# Efficient Hardware-assisted Out-place Update for Persistent Memory

Yifu Deng*, Jianhui Yue*, Zhiyuan Lu*, Yifeng Zhu†
*Computer Science, Michigan Technological University, Houghton, Michigan, USA
†Electrical & Computer Engineering, University of Maine, Orono, Maine, USA
{yifud, jyue, zhlu}@mtu.edu, yifeng.zhu@maine.edu

***Abstract*—Shadow paging can guarantee crash consistency for Persistent Memory (PM). However, shadow paging requires the use of an address mapping table to track shadow pages, and frequent accesses to this table introduce significant performance overhead. In addition, maintaining crash consistency at the granularity level of a page causes a large amount of unnecessary write traffic. This paper proposes a novel hardware-assisted fine-grained out-place-update scheme at the granularity level of a cacheline to efficiently support crash consistency for PM. Our design fully leverages the Address Indirection Table (AIT) available in commodity PM to implement remapping. To ensure the atomicity and durability of AIT updates, we propose two policies: eager persisting and lazy persisting. We also employ overflow log to handle the eviction of speculative AIT cache entries upon an overflow in the AIT cache. Evaluation results based on multicore workloads demonstrate that our proposed scheme can improve the transaction throughput over the state-of-the-art design by 24.0% on average.**

***Index Terms*—Computer Architecture, Persistent Memory, Crash Consistency, Logging, Shadow Paging**

## I. Introduction

Persistent memory (PM) is promising to bridge the gap between memory and traditional storage due to its large capacity, fast speed, non-volatility, and byte addressability. One of the most fundamental challenges of using persistent memory as legacy storage is the support of crash consistency. Crash consistency is to ensure that all data can be recovered to a consistent state in the event of a system crash or power loss.

Crash consistency requires that all updates within a transaction are persisted to PM always in a nothing-or-all manner, even upon a crash. Traditional systems adopt write-ahead logging, such as undo log, redo log, or a combination of both, to guarantee crash consistency. With logging, a transaction update is applied to in-place data only after the log of modification is stored in PM. The logging method suffers from inferior performance due to ordering constraints between logging and in-place update, which places the logging execution in the I/O critical path. To reduce the logging overhead, hardware/hardware-assisted logging methods [1]–[6] have been proposed to move the logging out of the critical path. However, no logging operations are eliminated in both software and hardware approaches.

Shadow paging [7] is another technique to ensure crash consistency. It takes a copy-on-write approach for avoiding in-place updates. Instead, when a data block is modified, another free block in PM is allocated to store the new data. A persistent mapping between physical and device addresses is maintained to track out-place updated data. When a transaction commits, the address mapping table is atomically updated to achieve crash consistency. Conventional shadow schemes often maintain the address mapping at the granularity of a page [8], and have an inferior performance for small writes due to the need of writing unnecessary data. To mitigate this issue, the Shadow Sub-Paging (SSP) [9] reduces the granularity of address mapping to a cacheline. SSP extends TLB by adding the current bitmap and the committed bitmap in a TLB entry to facilitate update atomicity. However, SSP suffers from the overhead of page consolidation when TLB entries are evicted. Moreover, it falls back to the conventional log if the transaction write set is larger than the capacity of TLB. Additionally, due to limited resources of TLB, SSP cannot efficiently support super-page, which is important for emerging big data applications.

In this paper, we introduce a novel hardware-based out-place-update scheme that fully leverages the Address Indirection Table (AIT) available in modern PM to efficiently achieve crash consistency for PM. Our design efficiently supports fine-grained writes and persistence. AIT is designed originally for wear-leveling and bad-block management. We propose to slightly modify AIT to incorporate the address mappings required by out-place updates. We tackle challenges in re-purposing AIT for the address mapping of out-place updates to ensure the crash consistency. After AIT cache entries are modified, the AIT cache buffers both speculative AIT entries and committed AIT entries. Without managing writing these modified cache entries, AIT entries could violate transactions update atomicity in case of a crash.

To address the AIT crash consistency issue, we propose an eager persisting policy that prevents speculative AIT cache entries from evictions and persists the committed AIT cache entries upon committing a transaction. To further optimize persisting performance, we design a lazy persisting policy that delays the persistence of committed AIT cache entries until they are over-written. Finally, we discuss the AIT cache overflow log to handle the speculative AIT cache entries evictions caused by the AIT cache overflow. We evaluate our design and state-of-the-art designs REDU [5] and SSP [9] under multicore workloads. Our evaluation shows that our design out-perform REDU and SSP by 41.2% and 24.0% on average, respectively.

In sum, this paper makes the following contributions:

This research is supported by the NSF grant SHF-1745748 and SHF-1618536. Corresponding author is Jianhui Yue.

- We present a new fine-grained hardware algorithm that leverages AIT in modern PM to efficiently provide crash consistency guaranty.
- We design two persistence schemes: eager persisting and lazy persisting.
- We incorporate the overflow log to solve the crash inconsistency issue caused by the eviction of speculative AIT cache entries due to the AIT cache overflow.

The rest of this paper is organized as follows. Section II discusses crash consistency for PM, and AIT in modern PM. We present our design and evaluation in Section III and IV. Section V summarizes related work, and Section VI concludes this paper.

## II. Background

### A. PM Crash Consistency

Crash consistency ensures that data on PM can be recovered into a consistent state after a system crash or power loss. Prior studies [1]–[3], [5], [10] have proposed multiple logging schemes to achieve crash consistency. Based on log content, these logging schemes can be classified into undo log and redo log. Hardware-assisted logging methods are proposed to improve the crash consistency performance and reduce the burden on programmers. However, these schemes still suffer from (1) handling complicate constrains on ordering write-log and write-data, and (2) generating significant log write traffic and reducing PM's lifetime.

Recently, SSP [9] proposes a shadow page to reduce log operations with a cacheline level mapping, requiring modifications of TLB. SSP flushes a speculative write to an alternative memory location, whose location is tracked by the current bitmap. Upon transaction committing, SSP swaps the commit bitmap with the current bitmap. However, SSP can not totally remove data movements required to maintain transaction atomicity when SSP performs page consolidations, which involves duplicated write operations similar to logging. In addition, the SSP also incurs log operations for the metadata. SSP falls back to logging when the number of updated pages of a transaction exceeds the TLB. Due to its remapping metadata overhead, SSP can not support super-page, which is widely used for the large data set.

### B. Hardware Supports in Commodity PM

Recently, Intel's Optane DIMM provides Asynchronous DRAM Refresh (ADR) [11] and Address Indirection Table (AIT) [12] shown in Fig. 1, to efficiently support PM update atomicity and wear-leveling. ADR leverages energy held in supercapacitors to ensures that all pending write requests received by the memory controller will be persisted to PM in the event of a power failure or system crash. In the memory controller, the queue that buffers all pending write requests (WPQ) is called *ADR buffer*. With the adoption of ADR, it is guaranteed that all write requests admitted in the ADR buffer will be persisted to PM. Thus, the ADR buffer becomes a part of persistence domain. To support wear-leveling, Optane DIMM introduces AIT [12], which records the mapping between physical addresses and PM device addresses. While physical addresses are associated with memory space exposed to ISA, PM device addresses are used by the memory controller to access storage media. The address mapping granularity is 256 bytes. When a block is moved to a different location in PM for wear-leveling, the memory controller updates its address mapping. AIT is stored in PM.

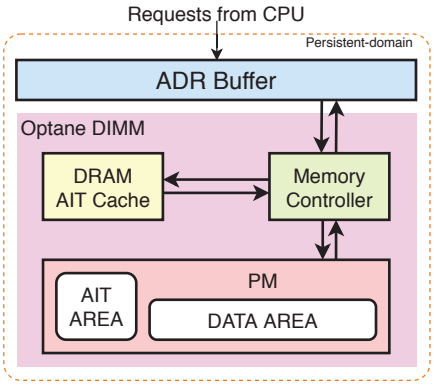


Fig. 1: Intel Optane PM

For each access, the physical address should be translated to a device address. This is done by reading AIT in PM. Since PM is slower than DRAM, Optane introduces AIT cache [12], which is implemented with the DIMM DRAM shown in Fig. 1. When wear-leveling maps a physical block to a new PM location, its corresponding entry in the AIT cache is updated to reflect the new mapping. To avoid losing remapped AIT cache entries in case of a crash, Optane extends the persistence domain to the AIT cache [12]. The super-capacitor in the Optane DIMM has sufficient power budget to flush AIT cache to PM upon a crash. In this paper, we extend the AIT to include the address mapping required by the out-place-update, to more efficiently support crash consistency.

## III. Design

We design a hardware-based algorithm to manage address remapping that supports crash consistency of persistent memory and also allows out-place-updates for data blocks with a fine granularity. In the following, we will first introduce a baseline design, and then present a new scheme called *lazy persisting*. Finally, we propose *overflow logging* to solve the evictions of speculative AIT entries caused by the lazy persist AIT entries.

### A. Baseline Design

We exploit the AIT (Address Indirection Table) technology widely available in modern PM to design a basic out-place-update scheme to support crash consistency. AIT translates an external physical address to an internal device address. Conventionally, AIT is used for wear-leveling and bad-block management in PM. We propose to slightly modify AIT entries to implement the remapping for out-place-updates, as shown in Fig. 2. An out-place-update does not directly overwrite previously committed blocks; instead, new data will be written to another new block in PM.

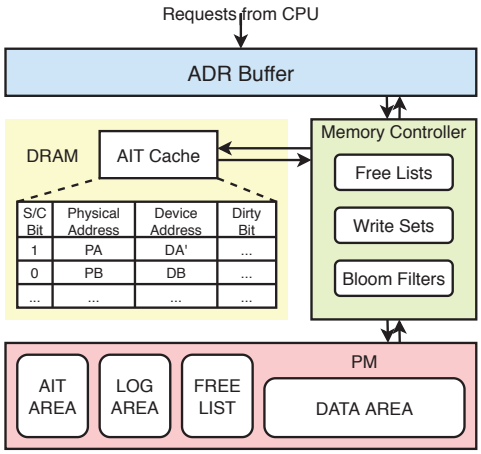


Fig. 2: Architecture

| Requests | AIT Cache | | | AIT | |
|---|---|---|---|---|---|
| store PA<br>store PB<br>store PE | S | PA | DA' | PA | DA |
| | S | PB | DB' | PB | DB |
| | S | PE | DE' | PE | DE |
| COMMIT | C | PA | DA' | PA | DA' |
| | C | PB | DB' | PB | DB' |
| | C | PE | DE' | PE | DE' |

Fig. 3: Example of Eager Persisting

We propose to implement out-place-update via modifying AIT cache entries to remap a physical address to a new device address. When a transaction updates a previously committed block with the physical address $PA$ which is stored at the device address $DA$, a free PM block, assuming its device address is $DA'$, is allocated to store the new data. The AIT cache should be updated to reflect the fact that $PA$ is re-mapped to $DA'$. Before the transaction is committed, there are two AIT entries for $PA$. The new entry stored in the DRAM's AIT cache is called *speculative entry*, and the other existing entry stored in the PM's AIT is called *committed entry*. This AIT modification is compatible with the wear-leveling. Because a write request, which triggers the wear-leveling, updates its AIT entry, and our modification also modifies the AIT entry for such a write write request with the device address assigned by the wear-leveling.

The AIT cache may hold both speculative and committed AIT entries. Each AIT cache entry has a flag bit, named $S/C$ bit, which indicates whether the corresponding AIT cache entry is committed or speculative. If $S/C$ bit is 1, the entry is a committed one. $S/C$ bit is valid only when the entry's dirty bit is set. For an on-going transaction, a update request's AIT cache entries is marked as speculative. When the transaction ends, all of these speculative AIT cache entries are changed to committed, and the corresponding $S/C$ bits are set to 1.

When the AIT cache runs out of space, we cannot evict out a speculative entry and persist it to AIT. Otherwise, the atomicity of transactions is violated and the crash consistency cannot be guaranteed. If a speculative entry is flushed to PM, the corresponding transaction is partially completed. If a crash or power failure takes at this moment, the system cannot be recovered to a consistent state. Therefore, we modify the AIT cache replacement algorithm so that no speculative entries can be chosen as victims.

| Requests | AIT Cache | | | AIT | |
|---|---|---|---|---|---|
| store PA<br>store PB<br>store PE<br>COMMIT | C | PA | DA' | PA | DA |
| | C | PB | DB' | PB | DB |
| | C | PE | DE' | PE | DE |
| store PA<br>store PB<br>store PC | S | PA | DA" | PA | DA' |
| | S | PB | DB" | PB | DB' |
| | S | PC | DC' | PC | DC |
| | C | PE | DE' | PE | DE |
| COMMIT | C | PA | DA" | PA | DA' |
| | C | PB | DB" | PB | DB' |
| | C | PC | DC' | PC | DC |
| | C | PE | DE' | PE | DE |

Fig. 4: Example of Lazy Persisting

To track a transaction's transient speculative writes, the memory controller maintains a write set for an active transaction, which the set of write requests issued by a transaction, so that we can correctly transition the speculative AIT entry to be committed and then empty the set after the committing. For multi-core, the memory controller sets up a dedicated write set for each core.

When a transaction ends, the basic design not only marks its speculative entries as committed but also flushed them to AIT. This is to avoid the scenarios that speculative entries take all AIT cache space. We call this basic design *eager persisting*. We will discuss how to handle rare situations in which AIT cache fails to find a non-speculative victim later.

Fig. 3 uses a simple example to illustrates *eager persisting*. Assume a transaction updates three data blocks with physical address $PA$, $PB$, and $PE$. To perform out-place-update, three free blocks with device address $DA'$, $DB'$, and $DE'$ are allocated for them, respectively. Therefore, their AIT cache entries hold new device addresses, and their AIT entries have old device addresses committed previously. Before this transaction commits, the AIT cache entries for these data blocks are marked as speculative (denoted as $S$). When this transaction commits, these three AIT cache entries are marked as committed (denoted as $C$). At the same time, the memory controller persists these three AIT cache entries into AIT when the transaction ends.

### B. Lazy Persisting

Eager persisting flushes committed AIT cache entries at the transaction commit stage. These flushing operations are on the transaction execution critical path, degrading the system performance. To accelerate a transaction committing speed and reduce flushing overhead, we propose a *lazy persisting* policy that delays the flushing of committed AIT cache entries for any transaction being committed. This effectively moves flushing

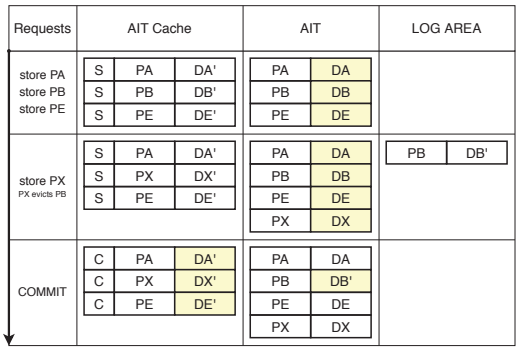


Fig. 5: Example of Overflow Log

operations out of the I/O critical path, without stalling the current transaction. Specifically, we flush the committed AIT cache entries when the committed cache entry is evicted by the cache replacement. At the commit stage, the lazy persisting policy only changes the speculative AIT cache entries, which are indicated by the write set for the committing transaction to be committed. This lazy persisting committed cache entries can guarantee the crash consistency for the corresponding committed transactions. Assume the AIT cache evicts some committed AIT entries of the committed transaction $tx$ to PM, and the remaining committed AIT cache entries of transaction $tx$ still reside in the AIT cache. If a crash occurs, the ADR can safely flush all committed AIT cache entries of $tx$ to PM, because the AIT cache is in the persistence domain, with the help of supercapacitors. After ADR flushes the committed AIT entries to PM, all committed AIT entries are persisted to PM and the update-atomicity of all committed transactions is ensured. This lazy persisting policy not only improves a transaction commit speed but also guarantees crash consistency, by exploiting the AIT cache protected by ADR.

Fig. 4 explains how the lazy persisting policy works under two transactions. When the first transaction commits, we only set the AIT cache entries in the write set to be committed, marked as $C$, rather than flushing them to AIT. Thus, AIT stores the previously committed device addresses for these three entries. During the second transaction execution, the block $PA$ and $PB$ are overwritten. These two speculative writes are directed to the newly allocated device addresses $DA''$ and $DB''$, and their previously committed device addresses $DA'$ and $DB'$ are lazily persisted to AIT. Since the block $PE$ is not overwritten by the second transaction, its AIT cache entry is not written to AIT, shown as $C$ state. After the second transaction commits, its write requests in AIT entries are set to be committed.

### C. AIT Cache Overflow Log

In lazy persisting, the AIT cache may evict speculative entries due to cache overflow. Such evictions could compromise the crash consistency, which is illustrated by an example given in Fig. 5. The transaction $tx$ has four write requests for cachelines $A$, $B$, $E$, and $X$. After $tx$ writes first three cachelines and the memory controller updates their AIT cache entries and marks them as speculative. Later on, $tx$ issues an request to write to $E$. Assume the AIT cache entry for $E$ is missed and loading $E$'s AIT entry to the cache leads to the eviction of the speculative cache entry for $B$. This eviction forces $B$'s speculative entry to be flushed to AIT. If a crash happens immediately after the persistence of $B$'s entry but before $tx$ commits, AIT only reflects that $tx$'s $B$ update is persisted and its update to $A$ and $E$ are lost, which are discarded by the memory controller as being speculative, leading to inconsistency. Note that this cache overflow could happen if a transaction write set is large and the conflict misses of the AIT cache are severe.

To address the crash inconsistency issue caused by the overflow of the AIT cache, we propose *overflow log*, which writes a log for each evicted speculative cache entry, preventing a speculative cache entry from being overwritten in AIT. Specifically, we write log entries to a log region in the PM for evicted speculative AIT cache entries. Each log entry includes the physical address and its corresponding device address. Log entries are organized as a FIFO stored in PM. To provide the latest mapping entry, each AIT cache miss need to check the active transaction's overflow log before reading the AIT. If the AIT request hits the overflow log, the AIT entry should be read from the log. For each AIT cache miss, reading the overflow log introduces extra latency, affecting the performance. To reduce this overhead, we adopt the bloom filter [13] to record the evicted speculative AIT entries' physical address. In case of hitting the bloom filter, we consult the overflow log for the AIT request. Upon a speculative AIT entry eviction, this entry is inserted to the bloom filer. In the previous example, we write an overflow log for $A$'s AIT entry and insert its physical address to the bloom filter. When a transaction commits, we apply its overflow log entries to AIT table in PM, and then clear log entries and its bloom filter. To further improve performance, we set extra small ADR protected log buffer. To support multi-core CPU, each CPU core has its dedicated log FIFO and a bloom filter for its active transaction. The lazy persisting and overflow log are collectively referred to as Efficient Hardware-assisted Out-place-Update (EHOU) in this paper.

We use a simple example to illustrate the overflow log, as shown in Fig. 5. Assume the AIT cache capacity is three entries and this running transaction write set contains four requests, including $PA$, $PB$, $PE$ and $PX$. When the last write request $PX$ arrives, there is no free entry in the AIT cache and the speculative entry $PB$ is evicted. This speculative entry won't be written to AIT. Instead, it is written to the overflow log, shown as the pair $PB$ and $PB'$. During committing, we update $PB$'s entry in the AIT, by applying the overflow log. The log entries are removed after they are applied.

### D. Crash and Recovery

Upon a crash, ADR only flushes all committed entries in the AIT cache to AIT and discards all speculative entries. In this way, we can guarantee update-atomicity for all committed transactions. Discarding speculative AIT cache entries avoids partial updates for the corresponding uncommitted transactions.

Essentially these uncommitted transactions are rolled back. Due to the nature of out-place-updates, serving a write request requires updating its address mapping from the physical address to the device address, and we extend AIT to maintain address mappings between physical addresses and device addresses. When a crash occurs, data blocks for the speculative write requests could be written to PM, while their speculative AIT entries are still in the AIT cache. Discarding their speculative AIT cache entries makes their corresponding data blocks inaccessible, leaking memory space.

To avoid memory space leakage, the allocated block addresses of all speculative AIT cache entries are kept in a reclaim list. When a speculative entry is turned to committed, the corresponding address is removed from the reclaim list. The reclaim list resides in the ADR domain of the memory controller and is atomically durable. A device address is removed from the reclaim list after its AIT entry becomes persisted. If a crash takes place, this reclaim list is appended to the free memory list of the PM. Since the size of the reclaim list is smaller than speculative AIT entries, ADR has a sufficient power budget to flush the reclaim list to the free memory list in PM.

When the system restarts from a crash, the recovery is performed as follows. If there are uncommitted overflow log entries, they are discarded. If there are committed log entries, they are replayed to update the AIT in the PM.

## IV. Evaluation

### A. Experiment setup

| | |
|---|---|
| Cores | 4 OoO core @2GHz,192 ROB entries, 48 STQ entries |
| TLB | L1: 6 sets, 4 ways; L2: 128 sets, 12 ways |
| L1 I/D Cache | private, 32KB, 2 cycles, 8 way |
| L2D Cache | private, 256KB, 8 cycles, 8 way |
| LLC | 8MB, 25 cycles, 16-way, |
| Memory Controller | 1 channel, 1 rank, 8 banks, 8GB PM<br>32 write queue entries, 32 read queue entries |
| PM Access Latency | 300(48) ns write(read) [14], [15] |
| AIT cache | 8192 sets, 16 ways, 100 cycles, 64MB |

TABLE I: System Parameters

The proposed design is implemented and evaluated by using ChampSim [16] with DRAMSim2 [17]. ChampSim is an Intel PIN [18] based simulator that models out-of-order micro-architecture at cycle level with detailed memory access behaviors, including LSQ memory dependence, TLB, and cache miss status holding registers (MSHR). To accurately model PM accesses, the cycle-level DRAMSim2 is incorporated with ChampSim. We enhance ChampSim to support *tx_begin*, and *tx_end*. The configurations of the processor and memory system used in our experiments are listed in Table I. The workloads used for evaluation include Array, Hash Table, Hash Map, B+ Tree, similar to ones used in Ref. [14]. We evaluate workloads B tree and RB tree with keys following Zipfian distribution, similar to SSP [9]. Real-world OLTP workload TATP [19] and TPC-C [20] are evaluated. All workloads run four-copies at the same time.

We evaluated the following designs.

- REDU: We implement the state-of-the-art hardware-assisted redo log with the DRAM cache [5], with models the latency for DRAM cache accesses for each PM read request.
- SSP: It implements the state-of-the-art shadow cacheline SSP [9], modeling TLB consolidation, journaling, and SSP cache.
- EHOU: We evaluate the proposed EHOU with lazy persisting AIT entries and overflow log.

Since the Optane does not disclose its wear-leveling algorithm, we do not model wear-leveling. In addition, each design models the AIT cache and AIT stored in PM.

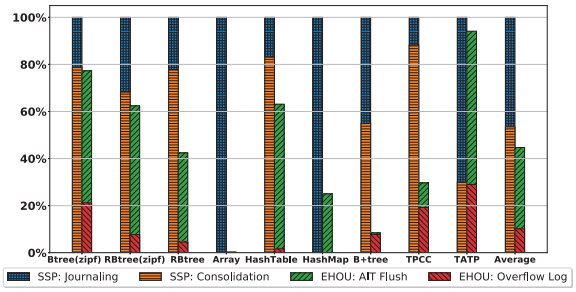


Fig. 6: PM-Write Overhead Reduction

### B. Memory Access Reduction

Fig. 6 compares PM write overhead introduced by SSP and EHOU. The write overhead is defined as the total number of PM write operations excluding those issued by CPU. While the write overhead in SSP is caused by TLB consolidations and metadata journaling, the write overhead in EHOU includes flushing AIT entries and overflow-log. The write overhead of EHOU is rated to SSP. Results show that EHOU can effectively reduce the write overhead by 55.6% on average, compared with SSP. This is because SSP suffers from frequent consolidations when TLB entries are often evicted to free up some cache space. In addition, SSP's TLB consolidation also introduces the same amount of PM read operations as PM write operations, which is one of the major sources of performance overhead. In our design, extra read operations are only caused by the overflow log, which is much smaller than SSP's read operations caused by the TLB consolidations.

### C. Transaction Throughput Improvement

Fig. 7 shows the transaction throughput improvement of SSP and EHOU, compared with REDU. Throughput is number of transaction committed in an unit time. First, EHOU achieves the highest throughput improvement. For example, under the workload ArrayStrm, the throughput of EHOU is 83.1% higher than REDU, and 70.7% higher than SSP. Across all workloads studied, the throughput of EHOU is 41.2% higher than REDU, and 24.0% higher than SSP, on average. EHOU's performance gain is caused by the reduction of both PM write and read operations, which is explained in the last paragraph. In addition, both SSP and EHOU achieve higher throughput than REDU, due to the elimination of frequent log operations.

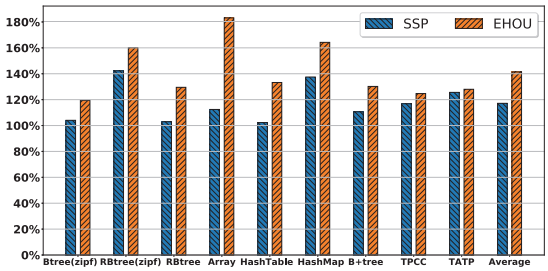


Fig. 7: Throughput Improvement Rated to REDU

## V. Related Work

Software log increases programmers' burden and introduces performance overhead caused by synchronizations between log and update. Hardware logging approaches have been proposed to address software log limitations. ATOM [1] proposes a hardware undo-logging design, which moves the logging out of the critical path, and reduces the undo logging metadata overhead. In order to improve in-place update performance of redo log, REDU [5] proposes the hardware redo-logging which stores transaction updates in a large DRAM-cache, and flushes them to PM, rather than reading from the log area in the slow PM. In addition to ensure data crash consistency, PM encryption requires maintaining recoverability of encryption metadata [14], [21].

Shadow page is another approach to achieve crash consistency. It adopts the copy-on-write (CoW) technique to make a shadow page for an update write request. Updated data is written to the shadow page, without overwriting the existing page committed previously. Conventional shadow schemes maintain address mapping at the granularity of a page [8], suffering from fine-granularity writes. SSP [9] proposes a novel shadow paging to reduce log operations with an interesting mapping at the cacheline level, and it requires the modifications of TLB. However, SSP cannot totally remove data movements required to maintain transaction atomicity when SSP performs page consolidations, which involves duplicated write operations similar to logging. In addition, SSP also incurs log operations for metadata. Lastly, SSP falls back to logging when the number of updated pages of a transaction exceeds the TLB capacity. Due to limited resources of TLB, SSP fails to support super-page, which is important for emerging persistent memory for big data applications. The limited address mapping tables proposed in prior research works poses challenges to efficiently support crash consistency. This paper proposes a novel out-place-update design that re-purposes AIT available in modern PM for address mapping. This significantly reduces the performance overhead caused by the limited address mapping table.

## VI. Conclusion

Shadow paging is a copy-on-write technique that can support crash consistency. However, for PM, shadow paging has two key issues: (1) frequent accesses to the mapping table cause significant overhead, and (2) page-level granularity causes unnecessary writes. This paper proposes a new hardware-assisted out-place update schedule to address the weaknesses of shadow paging. Specifically, we leverage the Address Indirection Table (AIT) available in modern PM to implement fine-grained addresses remapping to support crash consistency. We also designed two policies, including eager persisting and lazy persisting, to better utilize the AIT cache to achieve performance gain, without compromising crash consistency. We compare our design with two state-of-the-art hardware-assisted schemes, REDU and SSP. The evaluation results demonstrate that our proposed scheme can improve the transaction throughput by 24.0% on average.

## References

[1] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: atomic durability in non-volatile memory through hardware logging," in *HPCA*, 2017, pp. 361–372.
[2] K. Doshi, E. Giles, and P. J. Varman, "Atomic persistence for SCM with a non-intrusive backend controller," in *HPCA*, 2016, pp. 77–89.
[3] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for NVM," in *MICRO*, 2017, pp. 178–190.
[4] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *ISCA*, 2017, pp. 175–186.
[5] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *MICRO*, 2018, pp. 520–532.
[6] Z. Lu, J. Yue, Y. Deng, and Y. Zhu, "Improving the performance of nvm crash consistency under multicore," in *The 38th IEEE International Conference on Computer Design(ICCD)*, 2020.
[7] M. M. Astrahan and etc., "System r: Relational approach to database management," *ACM Trans. Database Syst.*, vol. 1, no. 2, p. 97–137, Jun. 1976.
[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.
[9] 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 *MICRO*, 2019.
[10] T. Nguyen and D. Wentzlaff, "PiCL: A software-transparent, persistent cache log for nonvolatile main memory," in *MICRO*, 2018, pp. 507–519.
[11] D. Mulnixl. Intel Xeon processor D product family technical overview. https://software.intel.com/en-us/articles/intel-xeon-processor-dproduct-family-technical-overview/.
[12] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *FAST*, 2020, pp. 169–182.
[13] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, 1970.
[14] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *HPCA*, 2018, pp. 310–323.
[15] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *ISCA*, 2019.
[16] Champsim. https://github.com/ChampSim/.
[17] Dramsim. https://github.com/umd-memsys/DRAMSim2.
[18] 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 *PLDI*, 2005.
[19] A. W. Markku manner Vilho Raatikka Simo Neuvonen. TATP telecommunication application transaction processing (benchmark description). http://tatpbenchmark.sourceforge.net/TATP-Description.pdf.
[20] Transaction processing performance council (TPC), TPC-C. http://www.tpc.org/tpcc/default.asp.
[21] Z. Zhang, J. Yue, X. Liao, and H. Jin, "Efficient hardware-assisted crash consistency in encrypted persistent memory," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020.