/bit [28], [40]

TABLE III: Benchmarks used in our experiments.

	Workload	Description	Stores/TX	Write/Read
Synthetic	Vector [23]	Insert/update entries	8	100%/0%
	Hashmap [24]	Insert/update entries.	8	100%/0%
	Queue [47]	Insert/update entries.	4	100%/0%
	RB-tree [40]	Insert/update entries.	2–10	100%/0%
	B-tree [40]	Insert/update entries.	2–12	100%/0%
Real World	YCSB [23]	Cloud benchmark.	8–32	80%/20%
	TPCC [36]	OLTP workload.	10–35	40%/60%

cache hierarchy (❸), HOOP will use its home-region address to access the address mapping table in the memory controller. In the event of a mapping table hit, the requested data will be read from the OOP region (❹) and home region (❺) in parallel [20], since only the updated data is packed in OOP region. As the OOP address stored in the mapping table can either points to a location in the OOP data buffer, or an OOP block in NVM, HOOP can always obtain the latest version of the updated data. Since each memory slice packs maximum eight cache lines, the unpacking procedure introduces trivial overhead (a few cycles) by traversing the metadata cache line (see Figure 5b). As applications usually have access locality, the data with continuous addresses could be updated inside the same transaction, and thus packed in the same memory slice, which further reduces the unpacking overhead. With the original data in the home region, HOOP can reconstruct the latest version of the cache line with low overhead. In the event of a mapping table miss, HOOP will first check whether the requested cache line is cached in the eviction buffer or not. If yes, HOOP will directly load the data from the eviction buffer. Otherwise, the cache line will be fetched from the home region using the home-region address (❺).

Store Operation: We show the `store` operation in Figure 6. If the `store` (❶) has a cache miss in the L1 cache, the cache coherence mechanism will fetch the cache line in the cache hierarchy (❷). Eventually, the latest version of the cache line will be retrieved from another cache or NVM. Once the cache line is loaded into the L1 cache, it will be updated and the *persistent bit* in the cache line will be set. Because the vast majority of L1 caches are virtually-indexed and physically-tagged (VIPT), the TLB will perform the virtual-to-physical address translation and then return the physical address to the L1 cache. As a result, the cache controller will send the modified data and its home-region address to HOOP (❸).

HOOP stores the updated data in the OOP data buffer. The metadata content in the OOP data buffer will also be updated. In particular, a transaction ID (*TxID*) will be assigned by the memory controller. Other metadata like the home-region address and slice count are also stored in the OOP data buffer. If a transaction has filled the buffer, HOOP will allocate a free memory slice and persist the memory slice in NVM (❹). At the end of a transaction, the processor executes the *Tx_end* instruction, and HOOP ensures all data in the OOP data buffer is flushed to the OOP region.

H. HOOP Implementation

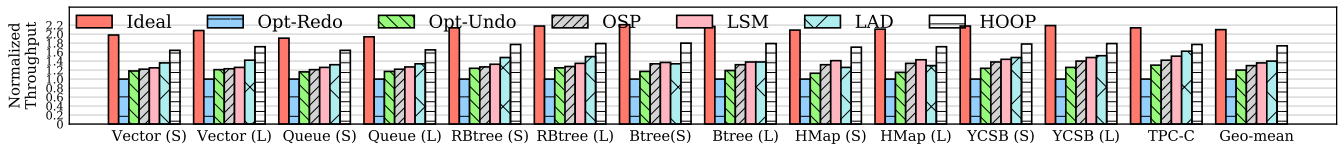
We implement HOOP in McSimA+, a Pin-based many-core simulator [5], with the combination of an NVM simulator. We configure the simulator to model an out-of-order processor with 16 cores and NVM. The detailed system configuration is described in Table II. We use 512 GB of NVM in our experiment, and 10% of its capacity as OOP region by default. The GC in HOOP will execute periodically (in every ten milliseconds by default). Its read and write latencies are configured as 50 ns and 150 ns, respectively. We will vary the NVM latency and bandwidth in our sensitivity analysis. HOOP requires minimal modifications to the memory controller, with the integration of a mapping table (2MB), an OOP data buffer (1KB per core), and a cache-line eviction buffer (128KB). HOOP requires one persistent bit per cache line to track the cache lines that need to be persisted in NVM. We use CACTI 6.5 [35] to estimate the area cost of HOOP. Based on the Sandy Bridge processor package (64KB L1 cache and 256KB L2 cache per core, 20MB LLC, and integrated memory controller), we model the area overhead with the increased buffer size. HOOP introduces only 4.25% area overhead. According to a recent study [51], the released Intel 3D XPoint DIMM has employed a buffer in its memory controller, which makes us believe HOOP is a practical solution.

I. Discussion and Future Work

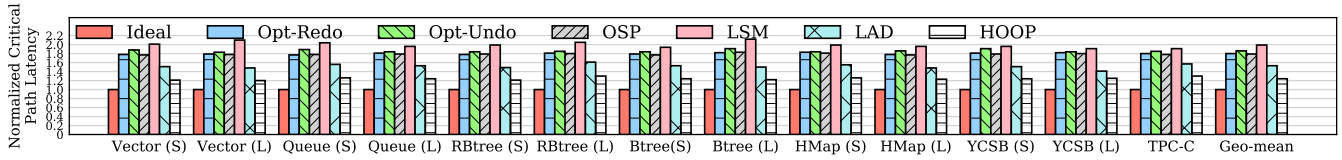
HOOP can be extended to support multiple memory controllers with the two-phase commit protocol [16]. In the Prepare phase, the cache controller will send the modified data in a transaction to the OOP data buffer. When the processor executes the *Tx_end* instruction, the cache controller waits for all outstanding flushes to be acknowledged by the memory controllers. In the Commit phase, the cache controller sends the commit message with the transaction identity to all memory controllers. Memory controllers will acknowledge the received commit messages and ensure the corresponding data in the OOP data buffer is flushed to the OOP region. As for data recovery, once multiple memory controllers reach a consensus regarding the committed transactions through defined communication protocols, HOOP can recover the data to a consistent state by checking the OOP blocks in reversed time order. Moreover, to reduce the mapping table size in HOOP, we can condense multiple mapping entries into one by exploiting the data locality [12]. We wish to explore this in the future.

IV. EVALUATION

Our evaluation demonstrates that (1) HOOP significantly improves the transactional throughput for NVM system (§IV-B); (2) It reduces critical-path latency (§IV-C) and write traffic (§IV-D) by avoiding the extra logging; (3) It suffers from



(a) Transaction throughput (higher is better)



(b) Critical path latency (lower is better)

Fig. 7: Transaction throughput and critical path latency for system benchmarks. HOOP improves transaction throughput by 74.3%, 45.1%, 33.8%, 27.9%, and 24.3% compared with Opt-Redo, Opt-Undo, OSP, LSM, and LAD, respectively. For critical path latency, HOOP also achieves a critical path latency close to a native system without any persistence guarantee.

minimal GC overhead (§IV-F) in NVM, and (4) conducts data recovery (§IV-G) instantly by exploiting multi-core processors; (5) HOOP approach also scales for future NVM (§IV-H).

A. Experimental Setup

We evaluate HOOP with a set of synthetic workloads and real-world applications as shown in Table III. In the experiment, we run eight threads for each workload. For synthetic workloads, we issue insert and update operations randomly against five popular data structures, such as vector, hashmap, queue, RB-tree, and B-tree using transactions, respectively. Each workload has two different data sets consisted of 64 bytes and 1 KB items, respectively. For real-world benchmarks, we run two typical workloads from the WHISPER benchmark suite [36]: YCSB and TPC-C. We use an N-store [7] database as the back-end store, where each thread executes transactions against its database tables. In YCSB, the ratio of reads to updates is 20:80, following the Zipfian distribution [11]. And each key-value pair size is 512 bytes and 1 KB, respectively. In TPC-C, we use its new order transactions which are the most write intensive workloads.

We compare our approach with several state-of-the-art solutions [13], [17], [24], [38], [39]. Specifically, we use four optimized crash-consistency techniques: redo logging, undo logging, shadow paging, and log-structured memory.

- **Opt-Redo:** We implement hardware-based redo logging following the work [13]. It supports asynchronous data checkpointing, log truncation, and combination. After checkpointing the data, it performs in-place update, and truncates logs.
- **Opt-Undo:** We implement hardware-based undo logging based on the work ATOM [24]. It enforces the log and data ordering in the memory controller to reduce the critical-path latency of persistence operations.
- **OSP:** We implement optimized shadow paging based on SSP [38], [39]. In the shadow paging scheme, each virtual cache line is associated with two cache lines, and page consolidation scheme is used to improve the spatial efficiency.
- **LSM:** We implement the log-structured NVM based on the prior work LSNVMM [17]. We implement its address mapping tree using skip list [3], and cache it in DRAM for fast index lookup. For fair comparison, we conduct GC operations in LSNVMM at the same frequency as HOOP.
- **LAD:** We implement the logless atomic durability based on the work LAD [16]. It caches the updates from a transaction in the memory controller until they are committed to NVM.

B. Improving Transaction Throughput

We show the normalized throughput of running each benchmark in Figure 7a. We use the hardware-based optimized redo logging as the baseline. HOOP performs better than all other five state-of-the-art approaches, while ensuring crash consistency. Specifically, HOOP improves transaction throughput by 74.3%, 45.1%, 33.8%, 27.9%, and 24.3% compared with Opt-Redo, Opt-Undo, OSP, LSM, and LAD, respectively. Compared with a native system without any persistence support (Ideal in Figure 7), HOOP delivers 20.6% less throughput.

Opt-Redo persists both the data and metadata for a single update using two cache lines, which wastes memory bandwidth. Compared with Opt-Redo, Opt-Undo maintains the ordering of data and undo log in the memory controller, which reduces the persistency overhead. LAD removes the logging overhead, however, it persists updated data at cache-line granularity. HOOP uses a word granularity for data packing, in which eight data updates and their metadata can be packed into two cache lines. Thus, HOOP can consumes less memory bandwidth, compared with the schemes without data packing.

HOOP outperforms OSP by 33.8%. OSP applies a lightweight copy-on-write mechanism to address the write amplification issues caused by page-level shadow copying. However, OSP may suffer from three performance issues. First, to enforce the transaction durability, it must persist the updated cache lines frequently. This eager persistence greatly affects the transaction throughput. Second, updating the virtual-to-physical address mapping frequently during transaction execution would cause frequent TLB shootdowns on multicore machines. Third, page consolidation in the optimized shadow paging approach incurs addition data copy overhead.

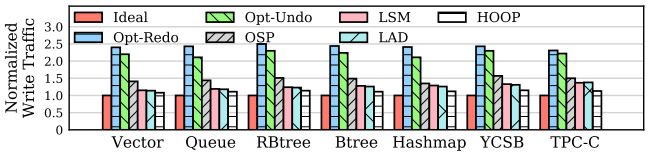


Fig. 8: Write traffic produced by different approaches.

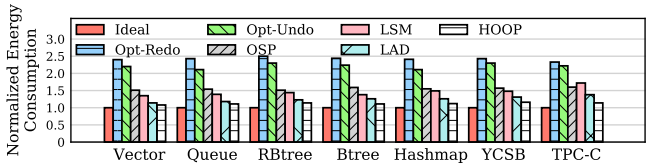


Fig. 9: Energy consumption of different approaches.

As discussed in § II, LSM uses a software-based approach to log the data updates, and it leverages an index tree for address mapping, which incurs significant index lookup overhead. HOOP utilizes the data packing to further reduce the write traffic to NVM, resulting in 27.9% improvement of overall performance, compared with LSM. We next show how HOOP improves the critical-path latency for each transaction.

C. Reducing Critical-Path Latency

We define the critical path latency as the time taken to execute the entire transaction, starting from the *Tx_begin* to *Tx_end*. We use the critical-path latency of the native system as the baseline, and show the results in Figure 7b.

HOOP achieves a significantly shorter critical-path latency than other approaches. It reduces the critical-path latencies by 45.1%, 52.8%, 44.3%, 60.5%, 21.6% on average, compared with Opt-Redo, Opt-Undo, OSP, LSM, and LAD, respectively. HOOP achieves a critical-path latency close to the native system, being 24.1% longer on average. This is because HOOP leverages the OOP data buffer to persist each data update at small granularity, reducing the persistency overhead.

To further understand the data unpacking overhead in HOOP, we profile the number of memory read operations. The results show that one LLC miss incurs 1.28 memory load operations on average for all these workloads, Note that this analysis includes the synthetic workloads that generate random data access patterns. Our experiments demonstrate that HOOP introduces minimal overhead to the miss penalty of LLC. This is for three reasons. Beyond taking advantage of the access locality of workloads, HOOP provides two mechanisms to achieve reduced read latency. First, its GC will run periodically, after which a LLC miss will directly read from the home region. Second, HOOP will issue read requests in parallel (parallel reads) upon the case that data needs to be read from both the home and OOP region. It is worth noting that the possibility of parallel reads is low (3.4% on average). According to our profiling analysis, these benchmarks have a LLC miss ratio of 12.1% on average, and only 28.3% of the LLC misses will incur parallel reads.

Opt-Undo enforces the log and data ordering at the memory controller, which separates data persistence operations from store operations [24]. However, it still performs worse than Opt-Redo, due to the strict persist ordering between log and data residing in the critical path of transaction execution. Furthermore, the asynchronous log truncation and data checkpointing in Opt-Redo accelerate its critical path execution. OSP delivers a longer critical-path latency than HOOP by 44.3%, due to the expensive TLB shootdown. LSM incurs long critical-path latency, due to the software-based index update and lookup. LAD utilizes the queues in the memory controller to cache the updated data, however, it still persists data at cache-line granularity upon transaction commits.

TABLE IV: Average data reduction in the GC of HOOP.

Tx Num.	Vector	Queue	RBtree	Btree	Hash map	YCSB	TPCC
10^1	29.1%	24.3%	23.5%	26.3%	27.7%	23.2%	24.3%
10^2	50.2%	51.8%	53.4%	48.2%	52.4%	49.6%	50.1%
10^3	74.1%	76.4%	73.5%	70.6%	71.2%	70.1%	72.0%
10^4	85.3%	82.2%	81.1%	83.2%	82.5%	81.3%	83.2%

D. Reducing Write Traffic to NVM

Reducing write traffic is important to extend the lifetime of NVM devices. In this section, we measure the write traffic caused by these crash-consistency techniques. We define the write traffic as the number of bytes written for data persistence on a per-transaction basis. We use the native system without persistence support as the baseline (ideal case). We show the normalized write traffic of various benchmarks in Figure 8.

HOOP delivers the lowest number of NVM writes, compared with other five approaches. Both Opt-Redo and Opt-Undo introduce additional writes for each data update, resulting in heavy write traffic during transaction execution. Opt-Undo mitigates this issue through log removal, generating lower write traffic than Opt-Redo by an average of 9.1%. However, they introduce 2.1× and 1.9× more NVM writes than HOOP.

HOOP has lower write traffic than OSP, LSM, and LAD by an average of 21.2%, 12.5%, and 11.6%, respectively. To further understand why HOOP reduces write traffic, we profile the data size updated by transactions, and the data size migrated in the GC of HOOP. We define the data reduction ratio as the percentage of bytes modified by transactions which are not written back to the home region, due to data coalescing during the GC in HOOP. We measure the average data reduction ratio of HOOP, when varying the number of transactions. We show the profiling results in Table IV. As the number of transactions increases, HOOP reduces more write traffic. When the number of transactions exceeds 10^4, HOOP needs to write only a small portion of data (less than 15%) back to their home region by exploiting the data locality.

E. Improving Energy Efficiency of Using NVM

To quantify the energy efficiency of using NVM, we collect both read and write traffics, and use the energy model for NVM read/write discussed in [28], [40]. We list the energy parameters in Table II, and show the results in Figure 9. HOOP achieves the best energy efficiency, although it could

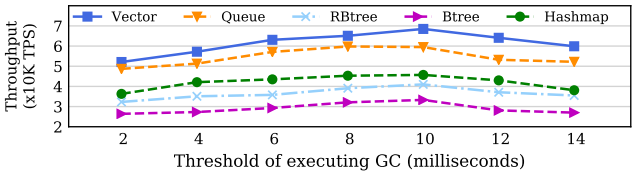


Fig. 10: GC efficiency with different timing thresholds.

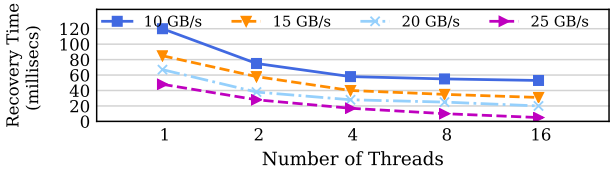
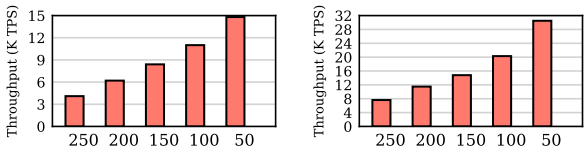


Fig. 11: Recovery performance of 1GB OOP region with various number of recovery threads and memory bandwidth.



(a) NVM read latency (ns). (b) NVM write latency (ns).

Fig. 12: YCSB throughput with various NVM latency.

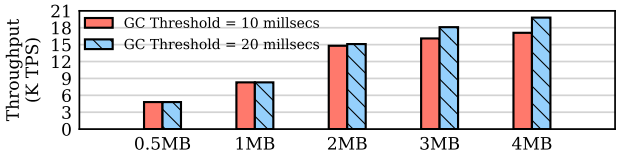


Fig. 13: YCSB throughput with various mapping table size.

incur extra read operations due to parallel reads and GC operations. Compared with the competitive approaches OSP, LSM, and LAD, HOOP reduces the energy consumption by 37.6%, 29.6%, and 10.8% on average.

F. Performance Impact of Garbage Collection

To measure the GC efficiency in HOOP, we vary the triggering threshold from 2 milliseconds to 14 milliseconds. We measure the transaction throughput of the five synthetic benchmarks, and show the results in Figure 10.

As expected, when the period is short, GC is triggered more frequently to migrate updated data from the reserved OOP region to the home region. However, an eager policy like this reduces the possibility of data coalescing. As a result, more NVM bandwidth is consumed by the GC process for writing updated data back to their home locations. And the cycles per transaction is increased by 6.8%–17.8%, as we double the GC frequency. As the trigger threshold becomes longer, the transaction throughput increases constantly. This is because a larger number of data modified by transactions can be coalesced in the reserved OOP region, significantly reducing NVM write traffic. As shown in Figure 10, almost all benchmarks achieve their peak throughput, when the period is about 8–10 milliseconds. When the period exceeds 11 milliseconds, the performance could be constrained by the GC, since there is not enough NVM space to hold committed transactions in the reserved OOP region, and on-demand GC has to take place on the critical path.

G. Fast Data Recovery

To facilitate data recovery upon system crashes or failures, HOOP leverages multiple threads to accelerate recovery procedure. In this experiment, we vary the number of threads performing recovery and the available memory bandwidth to measure the time taken to recover the system state. We show the experimental results in Figure 11.

As the available memory bandwidth increases, it linearly takes less time to recover the system. When the NVM bandwidth exceeds 25 GB/s, it only takes 47 milliseconds for HOOP to recover 1GB of data in the reserved OOP region, which is 2.3× faster than the NVM system with only 10 GB/s memory bandwidth. As the number of recovery threads increases, HOOP scales the data recovery with the parallel scanning of committed transactions in the OOP region. For low-bandwidth NVM, the memory controller becomes the bottleneck, as we further increase the number of recovery threads, which would saturate the memory bandwidth.

H. Sensitivity Analysis

We now perform a sensitivity study of HOOP to understand how various mapping table size and NVM latency will affect its performance. We use YCSB benchmarks that generate a mix of 20% read and 80% update requests against an N-store database. Each key-value pair size is 1KB. We show their results in Figure 12 and Figure 13.

As shown in Figure 12, as we decrease the read latency from 250 nanoseconds to 50 nanoseconds, we keep the write latency as the default value (150 nanoseconds), and vice versa. We observe that HOOP performs better as we decrease the NVM latency. This is expected as the performance of both the `load/store` operation and GC operation will be improved.

As expected, HOOP obtains better performance with a larger mapping table size. When the mapping table size is small, the GC has to be triggered more frequently, because there is not much space to index the out-of-place updates in the OOP region. Based on our experiments, the mapping table with 2MB size provides a reasonable performance. As we further increase the mapping table size, the performance is only slightly increased, because the GC will be executed in every ten milliseconds by default (see § IV-F). Delaying the GC would further increase the application performance as we increase the mapping table size, however, once the mapping table is becoming full, HOOP will kick off the GC. Therefore, consider the tradeoff between the mapping size and GC frequency, we use 2MB mapping table and set the GC frequency as ten milliseconds by default in HOOP.

V. RELATED WORK

Crash consistency for NVM. Many approaches have been proposed to reduce the crash-consistency overheads with

NVM [16], [21], [23], [24], [29], [37], [40], [47]. Mnemosyne defers data checkpointing and log truncation to eliminate them from the critical path of transaction execution [49]. SoftWrAP [14] and DudeTM [29] adopt shadow memory to alleviate the redo logging from the critical path. They keep the data updates in DRAM and persist log entries to NVM asynchronously. Although they reduce the critical-path overhead, persisting log entries still incurs additional write traffic. Furthermore, durable transactions have ordering requirements within and between transactions. Prior work like DCT [27], LOC [32] and HOPS [36] relax this for improved performance. For instance, DCT [27] applies deferred commit to achieve this, HOPS [36] proposes new ISA primitives to decouple the ordering from durability, and BPFS [10] adopts the similar techniques. Our work HOOP has the same goal as those work, and exploits the hardware-assisted out-of-place update in memory controllers to relax the persistence ordering.

Hardware-based logging. Hardware-based logging approaches, such as those for undo logging [24], [37], [47], redo logging [13], [14], [23], [29], [31], [32], [49], and undo+redo logging [40], have been proposed to eliminate the costly cache flushes and enforcement of persistence ordering. However, they inevitably incur additional write traffic to NVM. HOOP reduces the logging traffic significantly with the proposed hardware-assisted out-of-place updates, data packing, and data coalescing techniques, according to our experimental results.

In-place update. Recent work Kamino-Tx [33] and Kiln [54] proposed in-place updates for reducing data persistence overhead. However, supporting in-place updates in modern memory hierarchy is non-trivial. They either require the integration of a non-volatile last-level cache into the chip, or have to preserve a shadow copy for data updates, which incurs significant storage cost. HOOP requires minimal hardware cost by implementing the out-of-place update mechanism in modern memory controllers, and runs optimized garbage collection periodically to reduce the storage cost for the reserved OOP region.

VI. Conclusion

Enforcing data persistence on NVM is expensive. In this paper, we propose a new hardware-assisted out-of-place update approach, named HOOP, to reduce memory persistency overheads. We further improve HOOP with the proposed data packing and coalescing techniques in memory controllers. Our evaluation shows that HOOP achieves a low critical-path latency, which is close to that of a native system providing no persistence guarantee. HOOP also provides up to 1.7× higher throughput and 2.1× less write traffic than state-of-the-art crash-consistency techniques, while ensuring the same strong atomic durability guarantee.

Acknowledgment

We thank the anonymous reviewers for their helpful comments and feedback. We also thank Moinuddin K. Qureshi for an initial discussion on this work. This work was partially supported by NSF grant CNS-1850317 and CCF-1919044.

References

[1] "Libraries and Examples for Persistent Memory Programming," https://github.com/pmem/, 2018.

[2] "Intel Optane DC Persistent Memory," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html, 2019.

[3] "Skip List," https://en.wikipedia.org/wiki/Skip_list, 2019.

[4] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, Providence, RI, USA, 2019.

[5] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software, Austin, TX, USA, 21-23 April, 2013*, 2013, pp. 74–85.

[6] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Write-Rationing Garbage Collection for Hybrid Memories," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*, Philadelphia, PA, 2018.

[7] J. Arulraj, A. Pavlo, and S. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 2015, pp. 707–722.

[8] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *PVLDB*, vol. 8, no. 7, pp. 786–797, 2015.

[9] S. Cho and H. Lee, "Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance," in *Proceedings of 42nd International Symposium on Microarchitecture (MICRO'09)*, New York, USA, 2009.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, Big Sky, Montana, Oct. 2009.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, Indianapolis, Indiana, Jun. 2010.

[12] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, Xi'an, China, 2017.

[13] K. Doshi, E. Giles, and P. J. Varman, "Atomic persistence for SCM with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, 2016, pp. 77–89.

[14] E. Giles, K. Doshi, and P. J. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*, 2015, pp. 1–14.

[15] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09), Washington, DC, USA, March, 2009*.

[16] S. Gupta, A. Daglis, and B. Falsafi, "Distributed Logless Atomic Durability with Persistent Memory," in *Proceedings of 52st International Symposium on Microarchitecture (MICRO'19)*, Columbus, OH, 2019.

[17] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda, "Log-structured non-volatile main memory," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, 2017, pp. 703–717.

[18] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified Address Translation for Memory-mapped SSDs with FlashMap," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, Portland, OR, 2015.

[19] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," in *Proceedings of 41st International Conference on Very Large Data Bases (VLDB'15)*, Kohala Coast, Hawaii, 2015.

[20] Intel, "6th generation intel processor families for s-platforms," *White Paper*, 2018.

[21] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-Atomic Persistent Memory Updates via JUSTDO Logging," in *Proceedings of 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, Atlanta, GA, 2016.

[22] B. Jacob, S. Ng, and D. Wang, *Memory Systems*. Morgan Kaufmann, 2007.

[23] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 178–190.

[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'17), Austin, TX, USA, February, 2017*.

[25] T. Kawahara, "Scalable spin-transfer torque RAM technology for normally-off computing," *IEEE Design & Test of Computers*, vol. 28, no. 1, pp. 52–63, 2011.

[26] 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 (ISCA'17)*, Toronto, ON, Canada, 2017.

[27] A. Kolli, S. Pelley, A. G. 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, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, 2016, pp. 399–411.

[28] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, 2009.

[29] 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 (ASPLOS'17), Xi'an, China, April, 2017*.

[30] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA'19)*, Phoenix, AZ, 2019.

[31] Y. Lu, J. Shu, and L. Sun, "Blurred persistence in transactional persistent memory," in *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST'15), Santa Clara, CA, USA, May, 2015*.

[32] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19-22, 2014*, 2014, pp. 216–223.

[33] 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, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, 2017, pp. 499–512.

[34] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, 1992.

[35] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," *HP laboratories*, 2009.

[36] 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 (ASPLOS'17)*, Xi'an, China, 2017.

[37] T. M. Nguyen and D. Wentzlaff, "Picl: a software-transparent, persistent cache log for nonvolatile main memory," in *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18), Fukuoka, Japan, October, 2018*.

[38] Y. Ni, J. Zhao, D. Bittman, and E. L. Miller, "Reducing NVM writes with optimized shadow paging," in *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018.*, 2018.

[39] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "SSP: Eliminating Redundant Writes in Failure-Atomic NVRAMs via Shadow Sub-Paging," in *Proceedings of 52st International Symposium on Microarchitecture (MICRO'19)*, Columbus, OH, 2019.

[40] M. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+redo logging for persistent memory systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*, Vienna, Austria, 2018.

[41] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14), Minneapolis, MN, USA, June, 2014*.

[42] Ping Zhou and Bo Zhao and Jun Yang and Youtao Zhang, "Energy Reduction for STT-RAM using Early Write Termination," in *Proceedings of 2009 International Conference on Computer-Aided Design (ICCAD'09)*, San Jose, CA, 2009.

[43] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of phase change memories via start-gap wear leveling," in *Proceedings of the 42nd International Symposium on Microarchitecture (MCIRO'42)*, Austin, TX, 2009.

[44] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *36th International Symposium on Computer Architecture (ISCA'09), June, 2009, Austin, TX, USA*.

[45] R. Ramakrishnan and J. Gehrke, *Database Management Systems, 3th Edition*. McGraw-Hill Education, 2002.

[46] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP'91), Asilomar Conference Center, Pacific Grove, California, USA, October, 1991*.

[47] 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 (MICRO'17), Cambridge, MA, USA, October, 2017*.

[48] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *nature*, vol. 453, no. 7191, p. 80, 2008.

[49] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), Newport Beach, CA, USA, March, 2011*.

[50] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, "Espresso: Brewing java for more non-volatility with non-volatile memory," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, Williamsburg, VA, 2018.

[51] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistency memory," *https://arxiv.org/pdf/1908.03583, will appear at FAST'20*, 2019.

[52] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, 2015, pp. 167–181.

[53] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-Efficient Encryption for Non-Volatile Memories," in *Proceedings of 20th International Conferrence on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, Istanbul, Turkey, 2015.

[54] 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 International Symposium on Microarchitecture (MICRO-46)*, Davis, CA, 2013.

[55] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *36th International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, 2009.