Write Prediction for Persistent Memory Systems

Suyash Mahar[*]
UC San Diego
San Diego, CA, USA
smahar@ucsd.edu

Sihang Liu
University of Virginia
Charlottesville, VA, USA
sihangliu@virginia.edu

Korakit Seemakhupt
University of Virginia
Charlottesville, VA, USA
korakit@virginia.edu

Vinson Young
Microsoft
Redmond, WA, USA
vinson.young@microsoft.com

Samira Khan
University of Virginia
Charlottesville, VA, USA
samirakhan@virginia.edu

***Abstract*—Persistent memories (PMs) provide byte-addressable persistent storage with an access latency close to DRAM. Programs can achieve better performance by directly managing persistent data in PM. Unfortunately, these programs need to explicitly order in which data becomes persistent to PM for recoverability. Such enforced orderings place the write-back latency on the critical path of the execution. On the other hand, PM systems integrate different operations to support security, bandwidth reduction, endurance, etc. These PM-support operations further increase the write latency as they first transform data (e.g., encrypt and compute integrity tree) before persisting it to PM. Prior work proposed to use software hints to pre-execute these operations but can increase programmers' burden. The goal of this work is to mitigate the overhead due to PM-support operations in a *software-transparent* manner.**

If we can determine the address and value of a PM write-back ahead of time, it is possible to precompute its PM-support operations and move them off the critical path. We observe that both the address and data of PM write-backs often appear as values in prior store-instructions during the execution. For example, a PM allocator first uses a store-instruction to assign the allocated address to a pointer; later, the follow-up procedure updates that allocated address with new values and performs a write-back to PM. In this example, the value carried by the PM allocator's store-instruction contains the address of the later PM write-backs. Likewise, the data values also often appear in prior store-instructions, e.g., when passing values to a PM update function. Therefore, it is possible to predict the address and data, and precompute the PM-support operations for future write-backs by tracking store-instructions. We propose PMWeaver that learns the correlation between a PM write-back and the associated store-instructions and *weaves* together the address and data value of a PM write-back from the correlated store-instructions. Our evaluation demonstrates that PMWeaver correctly predicts 81.16% of addresses and 49.90% of data of PM write-backs in ten PM workloads, and yields 1.63× and 1.26× speedup over a no-prediction baseline system by precomputing two types of PM-support operations: a combination of encryption and integrity verification, and deduplication.

***Index Terms*—persistent memory, prediction**

I. Introduction

The advancement of persistent memory (PM) technologies, such as Intel's Optane DC Persistent Memory [1], provides fast and byte-addressable access to persistent data. Programs can directly manipulate persistent data in PM through a load/store interface. Various software systems have been developed and deployed to leverage the benefits from the unified memory and storage systems. Examples include PM-optimized databases [2]–[6], file systems [7]–[11], and transactional libraries for various data structures [12]–[14]. These PM-based software systems tend to bypass the OS indirections (e.g., file systems) and directly manage persistent data in memory for better performance. As a result, these systems cannot rely on the OS to recover to a consistent state after a failure.

The burden of maintaining the recoverability of persistent data across system reboots or unexpected failures (i.e., crash consistency) now lies on the programmers. Programs need to manage writes to PM carefully using write-back (e.g., `CLFLUSH` and `CLWB`) and fence (e.g., `MFENCE` and `SFENCE`) primitives [15]–[26]. With these primitives that ensure orderings, programs typically implement certain crash consistency mechanisms, such as undo/redo logging [8, 12, 21, 27, 28], shadow paging [29, 30], and checkpointing [31, 32]. For example, the undo logging mechanism first creates a log of a persistent object and writes it back to PM *before* any modification to that object occurs. As a result, such write-backs and fences can block the execution and place the write latency on the critical path. On the other hand, a PM system requires not only crash-consistent software, but also additional support on the hardware side. These *PM-support operations* in the memory controller [33] guarantee the security of data using encryption and integrity verification [34]–[40], extend the memory lifetime through wear-leveling mechanisms [41]–[44], and overcome bandwidth limitation by employing data compression or deduplication [45]–[48]. These seemingly different operations exhibit one common characteristic—they all take place *before* data is written back to PM, e.g., encrypting a cache line *before* it is written back to PM. These operations are not only *computation heavy* but also *introduce additional memory accesses*. For example, an update to a Merkle Tree for integrity verification involves fetching existing nodes from PM and performing a series of hash computations from the leaf to its root [49]. Therefore, PM-support operations can take hundreds of nanoseconds, and their integration significantly increases the write latency and degrades the overall

[*]Suyash Mahar contributed to this work during his internship at the University of Virginia.

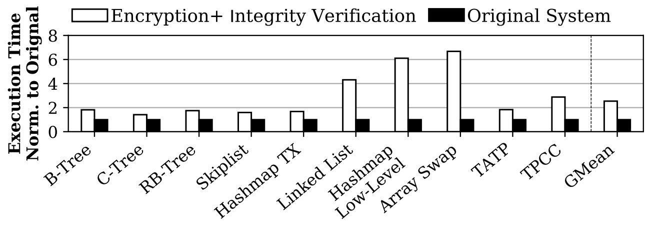
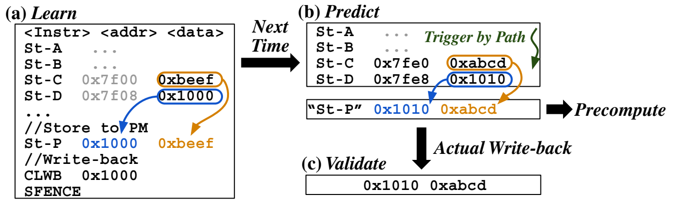


Fig. 1: Slowdown due to encryption and integrity verification over the original system without these PM-support operations.



```
<Instr> <addr> <data>
St-A    ...
St-B    ...
St-C  0x7f00  0xbeef
St-D  0x7f08  0x1000
...
//Store to PM
St-P  0x1000  0xbeef
//Write-back
CLWB  0x1000
SFENCE
```

```
St-A    ...
St-B    ...
St-C  0x7fe0  0xabcd
St-D  0x7fe8  0x1010
```

```
"St-P" 0x1010 0xabcd
```

```
0x1010 0xabcd
```

Fig. 2: High-level idea of PMWeaver.

performance. Figure 1 shows that a combination of encryption and integrity verification in a secure PM system introduces 2.54× slowdown on average (methodology in Section VI-A).

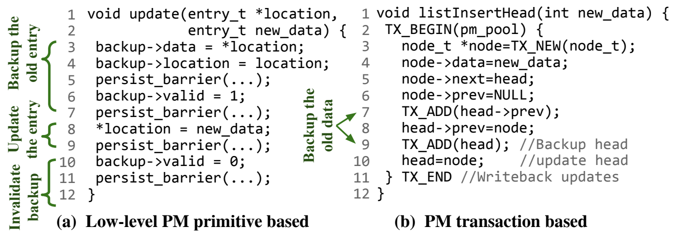
Prior works proposed different optimizations to reduce the latency of PM-support operations [33, 35, 45]–[47, 49, 50]. A recent work reduces this overhead by using software hints to precompute these operations and overlapping the precomputation with program execution [33]. Although effective in hiding the PM-support latencies, such a software-hint-based technique has several limitations. First, instrumenting software hints adds a significant burden on the programmers. Compiler support can reduce the programmer's burden, but it is limited to static information and cannot cover dynamically allocated memory, function calls, and loops. These constraints result in fewer opportunities for optimization and limited speedup from precomputing the PM-support operations. Second, such software hints complicate the ISA and impact the code portability—the code may become incompatible with new PM hardware platforms. Therefore, we argue that a generic precomputation technique for PM-support operations should be software-transparent. Compared to a hardware approach, the precomputation would not be limited by the software context or complicate the programming. The *goal* of this work is to overlap the extra PM-support latencies through hardware-based precomputation, *without* requiring any software modification.

Intuitively, PM-support operations can be initiated as soon as the address and data of a PM write-back become available. Therefore, a naive way to precompute these operations would be to start the computation once an update enters the memory hierarchy. If there was a sufficient time duration between data become available in cache (i.e., via a store) and the later write-back operation (e.g., a sequence of "CLWB;SFENCE" that persists the updates), we could overlap the entire PM-support computation. However, crash consistency mechanisms rely on frequent write-backs, making the time gap short in PM programs. Our evaluation with common PM workloads [12, 21, 33, 51] shows that the average latency for encryption + integrity verification is 610 ns but only 23.1% of the write-backs actually have such long time gaps. The majority of the write-backs (76.9%) need to start the precomputation much earlier to hide the long latency of these PM-support operations. Therefore, we cannot solely rely on precomputing PM write-backs from data in the cache but need an intelligent mechanism to identify the address and data of write-backs way ahead in time.

Predict the PM write-back's address and data based on values from prior store-instructions: The *first challenge* of this work is how the hardware can *identify the address and data* of a PM write-back early enough to hide the latency of precomputation. We observe that a PM write-back's address and data are often carried by prior *store instructions*, long before its actual access to PM. For example, a procedure that appends a new node to a persistent linked list *stores* the new node's address in a pointer and then *stores* the values of each field of the node before persisting the node. As such, the new node's address and data values are available in these store-instructions prior to its write-back to PM that enforces persistence, making it possible to precompute the PM-support operations from these values. Therefore, our key idea is to use values from prior store-instructions for predicting the address and data of future write-backs. Figure 2a shows an example where the address and data of a write-back to PM (CLWB to write-back St-P) can be found in instruction St-D and St-C. By learning this pattern in the *PM-addr/data generation relationship*, we can *predict* address and data values as soon as these store-instructions execute. Thereby, the precomputation can start much earlier.

Trigger prediction according to execution path: The key observation on store-instructions generating addr/data of a future write-back leads to the *second challenge*—how a prediction should be triggered? The prediction can start as soon as it has identified and learned the PM addr/data-generating relationship pattern, i.e., the relationship between a PM write-back and its prior store-instructions. A naive solution could trigger the prediction upon observing the first store-instruction in the learned pattern and wait for the remaining ones to become available. However, the same store-instruction can contribute to many different PM addr/data-generation patterns. For example, a PM allocator is frequently used in different functions and contributes to multiple PM addr/data-generation patterns, and thus, predictions triggered by its internal store-instructions are likely to cause mispredictions. We notice that each write-back to PM usually occurs in a relatively unique context—the sequence of store-instructions that appear before the write-back naturally forms an *execution path*. Thus, our key idea is to trigger the prediction only when the same *execution path* is observed again, as the execution path can help mitigate mispredictions. Figure 2b demonstrates that the prediction is triggered by the path of St-A, St-B, and St-C.

Resolve mispredictions using a validator: Once a prediction is generated, the predicted address and data values can initiate the precomputation. This leads to the *last challenge*—how to ensure that mispredictions can be resolved, *without*



```
 1 void update(entry_t *location,
 2              entry_t new_data) {
 3  backup->data = *location;
 4  backup->location = location;
 5  persist_barrier(...);
 6  backup->valid = 1;
 7  persist_barrier(...);
 8  *location = new_data;
 9  persist_barrier(...);
10  backup->valid = 0;
11  persist_barrier(...);
12 }
```

(a) Low-level PM primitive based

```
 1 void listInsertHead(int new_data) {
 2  TX_BEGIN(pm_pool) {
 3    node_t *node=TX_NEW(node_t);
 4    node->data=new_data;
 5    node->next=head;
 6    node->prev=NULL;
 7    TX_ADD(head->prev);
 8    head->prev=node;
 9    TX_ADD(head); //Backup head
10    head=node;    //update head
11  } TX_END //Writeback updates
12 }
```

(b) PM transaction based

Fig. 3: Examples of PM program using (a) low-level primitives and (b) PM transactions.

impacting the persistency model? We find that the persistency remains unaffected as long as the predicted data and associated metadata for the PM-support operations (e.g., hash values for integrity verification) do not become persistent or change any processor state (e.g., metadata cache). Therefore, our key idea is to have the predictor buffer the precomputed results and metadata and validate them against the actual write-back (Figure 2c).

Putting together the three key ideas above, we propose PMWeaver, a software-transparent mechanism to mitigate the performance overhead of PM write-backs by precomputing the PM-support operations. PMWeaver *learns* the addr/data-generating relationship between a PM write-back and prior store-instructions and *weaves* data values from store-instructions to predict the future PM write-backs.[1] The contributions are the following:

- This is the first work that *transparently* precomputes latency-critical PM-support operations before actual PM write-backs.
- We propose PMWeaver that learns the relationship pattern between a PM write-back and prior store-instructions and predicts the PM address and data values for precomputation from the associated store-instructions.
- Our evaluation shows that PMWeaver correctly predicts 81.16% and 49.90% of the address and data for precomputation, respectively. Compared to a baseline without prediction, it yields a speedup of 1.63× in a secured PM system with a combination of encryption and integrity verification, and 1.26× in a PM system with deduplication.

II. Background and Motivation

In this section, we start by introducing PM programming patterns that ensure data recoverability. Then, we describe the performance overhead of PM-support operations and motivate the need for predicting PM write-backs to move these operations off the critical path.

A. PM Programming Pattern

PM technologies, such as Intel's Optane DC Persistent Memory [1], offer latency benefit close to DRAM while ensuring data persistence, blurring the boundary between memory and storage. PM programs can bypass OS indirections (e.g., file systems) to manage persistent data directly. Because of the direct management, these programs also need to ensure recoverability in case of a failure, which requires writes to

[1]The simulator is available at https://pmweaver.persistentmemory.org.

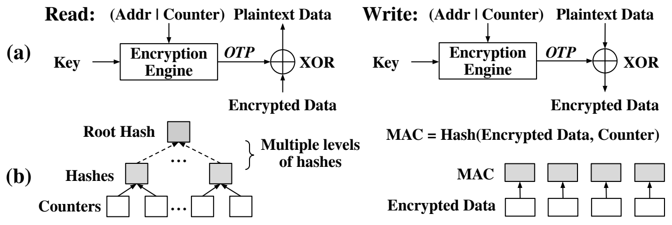


Fig. 4: (a) Encryption and (b) integrity verification.

be properly ordered. The hardware system provides low-level primitives to perform write-back (e.g., `CLWB`) and enforce ordering (e.g., `SFENCE`). For simplicity, we refer to a sequence of "`CLWB; SFENCE`" as a `persist_barrier()` [52]. Next, we describe the use of low-level primitives with an example.

Example based on low-level PM primitives.: Figure 3a shows an `update()` function that modifies a persistent array using low-level PM primitives for crash consistency by taking the following steps: backup the old data (line 3-4), set a valid bit of the backup (line 6), update in-place (line 8), and finally commit the update by invalidating the backup (line 10). In this procedure, the program places a `persist_barrier()` after each step to ensure all updates have been persisted before moving to the next step. This example demonstrates a typical way of programming with low-level primitives to carefully manage ordering and write-back of PM updates. As a result, write-backs happen immediately after the updates to PM objects (i.e., via store-instructions).

Example based on a PM transaction.: Alternatively, there are also libraries that abstract away the low-level primitives. A representative example is using PM transactions. For example, Intel's PMDK library [12] wraps up a transaction of PM updates with a pair of `TX_BEGIN` and `TX_END`, and lets programmers log the original data before modification with methods such as `TX_ADD` (i.e., undo log). Figure 3b shows a `listInsertHead()` function that appends a new node to the head of a persistent linked list. The program wraps the whole procedure inside a transaction, and uses `TX_ADD` for logging, and `TX_NEW` for allocation. These types of programs hide low-level primitives in the library function calls, where logs need to be persisted before the actual in-place update, and all updates need to be persisted at the end of a transaction.

B. PM-Support Operations

Besides the crash consistency guarantee, PM systems also need additional operations in the hardware as it unifies the characteristic of both the memory and storage. Examples include encryption and integrity verification techniques to ensure the security of persistent data [34]–[39], compression and deduplication to increase bandwidth and capacity [45]–[48, 53], and wear-leveling techniques to improve the lifetime of persistent memory [41]–[44]. Next, we describe two examples of PM-support operations.

1) Encryption and integrity verification.: Figure 4 demonstrates counter-mode encryption [54]–[56] and integrity verification [33, 37]–[39] in PM systems. Figure 4a shows the

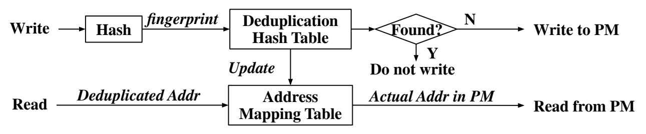


Fig. 5: Deduplication.

decryption and encryption procedure. During a write access, the encryption engine generates a one-time pad (OTP) using a key, a cache line address, and a counter. It then XORs the OTP with the plaintext data to generate the encrypted data, i.e.,

$$\text{EncData} = \text{Enc}(\text{Addr}|\text{Counter}, \text{Key}) \oplus \text{PlaintextData}.$$

Figure 4b shows the process of integrity verification using a Bonsai Merkle Tree [49], where the leaf nodes are the aforementioned encryption counters, and the intermediate nodes are the hashes of their child nodes. i.e.,

$$\text{ParentNode} = \text{Hash}(\text{ChildNode}_1, \text{ChildNode}_2, \ldots).$$

The nodes are hashed level by level, such that the root of the Merkle Tree becomes a hierarchical hash of all the counters (root is stored on-chip to prevent tampering). Modification of any counter will change the hashes and will indicate an unauthorized update. The encrypted data blocks are also encoded into message authentication codes (MACs) to protect data integrity, i.e.,

$$\text{MAC} = \text{Hash}(\text{EncData}, \text{Counter}).$$

The MACs are updated on write and verified on read. However, the memory controller needs to update hash values from the leaf *all the way* to the root during every write access to keep the Merkle Tree up-to-date. These operations are not only compute-intensive, but also introduce additional memory accesses to fetch nodes from PM. Even after hiding some of the latency using a metadata cache, on average, encryption and integrity verification incur 610 ns in our evaluation.

2) Deduplication.: Figure 5 shows the procedure of PM deduplication, which reduces the write bandwidth and increases the effective memory capacity [33, 50, 57]. During a write access, the memory controller first hashes the data block into a *fingerprint*, i.e.,

$$\text{Fingerprint} = \text{Hash}(\text{Data}).$$

Second, it looks up the fingerprint in a *deduplication hash table*. If found, the data turns out to be a duplicate and does not need to be written to memory. If the fingerprint does not match, the write is sent to PM, and the *address mapping table* is updated to store the location of the data block. The deduplication process significantly increases the write latency as it involves additional memory accesses when the lookup in the hash and mapping table misses the metadata cache. Our evaluation shows that deduplication increases the write latency by 334 ns on average.

C. Performance Overhead

As introduced in Section II-A, crash consistency mechanisms in PM programs usually strictly enforces the order in which updates write-back to PM. With the PM-support operations, the write-back latency is significantly increased. Recent PM systems introduce techniques, such as *asynchronous DRAM refresh* (ADR), that moves the write pending queue (WPQ) to the persistence domain using capacitors [58, 59], and thereby, a write-back becomes persistent before entering PM (demonstrated by the timeline in Figure 6a). However, PM-support operations are power-intensive and may introduce additional memory accesses (e.g., upon a counter cache miss during encryption). Consequently, it is not feasible to back all PM-support operations with a small amount of residual energy. As such, these operations must complete before a write becomes persistent, introducing a significant increase in the write-back latency (as shown in Figure 6b).

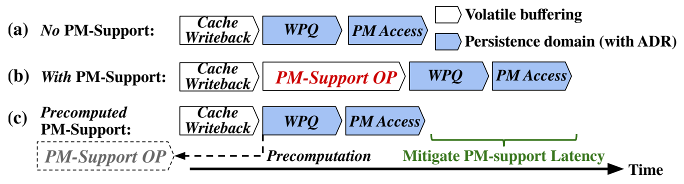


Fig. 6: Breakdown of the write-back latency.

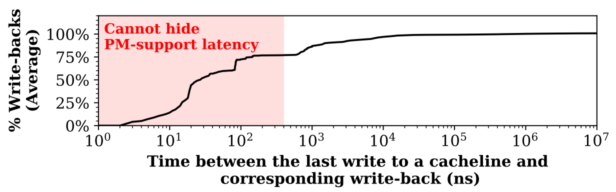
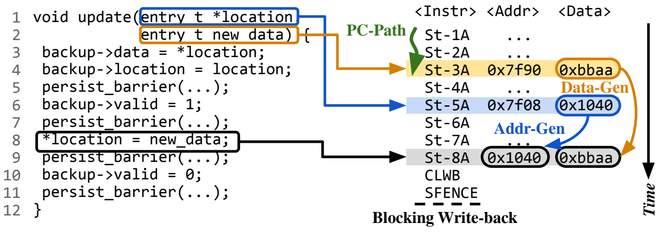


Fig. 7: The time duration of data being available in the cache, before being written-back.

Prior works proposed efficient implementation of these PM-support operations to reduce the write-back overhead [33, 35, 45]–[47, 49, 50]. A recent work [33] further reduces the latency by overlapping the operation with the program execution using software hints. Though this precomputation technique is highly effective in moving the overhead off the critical path, it has several shortcomings. The software hints requirement breaks the existing system abstraction, complicates the ISA, and degrades software's portability and programmability. Even with compiler support, the software-directed method is still largely limited by the static information—cannot adapt to dynamically allocated memories, function and loop boundaries, and control flows. Ideally, the mitigation should be transparent without requiring any software modification. The *goal* of this work is to provide a *software-transparent* solution to precompute these PM-support operations off the critical path, as shown in Figure 6c.

D. Need for Write-back Prediction

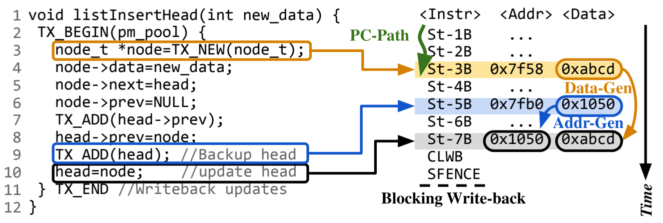
A naive method is to precompute these PM-support operations once data becomes available in the cache hierarchy (i.e., the last store to a cache line has reached the L1 cache), as data enters the cache before being written back. If the duration of time—from data becomes available in the cache to its write-back—is sufficient for PM-support operations to complete, their latencies can be overlapped. Figure 7 characterizes the write-back behavior of ten common PM workloads (methodology in VI-A). The X-axis shows the time during which data remains



```
1  void update(entry_t *location
2               entry_t new_data) {
3   backup->data = *location;
4   backup->location = location;
5   persist_barrier(...);
6   backup->valid = 1;
7   persist_barrier(...);
8   *location = new_data;
9   persist_barrier(...);
10  backup->valid = 0;
11  persist_barrier(...);
12 }
```

```
<Instr> <Addr>  <Data>
St-1A     ...
St-2A     ...
St-3A   0x7f90  0xbbaa
St-4A     ...
St-5A   0x7f08  0x1040
St-6A
St-7A     ...
St-8A   0x1040  0xbbaa
CLWB
SFENCE
```

Fig. 8: Execution trace in the example of Figure 3a.



```
1  void listInsertHead(int new_data) {
2   TX_BEGIN(pm_pool) {
3     node_t *node=TX_NEW(node_t);
4     node->data=new_data;
5     node->next=head;
6     node->prev=NULL;
7     TX_ADD(head->prev);
8     head->prev=node;
9     TX_ADD(head); //Backup head
10    head=node;    //update head
11  } TX_END //Writeback updates
12 }
```

```
<Instr>  <Addr> <Data>
St-1B     ...
St-2B     ...
St-3B   0x7f58  0xabcd
St-4B     ...
St-5B   0x7fb0  0x1050
St-6B     ...
St-7B   0x1050  0xabcd
CLWB
SFENCE
```

Fig. 9: Execution trace in the example of Figure 3b.

available in the cache. And, the Y-axis shows a CDF of data write-backs (average over these workloads) that happen within different time durations. On average, 76.9% of the write-backs happen within the average PM-support latency (610 ns of encryption and integrity verification latency), as indicated by the red region. Therefore, a large fraction of write-backs do not have a sufficient time duration to precompute, as their time gap between data becomes available in the cache, and its write-back is less than the average PM-support latency. We conclude that most of the write-backs need to be *predicted* earlier in order to overlap the latency of these PM-support operations. Hence, the rest of this paper will be focusing on the prediction of the address and data for the majority of the time-critical write-backs (Section IV and V). In addition, we provide a simple but effective mechanism that covers the remaining write-backs that can be precomputed using data from the cache (Section V-D2).

III. PM Write Prediction

So far, we have discussed the performance overhead of PM-Support operations. To enable early precomputation of these operations, we propose PMWeaver, a hardware-based prediction mechanism that transparently and timely mitigates the overhead of PM-support operations. Next, we discuss the challenges and our key ideas.

A. Prediction Based on Store-instructions

Challenge: As the workload characterization in Section II-D has shown, the majority of write-backs require their address and data to be directly predicted, way before the update to PM has happened. Therefore, the first challenge is how PMWeaver can obtain both values early enough to hide the latency of PM-support operations.

Key observations: Let us revisit the examples in Section II-A. Figure 8 shows the execution trace in the example of Figure 3a. Line 8 is a performance-critical write to PM, where the program updates `new_data` to `location` and then immediately writes it back using a `persist_barrier()`. We observe that the value of both *address* and *data* of this

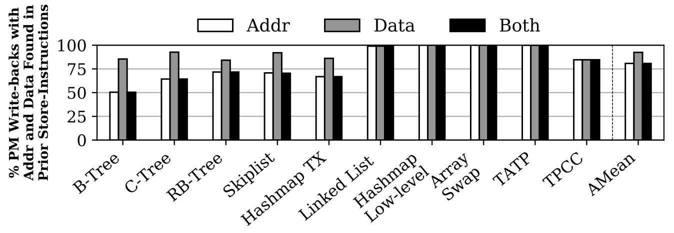


Fig. 10: The percentage of PM write-backs with address/data generated by prior store-instructions with an *ideal predictor*.

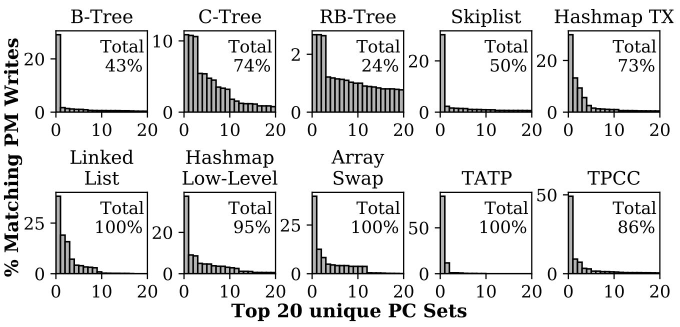


Fig. 11: Distributions of PC sets that generate data and address.

write-back have appeared in prior store-instructions (`St-5A` and `St-3A`) when `update()` is being called. Figure 9 shows the execution trace in the example of Figure 3b. Similar to the first example, the in-place update to the `head` pointer at line 9 is performance-critical as it is immediately written back at the end of the transaction (line 11). We observe that line 3 (`St-3B`) stores the *data value* of the new pointer and line 9 (`St-5B`) stores the *address value* of the existing `head` to the log. Therefore, we conclude that store-instructions prior to a PM write-back may generate its address and data values. We refer to such a correlation pattern between store-instructions and the later PM write-back as the *addr/data-generating relationship*. We further find that this relationship has two characteristics.

1) Addr/data-generation relationship is frequent: We first analyze the percentage of PM write-backs (cache-line-sized) that have data/address found in prior stores by profiling ten PM workloads. Figure 10 demonstrates the statistics using an *ideal* predictor that does not limit the search scope, where the address- and data-matching take a granularity of 8B and 4B, respectively. On average, 80.8%/92.4%/80.7% of PM write-backs have their address/data/both generated by prior store-instructions. Therefore, store-instructions frequently generate the address and data values of future PM write-backs.

2) Addr/data-generation relationship is stable: We then observe that the addr/data-generation relationship is stable as the same PM update procedure usually executes repeatedly, such as `update()` and `listInsertHead()` from the examples in Section II-A. Figure 11 shows the top-20 sets of unique addr/data-generating instructions (i.e., their PCs) in these ten workloads. We observe that they mostly follow a Pareto distribution—a small number of PCs generate the majority of the PM write-backs (represented by the total gray area in each sub-figure). On average, the top-20 repeated PC sets generate 68.6% of PM write-backs.

Key Idea: The observations above shows that the addr/data-generating relationship *frequent* and *stable*. Therefore, our key idea is predict the address and data of PM write-back based on store-instructions. First, PMWeaver records the addr/data-generating store-instructions. Later, when observing the same pattern of instructions, PMWeaver *weaves* together the values from these store-instructions to generate a prediction.

B. Prediction Triggering According to Execution Path

Challenge: With the addr/data-generating instructions identified, the next step is to trigger a prediction as soon as those store instructions are executed. A naive solution is to trigger the prediction when observing the oldest addr/data-generating instruction. However, a store-instruction can be executed in multiple contexts and therefore can contribute to multiple predictions. In example of Figure 9, the prediction will be triggered upon observing the oldest store-instruction, `St-3B`. Because this instruction is part of a PM allocation function (wrapped in `TX_NEW()`) that can be shared among different procedures, such a naive method can lead to inaccurate predictions. Thus, precisely triggering the prediction is the second challenge.

Key idea: We observe that, although an addr/data-generating instruction alone can be common to multiple predictions, the context it resides remains mostly unique. For example, in Figure 8, the store-instructions, `St-1A` and `St-2A`, executed prior to `St-3A` provide a unique context leading to this specific addr/data-generating pattern (marked with a green arrow). Such context can be represented as a sequence of store-instructions (i.e., their PCs) that happen prior to the oldest addr/data-generating store-instruction, which we refer to as a PC-path. Similarly, a sequence of `St-1B`, `St-2B`, and `St-3B` is a PC-path in Figure 9. In summary, our key idea is to trigger a prediction based on the PC-path.

C. Misprediction Handling

Challenge: Though write-backs are predictable, a predictor by itself does not guarantee correctness. Mispredictions can lead to incorrect values written back to PM, and cause other predictions that use the misprediction's PM-support metadata to be wrong. Therefore, the last challenge is how PMWeaver can ensure the write-backs, as well as their metadata, are correct in case of mispredictions.

Key idea: Our key idea is to buffer the precomputed prediction results until they have been validated against the actual write-back. PMWeaver stores the predicted write-backs and their associated metadata in a volatile buffer without mutating the persistent state or affecting any other processor states (e.g., registers, and data and metadata caches). Once the actual write-back arrives, the predictor validates the prediction by comparing the predicted addr/data with the actual ones. If the prediction is correct, the predictor can safely update the processor state, and the precomputed results and metadata can be persisted following the original persistency model. Otherwise, it discards the results and recomputes PM-support operations according to the actual write-back.

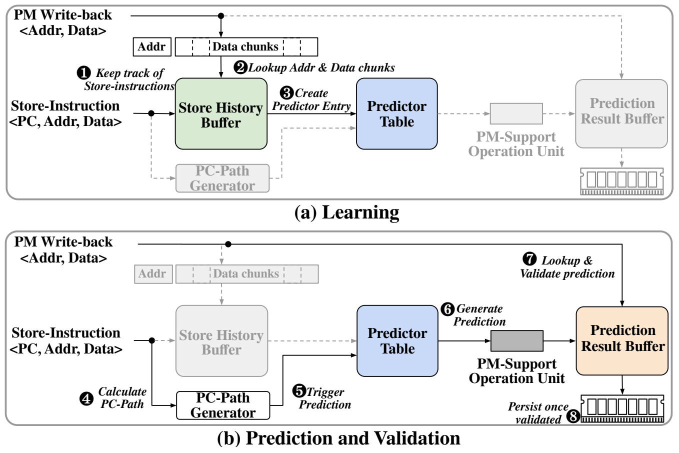


Fig. 12: Mechanism of PMWeaver. Components not involved in the current step are in gray.

IV. PMWeaver Overview

PMWeaver consists of three main components: a *learner* that records the addr/data-generating pattern for future predictions, a *predictor* that generates a prediction once observing the same learned pattern, and a *validator* that ensures correctness in case of mispredictions. Next, we present an overview of PMWeaver.

A. Learner: Identify addr/data-generating pattern

The *learner* in PMWeaver identifies the set of store-instructions that supply address or data values to subsequent PM write-backs. It maintains a *Store History buffer* (*StHistBuf*) that records recent store-instructions (step ❶ in Figure 12a). Upon a PM write-back,[2] the learner matches address and data values of the PM write-back to the values in the *StHistBuf* (step ❷), to determine the set of store-instructions that generate the address/data values of this PM write-back. As a PM write-back happens at cache-line-granularity but store-instructions are fine-grained, the learner breaks the cache line into smaller data chunks during matching (details in Section V-B1). Once identified, the learner adds this group of store-instructions (i.e., their PCs) to the Prediction Table (*PredTable*), where PMWeaver will use them to generate future predictions. An example of this matching procedure is presented in Section IV-D. With the *PredTable* entry created, the next step is to make a prediction when observing the same addr/data-generating stores again.

B. Predictor: Trigger prediction using execution path

As discussed in Section III-B, PMWeaver uses the execution path (i.e., a PC-path) to accurately trigger predictions. Thus, the predictor is implemented as a set-associative structure, index/tagged by the PC-path of a set of addr/data-generating store-instructions. As Figure 12b demonstrates, once observing the same PC-path again (step ❹), the predictor generates a prediction (step ❺) and waits for upcoming store-instructions to contribute their values. After collecting all values, the *predictor*

[2]PMWeaver determines whether a write-back is to PM based on the physical address from the TLB. However, this does not introduce extra TLB accesses as address translation always happens during a write-back.

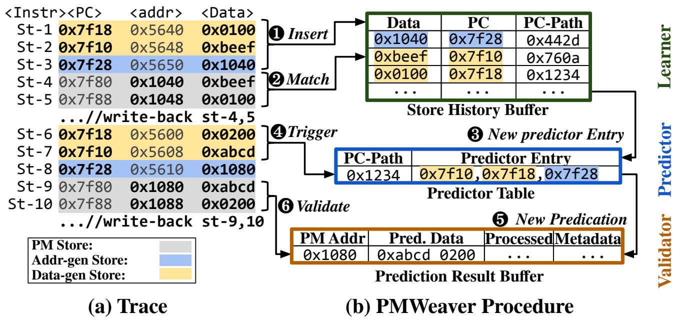


Fig. 13: A walk-through example.

sends the predicted PM write-back for precomputation of PM-support operations (step ❻). We describe more details about the PC-path generation in Section V-A1 and the handling of pending predictions in Section V-B2.

C. Validator: Resolve mispredictions and guarantee correctness

The last step is to validate the predicted PM write-back and invalidate mispredictions. PMWeaver maintains a *validator* that buffers the predictions in the *prediction result buffer (*PredResBuf*)*, as shown in Figure 12b. Upon reception of a new prediction, the *validator* inserts it into the *PredResBuf* after calculating its PM-support operations. When an actual PM write-back arrives, the *validator* looks up its destination address in the buffer, and compares the actual PM write-back with the prediction (step ❼). Once a prediction has been validated, the memory controller writes back the precomputed data and the associated metadata directly (step ❽), without breaking the original persistency model. To guarantee correctness, PMWeaver only updates the processor state after the validation (e.g., the encryption counter cache and the root node of the Merkle Tree). Note that a prediction can be incorrect when the metadata used at the time of precomputation differs from that at the time of PM write-back. PMWeaver invalidates such precomputations, to avoid persisting any incorrect states (more details in Section V-C3).

D. A Walk-through Example

Figure 13a is a store-instruction trace from a PM procedure that has been invoked *twice*. Figure 13b demonstrates the corresponding updates to the major structures in PMWeaver. First, the *learner* monitors incoming store-instructions and buffers them in the *StHistBuf* (step ❶). When a PM write-back is observed, it matches the values of its address and data in *StHistBuf* to find the addr/data-generating store-instructions (step ❷). This matching pattern is then added as an entry in the *PredTable* (step ❸). The *PredTable* entry also adds the associated PC-path to use it as the prediction trigger in the future. Then, the next time the same PC-path is observed, the *predictor* looks up the associated addr/data-generating store-instructions, and weaves together the address and data values of the future PM write-back from these store-instructions (step ❹). A completed prediction is then sent to the *PredResBuf* for precomputing the PM-support operations, without changing any processor or PM state (step ❺). Finally, when the actual write-back arrives, the *validator* compares the predicted values with the write-back (step ❻). After validation, it persists the data and metadata following the original persistency model and updates the processor state.

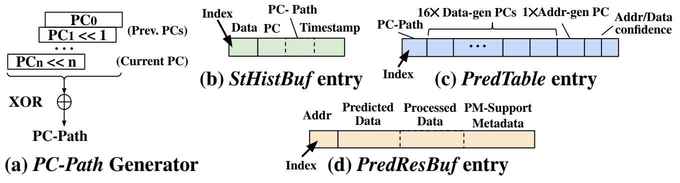


Fig. 14: Structure details of (a) PC-path, (b) store history buffer, (c) prediction table, and (d) prediction result buffer. The first field of each structure is the index (as indicated by the arrow).

V. PMWeaver Details

In this section, we will first discuss the details of the *leaner*, *predictor*, and *validator* (Section V-A to V-C), and then present two special-case predictors that work together with the main predictor (Section V-D).

A. Learner Details

This section describes two key components of the *learner* in detail, the *PC-path encoder* and the *store history buffer*.

1) Efficient PC-path encoding: PMWeaver encodes the PC-path with an XOR hash of 32 most recent store-instruction PCs. Our XOR hash maintains the information about instruction ordering by shifting n-th store-instruction by n bits, as shown in Figure 14a. Next, we describe how the encoded PC-path is stored in PMWeaver.

2) Store history buffer (StHistBuf) organization: To match the data and address of a write-back with prior store-instructions, the *StHistBuf* is implemented as a set-associative structure and is indexed/tagged by the data values. Figure 14b shows the fields in each *StHistBuf* entry. When inserting a new entry, *StHistBuf* first replaces older entries to track *n* most recent stores. The age of a store is determined by a *timestamp* in each *StHistBuf* entry. Then, it records the current PC-path in the same entry. This way, when this store-instruction is involved in a prediction, the predictor can choose the PC-path associated with the oldest entry.

B. Predictor Details

This section describes the key component in the *predictor*, the *prediction table*, and the detailed mechanism of gathering pending predictions from upcoming store-instructions.

1) Prediction table (PredTable) organization: The learner breaks the cache line into 16× 4B data chunks and 1× 8B address chunk before performing matching. Each of these chunks (16 data chunks and 1 address chunk) are matched to one store-instructions from the *StHistBuf* which predicts its value. These address and data generating chunks are then recorded in the *Predictor Table* (*PredTable*) for predicting future write-backs (Figure 14c). In case one data value matches

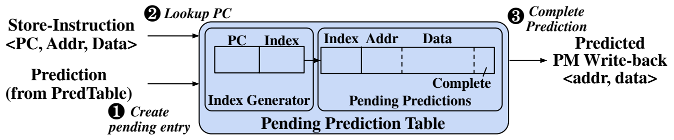
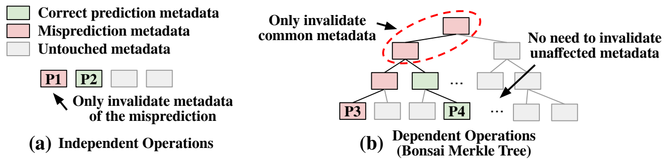


Fig. 15: Pending Prediction Table in PMWeaver.



Correct prediction metadata
Misprediction metadata
Untouched metadata
P1
P2
Only invalidate metadata of the misprediction
(a) Independent Operations
Only invalidate common metadata
No need to invalidate unaffected metadata
P3
P4
(b) Dependent Operations (Bonsai Merkle Tree)

Fig. 16: Misprediction invalidation.

multiple store-instructions in the *StHistBuf*, the learner takes the one with a lower timestamp value.

2) Pending prediction generation: The predictor maintains a *pending prediction table* (*PendPredTable*) to wait for the addr/data-generating store-instructions to contribute their values (Figure 15). The *PendPredTable* consists of two levels of cache-like structures. The first level is an *index generator* that maps the PCs of addr/data-generating store-instructions to the prediction result, and the second level is a table of *pending predictions* that is indexed by the first structure. Once the *predictor* generates a new prediction, the *PendPredTable* adds the PCs of the addr/data-generating instructions to the *index generator*, where each PC points to a common *pending prediction* entry (step ❶). Upon observing a store-instruction, the *predictor* determines whether it contributes to any pending prediction by looking up its PC in the *index generator* (step ❷). A matching PC then fills its data value into the indexed *pending prediction* entry. Once all of the pending addr/data-generating store-instructions have contributed their values to the prediction result, the predictor *merges* the predicted data with existing, unmodified data from the cache and sends the result to PM-support operation units for precomputation (step ❸).

C. Validator Details

In this section, we first talk about the decoupled precomputation for data and address, and then how the *validator* buffers predictions and handles mispredictions.

1) Decoupled address and data precomputation: PMWeaver takes the optimizations from a prior work [33], where PM-support operations are divided into smaller sub-operations that are dependent on address, data, or both. Thus, PMWeaver can precompute address- and data-dependent sub-operations separately. As predicting data is generally harder than address—all 16 data chunks need to be correct as compared to one address chunk—this decoupled precomputation allows a prediction with correct address but wrong data to partially reduce the overhead of the address-dependent sub-operations. Such decoupling is beneficial to secured memory systems where the encryption and Bonsai Merkle Tree nodes are mostly address-dependent. PMWeaver can thus precompute most sub-operations as long as the address is correctly predicted. Corresponding to the decoupled address and data precomputation, each *PredTable* entry maintains separate confidence bits for the address and data (Figure 14c), which can mitigate the utilization of PM-support units due to hard-to-predict address or data patterns.

2) Prediction result buffer (PredResBuf) organization: The *PredResBuf* tracks the predictions generated by the *predictor* to validate them when the actual write-back arrives at the memory controller. It is implemented as a set-associative cache, indexed by the predicted PM addresses. Each entry of *PredResBuf* keeps the predicted PM write-backs together with the associated PM-operation metadata (Figure 14d). Due to the difficulty in predicting the whole cache-line of data, in our implementation, the *PredResBuf* has 16 unique addresses, each of which keeps 16 distinct cache lines as candidates. Next, we describe how the *validator* resolves mispredictions and guarantees correctness.

3) Misprediction invalidation: A prediction can be incorrect when the metadata used at the time of precomputation differs from that of the write-back. This can happen when an unpredicted or mispredicted PM write-back modifies the metadata. We identify two invalidation scenarios: (1) predictions that only modify their own metadata, and (2) different predictions share metadata of PM-support operations. In deduplication and encryption, the precomputation result only depends on the predicted cache line (a hash value in the deduplication mechanism and an encrypted block in the encrypted memory), without affecting other locations. Therefore, the precomputation is independent of other predictions. Figure 16a demonstrates this independent PM-support operation, where invalidation only applies to one incorrect prediction. For example, prediction `P1` and `P2` do not share any metadata. Thus the incorrect prediction `P1` does not affect `P2`.

In contrast, the integrity-verification system uses a Bonsai Merkle Tree that consists of a hierarchical hash of per-cache-line counters (details in Section II-B). Thus, the metadata can be shared among predictions. As Figure 16b demonstrates, a prediction `P3` updates a series of Bonsai Merkle Tree nodes. And, the next prediction `P4` ends up dependent on some of the updated nodes (circled in the figure). Upon modification of the PM-support metadata, all dependent precomputations are *invalidated* and *recalculated* using the updated metadata. This might seem to be a substantial recalculation at a glance. Fortunately, as *PredResBuf* only keeps track of 16 unique prediction addresses (Section IV-C), the address precomputations are limited to at most 16 recalculations. Second, PMWeaver recalculates only the nodes that are affected by the metadata update from PM-support operation due to unpredicted/mispredicted write-backs (i.e., nodes close to the leaf often have unaffected metadata and do not need recalculation), as pointed out in Figure 16b. Finally, PMWeaver generates prediction early (evaluated in Section VI-B6), further reducing the performance impact of invalidations. In total, recalculations due to stale predictions only lead to a 3.8% performance overhead in a system with encryption and integrity-verification.

D. Special-case Predictors

This section describes two predictors integrated into PMWeaver for specific cases.

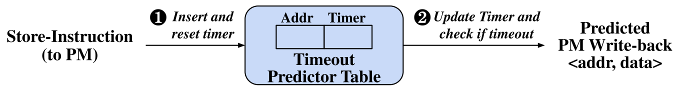

Store-Instruction (to PM)
❶ Insert and reset timer
Addr Timer
Timeout Predictor Table
❷ Update Timer and check if timeout
Predicted PM Write-back <addr, data>

Fig. 17: Integration of the timeout-based predictor.

1) Zero-value prediction: Common data values can degrade the prediction accuracy, as multiple store-instructions can match a common data value and lead to data mispredictions. PMWeaver handles the most common case of *zero values* by directly predicting zeros (learned by setting a zero-word bit) if a data chunk in the write-back is all-zero, rather than finding their data-generating PCs.

2) Timeout-based prediction: Although a large fraction of write-backs happen within a small time duration after data becomes available in the cache as shown in Section II-D, there are PM write-backs that happen long after data has arrived in the cache. Predicting them with store-instructions is unnecessary, as there is sufficient time to precompute these write-backs using data in the cache. Thus, we design an additional predictor that works collaboratively with the main store-instruction-based predictor to cover these write-backs. The challenge is how can one identify when a cache line has seen its last store and can be sent for precomputation? We exploit the observation that updates to PM are usually bursty [18, 21]. As such, if there has not been any store to a PM location for a while, it is likely that updates to this location have completed.

Design: To determine the time since the last store-instruction for a PM address, we integrate a *Timeout Predictor Table (TimeoutPred)* into PMWeaver. Each *TimeoutPred* entry maps a PM address to a timer that increments every 10 ns. Once the entry observes an incoming store-instruction to PM, the timer resets (Figure 17, step ❶). As our profiling result in Figure 7 has a knee point at around 150 ns, PM write-backs that happen before this point are handled by the prediction based on store-instructions, and those happen afterward are handled by this timeout-based prediction. Once a timer has reached this 150 ns threshold, PMWeaver predicts this cache line has seen its last update and will obtain the data from the cache for PM-support precomputation (step ❷). Predictions made by this timeout mechanism are validated by the same validator (Section IV-C) to guarantee correctness. Our timeout-based prediction is simple but effective (evaluated in Section VI-B5).

VI. Evaluation

A. Methodology

1) System configuration: We model PMWeaver on the cycle-accurate Gem5 simulator [61] with the configuration listed in Table I. We model PMWeaver's learner and predictor access latency as the L1 cache latency, and the validator's latency as the LLC latency. Note that the size and associativity of each prediction structure are small. Our design is not sensitive to the latencies of prediction-structures, as PMWeaver precomputes the PM-support operations substantially ahead of the write-back (Section VI-B6). We ensure that the metadata of PM-support operations are consistent with the corresponding persistent data, by persisting modifications to metadata with a write-through approach, similar to prior works on secured PM systems [34].

TABLE I: System configuration.

CPU	Out-of-Order Cores, x86, 3.0GHz
L1 I/D Cache	32kB each, 8-way, private, 1 ns
L2 Cache	128kB per-core, 4-way, private, 5 ns
L3 Cache	2MB per-core, 16-way, shared, 20 ns
DRAM	16GB DDR4 2400MT/s
PM	16GB PCM, with ADR support, 533MT/s: tRCD/tCL/tCWD/tFAW/tWTR/tWR = 48/15/13/50/7.5/300 ns [33, 51, 60]
Parameters of PM-Support Operations	AES-128: 40 ns, SHA-1 40 ns [49] Counter cache: 256kB, Integrity cache: 256kB
PC-Path Length	32
Store History Buffer	16-way 512 entries (8.75kB), per-core, 1 ns
Predictor Table	32 entries (4.5kB), per-core, 1 ns
Timeout Predictor Table	4 entries (34B), per-core, 1 ns
Pending Prediction Table	Index Generator: 128 entries (1.1kB), Pending Predictions: 8 entries (0.57kB), both per-core, 1 ns
Prediction Result Buffer	256 entries (38kB), (16 unique addresses, and 16 data entries per address), per-core, shared, 5ns

TABLE II: Workloads for evaluation.

	Name		Name
Transactions	B-Tree	**Low-level**	Linked List
	C-Tree		Hashmap Low-Level
	RB-Tree		Array Swap
	Skiplist		TATP
	Hashmap TX		TPCC

2) Workloads and design points: We evaluate workloads from the widely-used PMDK library [12] to cover the mainstream programming paradigm for PM systems, and popular workloads used in prior works [21, 33, 51] (Table II). Moreover, as PMWeaver transparently predicts the write-backs in hardware, it needs *no modification* to the workloads. We evaluate the following design points with two different sets of PM-support operations (introduced in Section II-B):

- *Original:* a system without any PM-support operations.
- *Enc+Veri:* a baseline system with PM-support operations for security, using counter-mode encryption and integrity verification.
- *Dedup:* a baseline system with a PM-support operation that performs deduplication.
- *Enc+Veri/Dedup w/ PMWeaver:* systems with PMWeaver to mitigate PM-support latencies.

In the rest of the evaluation, the term "coverage" refers to the number of correctly predicted PM write-backs over the total PM write-backs, and "accuracy" refers to the number of useful predictions over the total number of predictions generated.

B. Evaluation Results

1) Overall performance: We start with comparing the overall performance of PMWeaver to all the design points described in

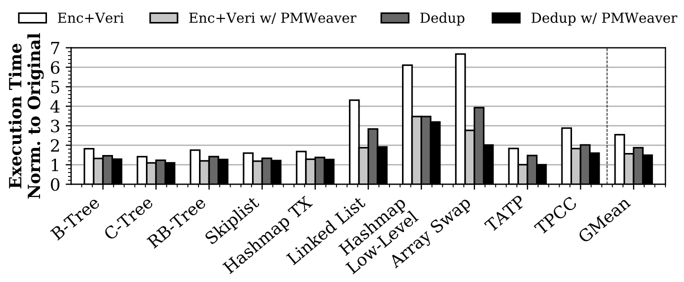


Fig. 18: Performance of PMWeaver.

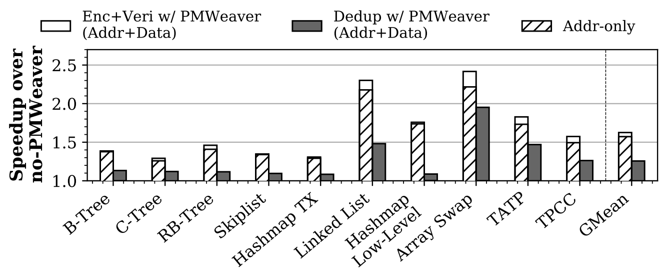


Fig. 19: Performance breakdown of PMWeaver.

Section VI-A. Figure 18 shows the execution time of two PM-support systems with and without PMWeaver, normalized to the *original* system without any PM-support operations. Integrating *Enc+Veri* and *Dedup* introduces 2.54× and 1.87× slowdown over the *original* system on average (Geo-mean), respectively. With PMWeaver, the slowdown is reduced to 1.56× and 1.49×. As such, PMWeaver achieves 1.63× and 1.26× speedup over the no-prediction baselines.

2) Performance breakdown: Figure 19 further breaks down the speedup from PMWeaver. The y-axis is the speedup of PMWeaver over the baseline design with PM-support operations for *Enc+Veri* and *Dedup*. First, we observe that workloads based on low-level primitives (right five) have higher speedup compared to those based on transactions (left five). This is because low-level-based workloads write back data more frequently, and thus mitigating PM-support latency gains more benefit. Second, we demonstrate the fraction of speedup that comes from predicting only the address (i.e., there is only precomputation for address-dependent PM-support operations) as indicated by the shaded area at the bottom of each bar. The breakdown shows that the *Enc+Veri* system significantly benefits from the address-only prediction, whereas the *Dedup* system mainly benefits from data prediction. The reason is that the encryption and integrity verification mechanisms heavily depend on the address, whereas the deduplication mechanism mostly depends on the data. We conclude that both the address and the data prediction methods are effective in PMWeaver.

3) Comparison with software-based precomputation: We compare with a recent work, Janus [33], that precomputes PM-support operations using software hints. We obtain annotations directly from Janus for the workloads based on low-level primitives (right five). And, we annotate both the application and the underlying library code for workloads based on PMDK transactions (left five), following the method described in Janus.

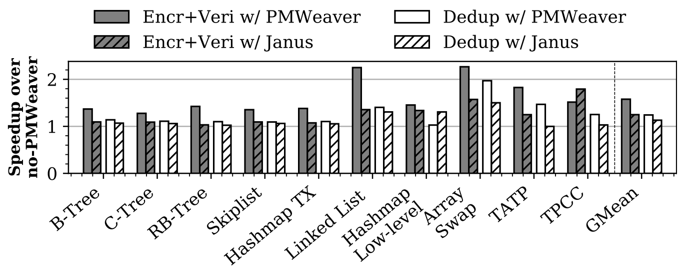


Fig. 20: PMWeaver vs. software-based precomputation [33].

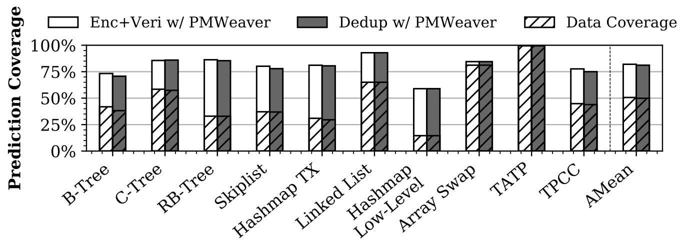


Fig. 21: Percentage of PM write-backs correctly predicted.

Figure 20 shows that the software-based solution from Janus provides less performance benefit than PMWeaver in most cases. On average, Janus provides 1.25× and 1.13× speedup in *Enc+Veri* and *Dedup*—20.9% and 9.0% slower than PMWeaver. The software-based solution is less effective due to the limited scope of annotation within functions or loops, as precomputing across dynamic control-flows is difficult. Overall, PMWeaver provides better performance compared to Janus, without the need for software modifications.

4) Prediction coverage: Figure 21 presents the prediction coverage of PMWeaver. The shaded area in the bottom represents the fraction of prediction where both address and data are correctly predicted, while the remaining solid bar represents the correct prediction only for addresses. First, we observe that both PM-support systems have reasonably high prediction coverage—on average PMWeaver covers 50.6% and 49.9% PM write-backs in two systems, respectively. Second, both systems have similar prediction coverage as the prediction mainly depends on the store-instructions and write-backs. Third, the coverage for address prediction is higher than the data prediction (81.16% for address, where 49.90% for data). Predicting data is more difficult than addresses, as a cache line consists of 16 data chunks and each of them needs to be predicted correctly to predict the data of the whole cache line.

5) Timeout-based predictor: PMWeaver employs a timeout-based predictor (Section V-D) that works alongside the main predictor. Figure 22 shows the breakdown of the prediction

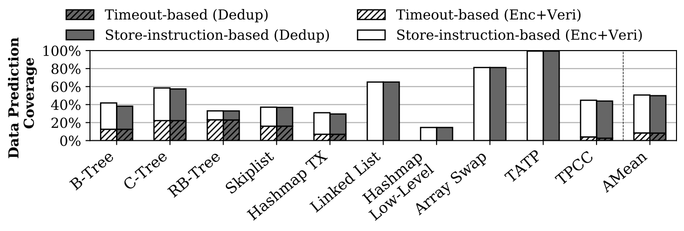


Fig. 22: Breakdown of data prediction.

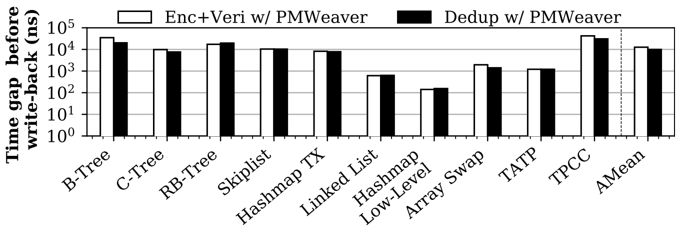


Fig. 23: Average time between prediction and write-back.

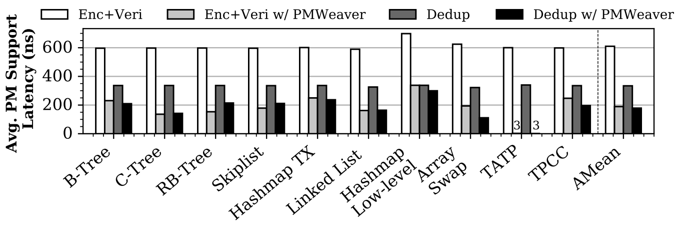


Fig. 24: Effective PM-support latency with PMWeaver.

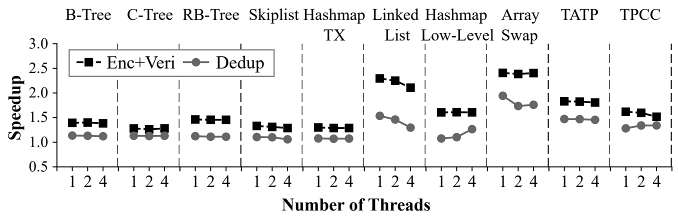


Fig. 25: Speedup of PMWeaver in multithreaded systems.

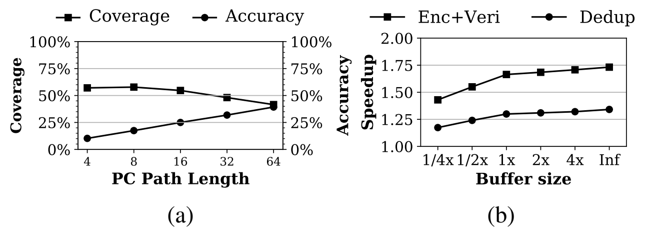


Fig. 26: A sensitivity analysis on variable (a) PC-path sizes and (b) predictor buffer sizes.

coverage (address and data), where the shaded area at the bottom indicates the fraction from the timeout-based prediction. On average, prediction from store-instructions provides 42.15% coverage (address + data), and the timeout-based prediction adds 8.48% coverage on top.

6) Prediction timeliness: Figure 23 shows the average time between the point when the prediction is generated (with all address and data fields complete) to the point when the actual PM write-back reaches PMWeaver' validator. First, we notice that the time gaps are almost identical in both systems. Second, the prediction-to-write-back time gap in most workloads (average 9976 ns) is more than sufficient to precompute all PM-support operations. On average, only 7% of predictions are validated too late for their precomputations to be beneficial. Hashmap Low-Level is an exception that has a time-gap of 155 ns. The reason is that it uses direct pointer updates to ensure crash consistency instead of logging, causing the write-backs to happen immediately after the address/data become available. Nonetheless, there is still a significant speedup of 1.76× and 1.09× in the two systems, respectively, as precomputation can overlap a large fraction of the PM-support latency. Figure 24 shows the reduction in the average PM-support latencies with precomputation (3.27× and 1.87× lower in the two systems, respectively). We expect that PMWeaver will still benefit PM-support operations that take longer latency.

7) Multithreaded evaluation: To demonstrate the scalability of PMWeaver, we show the speedup when 1, 2, and 4 instances of the workloads are running concurrently on separate cores in Figure 25. The per-core and shared PMWeaver structures are scaled up linearly to the number of the cores to accommodate the increased write traffic. Overall, PMWeaver maintains its performance benefits across multithreaded scenarios.

8) Sensitivity analysis: Figure 26a scales the PC-path length from 4 to 64 and presents the corresponding coverage (correctly predicted writes/total writes) and accuracy (correctly predicted writes/total predictions). As the previous evaluation has shown near-identical coverage in two different PM-support systems, we only demonstrate results from the *Enc+Veri* system. We observe that as the length of the PC-path increases, the coverage slightly decreases due to fewer matching paths but the accuracy gets better because of fewer incorrectly triggered predictions. Overall, we choose a PC-path length of 32 as it reaches a good balance between the prediction coverage and the accuracy.

With the PC-path length being 32, Figure 26b presents the speedup of PMWeaver while scaling the other PMWeaver buffers (listed in Table I) from 0.25× to infinity. Overall, the speedup increases with larger sizes. However, the speedup curve is flattened once the size reaches 1×, even compared with the ideal scenario. Therefore, as a trade-off between performance and hardware overhead, we choose the sizes listed in Table I.

9) Area overhead: Table I lists the size of the major hardware structures PMWeaver introduces for prediction. The learner and predictor structures have a storage overhead of 15.2 kB (per core), and the validator has an overhead of 38 kB (shared). With 22 nm technology [62], the per-core area overhead is about 0.24 mm^2, and the shared area overhead is 0.65 mm^2. Compared to the area of the CPU, PMWeaver only takes 3.1% extra area. Considering the significant speedup, we believe that the area overhead from PMWeaver is small.

10) Energy overhead: Figure 27 presents the energy breakdown of PMWeaver. We model the hardware overhead of

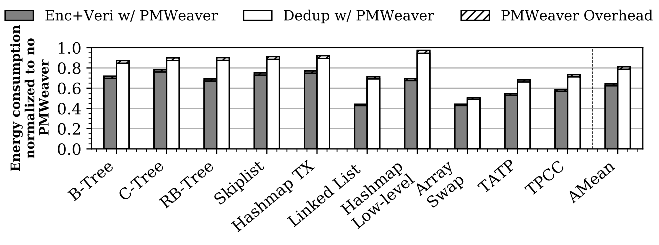


Fig. 27: Energy consumption breakdown of PMWeaver.

PMWeaver with McPAT (22 nm technology) [63]. We follow the energy cost parameters in [64, 65] and scale down to 22nm to model the PM-support overhead, and take mispredictions into account. On average, PMWeaver saves 38.5% and 20.4% energy for *Enc+Veri* and *Dedup* as compared to the baseline. More specifically, the average overhead from PMWeaver's prediction structure is only 2.9%, while mispredictions and stale-prediction invalidations incur 1.6% energy overhead. We conclude that, even accounting for the energy overhead of mispredictions, PMWeaver substantially improves energy.

VII. Related Works

In this section, we discuss the related works in the area of PM systems, and data value and write prediction.

Crash consistency support for PM: There have been an assortment of proposals and implementations that ensure a consistent recovery of persistent data on PM. Examples include persistency models [18]–[20], hardware transactions [66, 67], software libraries [12, 13, 68], and PM file systems [7, 8, 10, 11]. These crash consistency solutions tends to write-back data to PM in certain order [18, 21, 23]–[26, 69]–[72]. Therefore, the write-back latency is added to the critical path. PMWeaver can significantly reduce the overhead of writes without any software modification.

PM-support operations: PM systems also require backend memory operations that serve a variety of purposes. Due to the persistence property of PM, security-aware systems apply encryption and integrity verification to prevent unauthorized access and tampering to the data on PM [34]–[40]. PM systems also adopt deduplication and compression techniques to overcome the bandwidth limitation and achieve more efficient storage [45, 48, 53]. To mitigate the limited life time of PM devices, wear-leveling and error correction are also necessary for a practical PM system [41]–[44]. These operations, unfortunately further increase the performance-critical write latency. There have been works that design more efficient PM-support operations to reduce this latency [23, 34, 49, 50]. To further reduce the latency, a recent work, Janus proposes to precompute these operations using software hints [33]. However, such solution breaks the system abstraction and degrades portability and increases programmer's effort. In comparison. PMWeaver can hide most of the PM-support latency without requiring any software modification.

Value and write prediction: Value prediction has been used to break the dependency of latency-critical reads for better forward progress [73]–[77], and predict data-dependent branches [78]. Our work is different in two major aspects. First, PMWeaver predicts the PM write-backs to mitigate the performance-critical write latency. Second, the prediction uses data values written by addr/data-generating store-instructions, instead of directly predicting the values the stride or context. Prediction based on instructions is widely used for purposes such as prefetching [79, 80], memory disambiguation [81], cache replacement [82]–[85], and branch prediction [86, 87]. PMWeaver is different as it predicts not only the address but also the data values, and further combines the value predictions from *multiple* store-instructions to perform a cache-line-sized write prediction. Nonetheless, existing address and data prediction methods of prior works can be integrated into PMWeaver to achieve a better coverage and accuracy. For example, next pointer prediction [88]–[90] can improve the accuracy of PM address prediction, and address-value-delta based prediction [91] can augment PMWeaver to calculate next address/data predictions from previous predictions.

VIII. Conclusions

Software for persistent memory (PM) needs to ensure a consistent recovery in event of a failure, by carefully ordering writes to PM. Thus, the write-back latency is placed on the critical path. On the hardware side, there are different types of PM-support operations, such as encryption, integrity verification, compression, and deduplication, that are necessary but increase the write latency. We observe that the address and value associated with PM write-backs are often available in prior store-instructions. Therefore, we propose PMWeaver that *learns* the correlation of address and data of PM write-backs with prior store-instructions, and *predicts* the address and data of write-backs to precompute these PM-support operations off the critical path. On average, PMWeaver predicts 81.16% addresses and 49.90% of data of PM write-backs in common PM workloads, and provides 1.63× and 1.26× speedup over a no-PMWeaver baseline by precomputing two types of PM-support operations: a combination of encryption and integrity verification, and deduplication.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work is supported by the Google fellowship program, NSF awards CCF-1822965 and CNS-2046066, and the SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP).

Appendix

A. Abstract

PMWeaver is implemented using Gem5 [92], including the predictor structures, the PM-support operations, and support for PMDK libraries [12]. The artifact includes the simulator of PMWeaver that is implemented on top of Gem5 [61], PMDK modified to work with Gem5, and PM workloads from previous works [33, 36, 51]. The artifact also includes the scripts to run and plot the results in this paper.

B. Artifact check-list (meta-information)

- **Program:** PMWeaver
- **Compilation:** gcc/g++-5
- **Metrics:** Speedup using PMWeaver, workload execution time with or without PMWeaver, data and address coverage of PMWeaver's predictions.
- **Output:** Performance plots for PMWeaver.
- **Experiments:** (1) Slowdown due to Encryption and Verification, Figure 1. (2) Time gap between a PM update and corresponding write-back, Figure 7. (3) PMWeaver execution Time Normalized to Original, Figure 18. (4) Performance breakdown of PMWeaver,

Figure 19. (5) Data Prediction Coverage, Figure 22. (6) Average PM Support Latency (ns), Figure 24.
- **Disk space required:** 16 GiB
- **Recommended system memory:** 32 GiB+
- **Recommended CPU count:** 16+
- **Time needed to prepare workflow:** 10 CPU hours
- **Time is needed to complete experiments:** 50 CPU hours
- **Publicly available:** Yes
- **Code license**: Revised BSD License
- **Archive DOI:** 10.5281/zenodo.5136539

C. Description

1) How to access: We maintain a GitHub repository for the artifact at https://pmweaver.persistentmemory.org. There is also an archived version at https://doi.org/10.5281/zenodo.5136539.

Software dependencies: This artifact depends on the following environment.
- Docker 20.10.7 or higher
- Ubuntu 18.04 or higher
- Python 2.7, Python 3
- g++-5
- *gem5*: `swig`, `m4`, `libprotobufdev`, `libboost-all-dev` `libgoogle-perftools-dev`, `protobuf-compiler`,
- *pmdk*: `libdaxctldev`, `libjemalloc1`, `libjemallocdev`, `libndctldev`
- *Python2 packages*: `scons`, `six`, `python-config`
- *Optional*: Anaconda and task-spooler

Data sets: We evaluated the following workloads.
- PMDK libpmemobj examples: BTree, CTree, Skip List, and Hashmap-TX [12]
- Array Swap, Hashmap Low-Level, Linked List, TATP and TPCC [33]

D. Installation

This artifact has the following structure:
- `Dockerfile`: Dockerfile for building PMWeaver with all the dependencies.
- `gem5/`: Gem5 with PMWeaver implementation.
- `pm_images/`: Pre-generated PM images for PMDK `data_store` workload.
- `pmdk/`: Intel PMDK with modifications to run on Gem5.
- `janus_workload/`: PM workloads from Janus [33].
- `scripts/`: Scripts to run experiments and plot results.
- `scripts/run_part1.py`: Runs experiments for Fig. 1, 8, 18, 22, 23, and 24.
- `scripts/run_part2.py`: Runs experiments for Fig. 19.
- `scripts/plot_scripts/plot_part1.py`: Generate Fig. 1, 8, 18, 22, 23, and 24.
- `scripts/plot_scripts/plot_part2.py`: Generate Fig. 19.

Setup Environment: PMWeaver artifact comes with a Dockerfile that includes the source code, the dependencies, and the scripts to run experiments. To use it, please install Docker from https://docs.docker.com/get-docker/.

Once docker installation is complete, build the PMWeaver image using the following command from the repository's root:

```
docker build -t pmweaver .
```

Docker would now build gem5, pmdk, and all the included workloads. Once the image is built, you can access the shell in the container using the following command:

```
docker run -i -t -p 8888:8888 pmweaver /bin/bash
```

E. Experiment Workflow

PMWeaver artifact is setup to run the experiments, generate the raw simulation data, and then use a separate script to plot the results in the paper. The experiments are split into two parts, part-1 and part-2, each of which runs a different set of experiment and plots a different set of figures from the paper.

The step to run the experiment and plot the corresponding data is automated using the scripts under the directory `scripts/` of the artifact.

Please follow these common directions before starting an experiment:

1) Scheduling jobs and checking progress: PMWeaver uses Task Spooler to schedule large number of jobs on a limited number of CPUs. All the scripts to run the experiments (`scripts/run_*`) are set up to use the Task Spooler correctly. To check the progress of an experiment, you may run the following command at any time during or after the execution:

```
scripts/helper_scripts/check_progress.sh
```

2) Terminating stale jobs: To make sure all the stray processes are terminated after an experiment, run the following script before starting a new experiment:

```
scripts/helper_scripts/kill_stray.sh
```

3) Managing container's lifecycle: While running the experiments, please note that exiting the shell for the docker container would kill all the processes in that container. Thus, when running on a local computer, make sure not to exit the shell. When running over SSH, please use Docker in a GNU Screen session [93].

4) Generating and viewing plots: After running an experiment and corresponding script to generate the plots, figures will be written to the directory `/pmweaver_ae/scripts/plots/`. To view the plots, run the following command:

```
scripts/helper_scripts/start_server.sh
```

The plots will then be available at http://localhost:8888/ under the `plots/` directory.

F. Evaluation and Expected Result

PMWeaver artifact evaluates the major performance results:
1) Improvement in execution time across workloads (Fig. 18 and 19).
2) PMWeaver characteristics, including prediction coverage (Fig. 22), prediction timing (Fig. 23) and PM-support operation latency improvement (Fig. 24).
3) Slowdown with Encryption and Verification PM-support operations (Fig. 1) and the time gap between a PM update and the corresponding write-back (Fig. 7).

Before running a script, please execute the following command to make sure there are no stale jobs:

```
scripts/helper_scripts/kill_stray.sh
```

Part 1: Coverage and Performance Comparison: To reproduce the results of Fig. 1: Slowdown due to Enc+Veri, Fig. 7: PM update and write-back time gap, Fig. 18: Performance of PMWeaver, Fig. 22 Data prediction coverage, Fig. 23: Time between prediction and write-back, and Fig. 24: PM-support latency, please run the following script.

```
scripts/run_part1.py
```

To check the progress, run `check_progress.sh`:

```
scripts/helper_scripts/check_progress.sh
```

Once done, all the jobs will show up with the status "`finished`". To plot the results, please run the following script:

```
scripts/plot_scripts/plot_part1.py
```

Part 2: Performance Breakdown: To generate Fig. 19: PMWeaver performance breakdown, please run the following script after making sure that there are no active jobs:

```
scripts/run_part2.py
```

Once all the jobs are completed, please run the following script to plot the results:

```
scripts/plot_scripts/plot_part2.py
```

G. Experiment Customization

Manually executing workloads: To run Gem5 with PMWeaver please use the following command with a workload:

```
/pmweaver_ae/gem5/build/X86/gem5.opt \
   /pmweaver_ae/gem5/configs/example/se.py \
   --cpu-clock=3GHz --mem-size=8GB --caches \
   --mem-type=DDR4_2400_8x8 --l2cache --l3cache \
   --cpu-type=DerivO3CPU --l1i_size=32kB \start
   --l1d_size=32kB --l2_size=256kB \
   --l3_size=2MB -c \
   </path/to/workload/binary> \
   -e /pmweaver_ae/gem5/env -n 1 -o \
   <workload options>
```

- `</path/to/workload/binary>`: Path to the workload.
- `<workload options>`: Cmd line options for the workload.

PMWeaver version of Gem5 also reads the following environment variables to configure different parameters:

- `PATH_HISTORY_SIZE`: Sets the length of the path history
- `ENABLE_DW`: Enables Dedup PM-support operation.
- `ENABLE_EV`: Enables Env+Veri PM-support operation.
- `USE_PREDICTOR`: Enables PMWeaver predictions.

References

[1] Intel, "Intel Optane DC persistent memory," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[2] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database management system," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, 2017.

[3] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. B. Zdonik, and S. Dulloor, "A prolegomenon on OLTP database systems for non-volatile memory," in *ADMS@VLDB*, 2014, pp. 57–63.

[4] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," in *International Conference on Very Large Data Bases (VLDB)*, 2014.

[5] Intel, "Redis," https://github.com/pmem/redis/tree/3.2-nvml, 2019.

[6] Lenovo, "Memcached-pmem," https://github.com/lenovo/memcached-pmem, 2018.

[7] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *European Conference on Computer Systems (EuroSys)*, 2014.

[8] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[9] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[10] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[11] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[12] Intel, "Persistent memory programming," https://pmem.io/.

[13] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memeory," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[14] S. Haria, M. D. Hill, and M. M. Swift, "MOD: Minimally ordered durable datastructures for persistent memory," 2020.

[15] S. Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, 2020, p. 438.

[16] J. Jeong and C. Jung, "Pmem-spec: Persistent memory speculation (strict persistency can trump relaxed persistency)," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[17] S. M. Shahri, S. Armin Vakil Ghahani, and A. Kolli, "(almost) Fence-less persist ordering," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[18] 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)*, 2017.

[19] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[20] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[21] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[22] 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)*, 2017.

[23] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "PMTest: A fast and flexible testing framework for persistent memory programs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[24] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan, "Cross-failure bug detection in persistent memory programs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[25] I. Neal, B. Reeves, B. Stoler, A. Quinn, Y. Kwon, S. Peter, and B. Kasikci, "AGAMOTTO: How persistent is your persistent memory application?" in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[26] I. Neal, A. Quinn, and B. Kasikci, "Hippocrates: Healing persistent memory bugs without doing any harm," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[27] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, 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), 2017.

[28] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "REWIND: Recovery write-ahead system for in-memory non-volatile data-structures," *PVLDB*, 2015.

[29] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "NVthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[30] 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 *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[31] E. Giles, K. Doshi, and P. Varman, "Continuous checkpointing of HTM transactions in NVM," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2017.

[32] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2018.

[33] 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 International Symposium on Computer Architecture (ISCA)*, 2019.

[34] P. Zuo, Y. Hua, and Y. Xie, "SuperMem: Enabling application-transparent secure persistent memory with low overheads," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[35] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-efficient encryption for non-volatile memories," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[36] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[37] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-NVM: Persistency for integrity-protected and encrypted non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[38] K. A. Zubair and A. Awad, "Anubis: Ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[39] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[40] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for CMP scaling," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[41] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," in *Proceeding of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010.

[42] M. K. Qureshi, M. Franchescini, V. Srinivasan, L. Lastras, B. Abali, and J. Karidis, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[43] M. K. Tavana, A. K. Ziabari, M. Arjomand, M. Kandemir, C. Das, and D. Kaeli, "REMAP: A reliability/endurance mechanism for advancing PCM," in *Proceedings of the International Symposium on Memory Systems (MemSys)*, 2017.

[44] R. Azevedo, J. D. Davis, K. Strauss, P. Gopalan, M. Manasse, and S. Yekhanin, "Zombie memory: Extending memory lifetime by reviving dead blocks," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[45] V. Young, S. Kariyappa, and M. Qureshi, "Enabling transparent memory-compression for commodity memory systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

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

[47] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[48] D. Das, D. Arteaga, N. Talagala, T. Mathiasen, and J. Lindström, "NVM compression—hybrid flash-aware application level compression," in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[49] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors OS- and performance-friendly," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[50] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[51] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[52] Intel, "Intel 64 and IA-32 architectures software developer's manual," https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2019.

[53] 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 the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[54] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[55] G. E. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[56] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and implementation of the aegis single-chip secure processor using physical random functions," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[57] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "CacheDedup: In-line deduplication for flash caching," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*, 2016.

[58] Intel, "Method and apparatus for completing pending write requests to volatile memory prior to transitioning to self - refresh mode," Patent, US20190147938A1, 2019.

[59] D. Mulnix, "Intel Xeon Processor D product family technical overview," https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview.

[60] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

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

[62] A. Satoh and T. Inoue, "ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS," *INTEGRATION, the VLSI journal*, 2007.

[63] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[64] S. Banik, A. Bogdanov, and F. Regazzoni, "Exploring energy efficiency of lightweight block ciphers," in *Revised Selected Papers of the 22Nd International Conference on Selected Areas in Cryptography (SAC)*, 2016.

[65] J.-P. Kaps and B. Sunar, "Energy comparison of AES and SHA-1 for ubiquitous computing," in *International Conference on Embedded and Ubiquitous Computing*. Springer, 2006.

[66] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, “Kiln: Closing the performance gap between systems with and without persistence support,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[67] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, “ATOM: Atomic durability in non-volatile memory through hardware logging,” in *Proceedings of The 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[68] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[69] S. Liu, S. Mahar, B. Ray, and S. Khan, “PMFuzz: Test case generation for persistent memory programs,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[70] P. Lantz, D. S. Rao, S. Kumar, R. Sankaran, and J. Jackson, “Yat: A validation framework for persistent memory software.” in *USENIX Annual Technical Conference (ATC)*, 2014.

[71] H. Gorjiara, G. H. Xu, and B. Demsky, “Jaaru: Efficiently model checking persistent memory programs,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[72] B. Di, J. Liu, H. Chen, and D. Li, “Fast, flexible, and comprehensive bug detection for persistent memory programs,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[73] Y. Sazeides and J. E. Smith, “The predictability of data values,” in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1997.

[74] B. Calder, G. Reinman, and D. M. Tullsen, “Selective value prediction,” in *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*, 1999.

[75] A. Perais and A. Seznec, “EOLE: Paving the way for an effective implementation of value prediction,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[76] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and load value prediction,” in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[77] T. Nakra, R. Gupta, and M. L. Soffa, “Global context-based value prediction,” in *Proceedings Fifth International Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[78] M. U. Farooq, L. K. John *et al.*, “Store-load-branch (SLB) predictor: A compiler assisted branch prediction for data dependent branches,” in *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[79] K. J. Nesbit and J. E. Smith, “Data cache prefetching using a global history buffer,” in *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, 2004.

[80] M. Grannaes, M. Jahre, and L. Natvig, “Multi-level hardware prefetching using low complexity delta correlating prediction tables with partial matching,” in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2010.

[81] A. Moshovos and G. S. Sohi, “Streamlining inter-operation memory communication via data dependence prediction,” in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, 1997.

[82] C. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, “SHiP: Signature-based hit predictor for high performance caching,” in *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[83] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, “Ship++: Enhancing signature-based hit predictor for improved cache performance,” in *The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA 2017)*, 2017.

[84] A. Jain and C. Lin, “Back to the future: Leveraging belady’s algorithm for improved cache replacement,” in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[85] Z. Wang, S. M. Khan, and D. A. Jiménez, “Improving writeback efficiency with decoupled last-write prediction,” in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[86] A. Seznec, “A 256 kbits L-TAGE branch predictor,” *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 2007.

[87] ——, “Exploring value prediction with the EVES predictor,” in *CVP-1 2018 - 1st Championship Value Prediction*, Los Angeles, United States, 2018. [Online]. Available: https://hal.inria.fr/hal-01888864

[88] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, “Pointer cache assisted prefetching,” in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2002.

[89] R. Cooksey, S. Jourdan, and D. Grunwald, “A stateless, content-directed data prefetching mechanism,” in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[90] A. Roth, A. Moshovos, and G. S. Sohi, “Dependence based prefetching for linked data structures,” in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[91] O. Mutlu, H. Kim, and Y. N. Patt, “Address-Value Delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns,” in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2005.

[92] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The Gem5 simulator,” *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[93] Free Software Foundation, “Screen - GNU project,” https://www.gnu.org/software/screen/, 2021.