NearPM: A Near-Data Processing System for Storage-Class Applications

Yasas Seneviratne
University of Virginia
United States
yasas@virginia.edu

Korakit Seemakhupt
University of Virginia
United States
korakit@virginia.edu

Sihang Liu*
University of Waterloo
Canada
sihangliu@uwaterloo.ca

Samira Khan
University of Virginia
United States
samirakhan@virginia.edu

Abstract

Persistent Memory (PM) technologies enable both fast memory access and recovery in case of a failure. To ensure crash-consistent behavior, programs need to enforce persist ordering and employ mechanisms that introduce additional data movements such as logging, checkpointing, and shadow-paging. The emerging near-data processing (NDP) architectures can effectively reduce this overhead. In this work, we propose NearPM, a near-data processor that accelerates common, primitive operations that are crucial to crash consistency. Using these primitives, NearPM accelerates commonly-used crash-consistency mechanisms. NearPM further reduces the synchronization overheads between the NDP and the CPU by handling ordering near memory. We propose Partitioned Persist Ordering (PPO) that ensures a correct persist ordering between CPU and NDP devices, as well as among multiple NDP devices. We prototype NearPM on an FPGA platform. NearPM executes the data-intensive operations of crash-consistency mechanisms with correct ordering guarantees, while the rest of the program runs on the CPU. We evaluate nine PM workloads, each implemented in three crash consistency mechanisms: logging, checkpointing, and shadow paging. Overall, NearPM achieves 4.3−9.8× speedup in the NDP-offloaded operations and 1.22 − 1.35× speedup in the whole applications.

CCS Concepts: • **Hardware** → **Memory and dense storage**; • **Computer systems organization** → *Other architectures.*

Keywords: Persistent Memory, Persistency, Near-data Processing, Accelerator

1 Introduction

Persistent main memory (PM) technologies offer both high performance and data persistence. For example, Optane PM [38] can be accessed through the DDR interface. Other alternative PM systems [10, 76] are also being developed for the upcoming PCIe-based Compute Express Link (CXL) standard [16]. Unlike conventional storage devices (e.g., HDD and SSD), these PM systems enable applications to perform direct access to PM, without going through the file system intermediaries. Thus, PM-optimized applications can benefit from a faster data path. These opportunities have inspired research on developing and deploying PM [4, 6, 8, 13, 31, 48, 56, 79]. However, a new challenge arises—without the file system, it is now up to the applications to manage the recovery of persistent data. In case of a failure (e.g., a system crash or power outage), applications that directly access PM need to ensure that the persistent data is maintained in a recoverable state. We call this property the crash consistency guarantee.

There have been myriad solutions that provide crash consistency guarantees for PM-based applications. For example, the undo-logging approach makes a backup of the existing data to PM before updates [11, 12, 14, 15, 21, 32, 40, 54, 58]; the checkpointing method periodically makes a snapshot of persistent data to keep a consistent, recoverable copy [5, 23, 30, 51, 71, 73]; the shadow-paging mechanism redirects writes to a shadow memory and changes page reference at commit [37, 68, 69, 88]. However, these mechanisms come with a performance cost. First, crash consistency guarantees require writes to become persistent in a specific order, introducing additional ordering constraints. For example, the undo-logging mechanism backs up the to-be-updated persistent data *before* performing the update. Therefore, crash consistency mechanisms introduce additional stalls to the program execution, as they need to maintain a correct persist ordering [54, 66, 72, 82, 91]. Second, these crash consistency mechanisms usually make extra copies of

*This author contributed to this work at the University of Virginia.

This work is licensed under a Creative Commons Attribution International 4.0 License.

EuroSys '23, May 8–12, 2023, Rome, Italy
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9487-1/23/05.
https://doi.org/10.1145/3552326.3587456

data [12, 14, 17, 37, 40, 47] in order to recover in case of a failure. Such data movement introduces additional memory bandwidth utilization. Combining these two performance bottlenecks, crash-consistency mechanisms can place extra data-intensive operations on the critical path.

Near-data processing (NDP) is an emerging computer architecture design that has the potential to mitigate these overheads. By bringing computation closer to data, NDP can reduce the data movement between memory and processor (e.g., CPU) [22, 35, 52, 62]. In particular, as the new CXL standard [16] is around the corner, more opportunities for processing closer to PM devices are opening.

In this work, we propose NearPM, a near-data processing system for PM-based storage-class applications. NearPM is integrated into PM memory devices, with access to the full memory of that device. However, it is not straightforward to accelerate PM programs as each program may follow a different crash consistency mechanism—simply adding hardware acceleration logic for each mechanism is impractical. We observe that different crash consistency mechanisms share common primitive operations. For example, undo-logging copies the original persistent data to a log and updates the persistent data in-place. On the other hand, redo-logging first updates logs and then copies the logs back to the original location. These two crash consistency techniques both copy data to a log and update PM. Thus, by supporting common accelerable primitives in the hardware and using them as building blocks, NearPM is capable of accelerating various crash consistency mechanisms.

Accelerating primitive operations speeds up the execution but the offloaded execution still needs to satisfy the ordering constraints. Naively, one could enforce the same ordering guarantees as the original CPU-based program but at the cost of excessive CPU-NDP synchronization. To overcome the synchronization overheads, our approach is to handle ordering near memory, enabling NDP execution to overlap with CPU procedures.

However, handling ordering near memory has new challenges. Ordering guarantees must be satisfied even though the execution is partitioned between the CPU and the NearPM. For example, when offloading undo-logging to NearPM, the log must be persisted by NearPM to memory before the CPU performs an in-place update. Furthermore, the program execution is not only partitioned between CPU and NearPM but may also happen across multiple NearPM devices. For example, two interleaved NearPM devices may both hold a fraction of the persistent data and the execution flows on both devices can be out of sync. Consider again the undo-log example, assume that some object is interleaved among two devices and that one device has the update committed but the other is still backing up data to the log. When a failure occurs, recovery might keep the updates in one device but roll back updates on another, leading to inconsistencies.

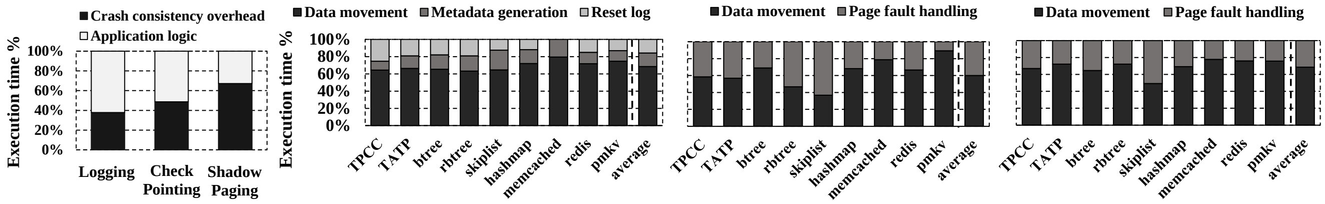
To overcome the challenges we define Partitioned Persist Ordering (PPO) for correct offloaded execution in NDP-enabled PM systems. PPO defines persist ordering in two scenarios. The first scenario is the order between the CPU and NearPM operations. Naively, one would enforce strict persist ordering between CPU and NearPM to provide the same crash consistency guarantees as the original program. However, this approach offsets the performance benefits of NDP. We observe that persists from NearPM to memory addresses that are managed by NearPM only but not shared with the CPU, such as logs, do not need to follow the original ordering constraints. Therefore, such a relaxed persist ordering mitigates CPU-side stalling and further allows NearPM operations to execute in parallel. For example, without back-and-forth synchronization, the CPU-side procedure can issue multiple independent logging operations to NearPM and have them executed in parallel.

The second scenario is the synchronization among multiple NearPM devices. Naively, frequent synchronizations among NearPM devices after every offloaded crash consistency operation keep them at the same pace and ensure their completion before committing updates. Similar to the case of CPU-NDP ordering, such a naive solution degrades performance benefits from NDP. Because NDP-managed memory, such as logs, would not be accessed nor exposed to the CPU-side procedure unless recovery happens. Therefore, it is possible to delay the synchronization and move it off the critical path of the program execution. As long as data needed for recovery remains intact until the completion of a series of PM operations (e.g., commit in a PM transaction), data can still recover successfully.

With the key insights above, we provide primitive operations in crash consistency mechanisms to cover a wide range of PM-base storage-class applications and introduce PPO to mitigate overheads stemming from the crash consistency guarantees in partitioned execution of NDP systems. To evaluate our design, we prototype NearPM on an FPGA platform that connects to the host CPU through PCIe. [1] We emulate PM using the on-board memory on the FPGA platform, allowing the CPU and the NDP device on the FPGA to access memory using loads/stores. We evaluate nine PM-optimized workloads with two NearPM devices. Each workload has implementations for logging, checkpointing, and shadow paging. The main contributions are the following:

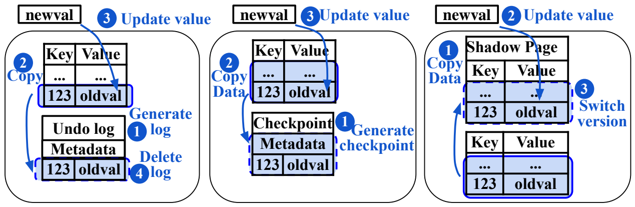
- We propose a near-data processing architecture that accelerates crash consistency mechanisms, by supporting common accelerable primitives.
- We define *partitioned persist ordering* (PPO) that defines the persist ordering in NDP systems. PPO ensures correctness

[1] The software and hardware implementation of NearPM are available at https://github.com/Systems-ShiftLab/NearPMSW and https://github.com/Systems-ShiftLab/NearPMHW, respectively.



(a) Crash consistency overhead **(b)** Logging **(c)** Checkpointing **(d)** Shadow paging

Figure 1. Crash consistency overheads (a) and the breakdown in logging, checkpointing, and shadow paging (b–d).



(a) Logging (undo) **(b) Checkpointing** **(c) Shadow paging**

Figure 2. Procedures in crash consistency mechanisms.

and performance when execution is partitioned among the CPU and multiple NDP devices.

- We prototype NearPM using an FPGA platform. Our evaluation shows that NearPM reduces the crash consistency overhead by 6.97×, 4.26×, and 9.76× in logging, checkpointing, and shadow-paging-based programs, compared to the CPU-only baseline. In terms of the end-to-end performance in the whole program, it achieves 1.35×, 1.22×, and 1.33× speedup over the baseline, respectively.

2 Background and Motivation

2.1 Crash Consistency and PM Programming

Persistent memory technologies (PM) feature high performance, data persistence, and byte-addressable direct access to persistent data bypassing the file system. Intel Optane PM [38] is a memory module that shares the memory bus with DRAM modules; other upcoming PM technologies [7, 10, 76] will be on the PCIe bus but enable direct access via the Compute Express Link (CXL) [16].

Direct access to persistent data reduces overhead on the data path, but at the same time moves the burden of managing data recovery to applications. We refer to the ability to restore persistent data after a failure (e.g., a power outage or system crash) as the *crash consistency guarantee*. Past research on crash-consistent programming has proposed various mechanisms for the crash consistency guarantee, such as undo-logging redo-logging [11, 12, 37, 40, 54], checkpointing [23, 51], and shadow paging [17, 68].

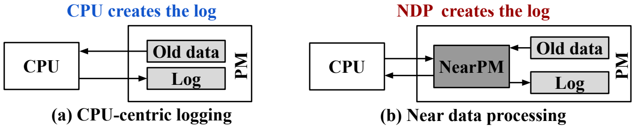
Logging replicates persistent data to a separate location (e.g., an undo or a redo log) before updating the persistent state. As shown in Figure 2a, undo-logging makes a fine-grained snapshot of the original data in a log, before persisting the in-place updates; it deletes the log only after the latter completes. Similarly, redo-logging redirects each update to a log, and applies the updates in-place only after the log has become persistent. Checkpointing (Figure 2b) maintains a coarse-grained snapshot of persistent locations prior to updates. Shadow paging first redirects the update to a newly allocated page and then changes the references to the original page to the new version (Figure 2c).

These mechanisms introduce performance overheads. The main overhead is due to intensive data movement. Figure 1a shows that logging, checkpointing, and shadow paging mechanisms take up 37.7%, 48.6%, and 67.2% of the execution time, respectively (methodology in Section 8.1). Figures 1b, 1c, and 1d further break down the crash consistency overhead, showing that 68.9%, 60.4%, and 70.5% of the overhead is from data movement in these crash consistency mechanisms, respectively. Thus, there is a huge opportunity for acceleration.

2.2 Near-Data Processing (NDP)

In traditional systems, the CPU is in charge of manipulating data. For instance, to create a copy of data in memory (as shown in Figure 3a), the CPU needs to fetch data through the cache hierarchy and write it to another CPU-manged memory location, leading to a high data movement overhead. To mitigate data movement overheads, researchers have introduced the paradigm of near-data processing (NDP) that places computation closer to the data [2, 22, 26, 27, 53, 65, 81, 93].

NDP is well-suited for applications or code regions that are memory-intensive and feature high parallelism. Therefore, it has the potential to mitigate the overhead of memory-intensive operations involved in crash consistency mechanisms. The existing and upcoming PM devices are also capable of hosting computation near or inside the PM device. For example, PCIe-based PM devices can integrate compression, query processing, and data movement logic [3, 7]; even the more compact PM devices, such as Optane DIMMs, already integrates controllers for data-intensive tasks [86] such as encryption, which can be extended to NDP. Figure 3b illustrates how an NDP unit copies data to a log without going through the CPU.



(a) CPU-centric logging

(b) Near data processing

Figure 3. CPU- and NDP-based log generation.

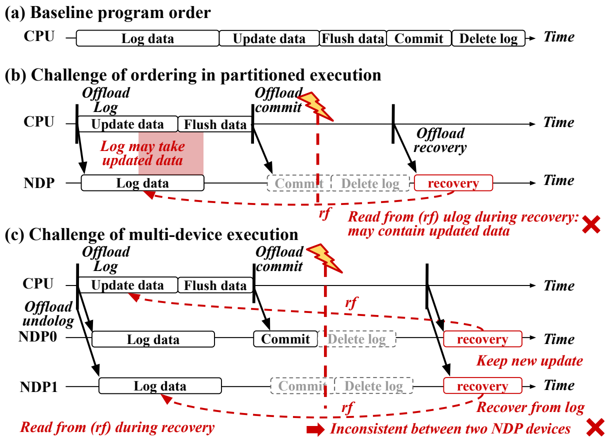


Figure 4. Challenges of ordering in partitioned execution

2.3 Challenges of NDP Crash Consistency

In this section, we discuss the challenges in supporting NDP processing for crash consistency operations.

Support for different crash consistency mechanisms. Although NDP can effectively accelerate memory-intensive procedures, NDP units are highly specialized. As a result, it is challenging to accelerate general programs, as programmers first need to identify NDP-friendly code regions and convert them based on NDP-accelerated hardware primitives.

As explained in Section 2.1, there is a diversity of crash consistency mechanisms, such as undo- and redo-logging, checkpointing, and shadow paging. Supporting every single one with its own dedicated acceleration logic is not realistic. Therefore, it is necessary to find a common ground in order to integrate NDP into a practical system. We will discuss our high-level ideas of NDP acceleration for crash consistency operations in Section 3.1.

Ensuring persist ordering near memory. As shown in Figure 2, crash consistency mechanisms enforce persist ordering. When offloading computation to an NDP-enabled PM device, program execution becomes *partitioned* between the CPU and the PM device. Figure 4 shows how a conventional CPU-centric system strictly orders undo-logging. Figure 4b offloads undo-logging to an NDP, while the other steps remain executed on the CPU. Such a partitioned execution breaks the ordering guarantees: the CPU concurrently persists to PM, while the NDP unit is creating an undo log. In case of a failure (as indicated by the red line), as the update was not committed, recovery attempts to read from (as indicated by the rf edge) the undo log. Due to incorrect ordering, the undo log might contain already updated data, leading to an inconsistent recovery.

Table 1. Evaluated crash consistency mechanisms.

Crash consistency mechanism	Common operations
Logging (undo) [11, 12, 14, 15, 21, 32, 40, 54, 58]	allocate, generate metadata, copy data, delete log, commit
Logging (redo) [29, 63, 82, 87]	
Logging (undo+redo) [14, 40]	
Checkpointing [5, 23, 30, 51, 71, 73, 82]	allocate, generate metadata, copy data
Shadow paging [37, 68, 69, 88]	allocate, copy data, switch page

In addition, the execution may become partitioned among multiple NDP devices. Figure 4c shows a scenario where two PM devices interleave. As such, a PM object can span both devices. The NDP units on the two devices operate on this partitioned object. However, without synchronizing the execution between them, the offloaded execution might progress at a different pace on both devices, a failure indicated by the red line, one PM device NDP0 has committed the update, but the other (NDP1) has not. As a result, during failure recovery, NDP0 maintains the in-place updates, as they were committed prior to failure, while NDP1 reads from the old copy in the log. Thus, the recovered data is inconsistent: partly from the original version and partly from the updated version. We will discuss our high-level ideas that ensure correct persist ordering between CPU and NDP devices, and among NDP devices in Section 3.2.

3 High-level Ideas

In this section, we will first discuss our high-level ideas of accelerating crash consistency operations with NDP, and then a persist ordering for partitioned execution among CPU and NDP devices.

3.1 Acceleration for Crash Consistency Operations

Our first insight is to divide the different crash consistency operations into smaller primitives, as shown in Table 1. We observe that different crash consistency techniques consist of common accelerable routines. For example, undo- and redo-logging both contain the following primitive operations: generate metadata (e.g., object ID, commit status, offset in PMDK [40] logs), copy data, and delete metadata. Therefore, our first key idea is to identify accelerable primitives in existing crash consistency mechanisms and implement NDP logic for these primitives in hardware near memory. We use the term *NDP procedure* to describe the execution of a series of consecutive NDP primitives that correspond to a crash consistency operation.

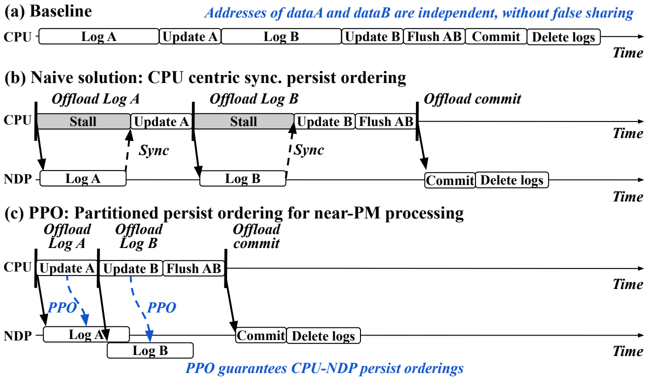


Figure 5. Partitioned execution between CPU and NDP.

Figure 5a and Figure 5b compare executions of two undo-logging procedures with and without NearPM. Figure 5a, everything goes through the CPU. All operations are ordered sequentially (even when independent operations Update A and Log B), and the memory-intensive operations creating and deleting the logs use the CPU. In Figure 5b accelerable primitives executed near-data. For example, copying data to Log A and Log B will take a shorter amount of time because they are NDP-friendly. Because persistent ordering is handled by the CPU when executing operations near memory, NDP devices need to synchronize with the CPU. Furthermore, the CPU might remain idle while the NDP operations are completed. This issue leads to our second key idea which is handling ordering near memory.

Handling ordering near memory removes the overhead of synchronization between NearPM and the CPU. In addition, this approach allows for more relaxed persistency, allowing the CPU and NearPM to execute in parallel. Figure 5c shows the benefit of handling ordering crash-consistent programs from the NDP side. Ordering PM programs near memory is not free because of the partitioned nature of the execution.

3.2 Persistent Ordering for Partitioned Execution

Executing crash consistency operations near memory partitions the program execution. To ensure correct execution and failure-recovery, we propose *partitioned persist ordering* (PPO) that ensures persist ordering between CPU and NDP, as well as among different NDP devices.

Persistent Ordering between CPU and NDP device. The major challenge demonstrated in Figure 4a lies in maintaining the persist ordering between the CPU and the NDP. To enable a correct persist ordering, a naive solution is to synchronize between the CPU and the NDP device actively. As Figure 5b illustrates, with such a naive solution, execution on the CPU side needs to wait until the execution on NDP has been completed. Though NDP procedure is faster than the CPU-only baseline (as illustrated in Figure 5a), frequent synchronization offsets the performance benefits.

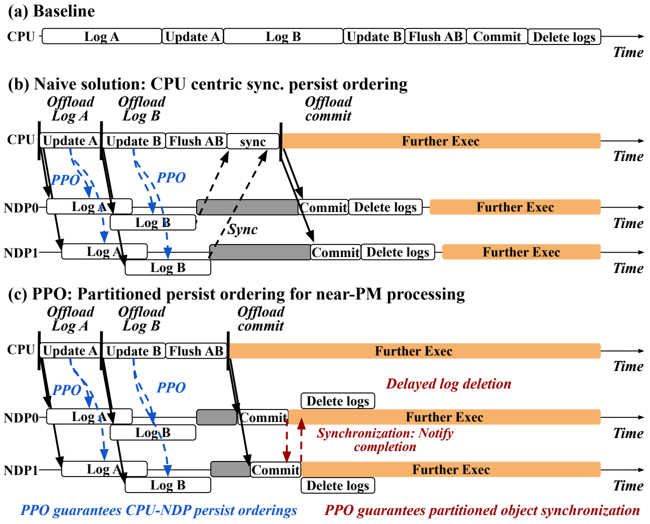


Figure 6. An undo-logging example in multi-device partitioned execution.

However, we observe that maintaining such a strict ordering is not always necessary. The partitioned execution on NDP does not always share the memory with the CPU. In the example of Figure 3b, the NDP procedure reads from memory but copies and persists it to a *separate memory location* that is only managed by NDP, i.e., an undo log. Therefore, the order of persists to the NDP-managed memory can be relaxed. The CPU-side update (e.g., “Update A” and “Update B” in Figure 5c) needs to persist *after* the associated NDP logging operations (e.g., “Log A” and “Log B” in Figure 5c). While, independent NDP operations, such as logging different addresses can happen in parallel, without being blocked by the CPU. In other crash consistency mechanisms, we observe similar opportunities. For example, a page-grained checkpointing operation on NDP only needs to *persist before* any update from the CPU toward the same page, while independent checkpointing operations can persist in parallel as they write to a separate memory location. Based on this observation, we see the opportunity to overcome the strict ordering between CPU and NDP units to exploit parallelization.

Synchronization among multiple NDP devices. In addition to CPU and NDP ordering, partitioned execution presents another ordering challenge among multiple devices because a persistent object may span multiple devices [46]. The persist ordering among multiple devices is another challenge because the execution is asynchronous among devices and their programs do not necessarily stay at the same pace. A naive way of maintaining the persist ordering among multiple NDP devices is to actively synchronize all NDP devices

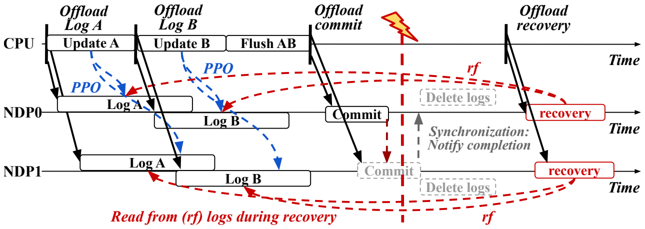


Figure 7. Recovery in multi-device partitioned execution.

after each crash consistency operation to make sure operations on all devices are complete before committing the updates. As demonstrated in Figure 6b, before updating the data in place (A or B), the CPU stalls before sending a commit operation to the NDP devices, until logging operations on both NDP devices have been completed. Thus, this naive solution avoids the unrecoverable scenario in Section 2.3, as the recovery program either recovers the logged data or keeps the in-place updates. Compared to the CPU-centric baseline in Figure 6a, even though this naive solution already provides better performance, both the CPU and the NDP devices still stall for synchronization.

We further observe that the data required for recovery is only managed by NDP and never exposed to the CPU, unless the recovery procedure happens. In the example of Figure 6c, if we relax the persist ordering between crash consistency operations and the later commit, and delay the synchronization among devices, the recovery program can still read from consistent data as long as the data required for recovery is not deleted before the delayed synchronization has completed (i.e., "Delete logs" for A and B). In Figure 7, if a failure happens when NDP0 has committed the update but NDP1 has not, the recovery procedure can still read from the consistent copy in the log in both NDP devices, as "Delete logs" on both devices only persists after a synchronization. At the same time, because the synchronization is delayed, it avoids additional stalling on CPU or NDP devices.

4 Partitioned Persist Ordering

In this section, we will provide more formal definitions for PPO in two scenarios: ordering between CPU and NDP, and among NDP devices.

4.1 CPU-NDP Ordering

We first denote an NDP procedure, N, that performs a sequence of memory accesses offloaded from CPU. Then, we define basic memory operations to memory address x:

- R_x and W_x: read and write memory accesses, respectively (from either CPU or NDP).
- $R_{x,CPU}$, $W_{x,CPU}$, $R_{x,NDP}$, and $W_{x,NDP}$: CPU read, CPU write, NDP read, and NDP write accesses, respectively.
- $R_{x,NDP} \in N$ and $W_{x,NDP} \in N$ stand for memory read or write issued by NDP procedure N to memory address x.

Then, we define ordering between memory accesses and NDP procedures:

- $\xrightarrow{po}$ denotes program order.
- $\xrightarrow{hb}$ denotes happens-before order.
- $\leq_p$ denotes persist-ordering.

PPO separates out memory addresses that are only managed by NDP procedures, without sharing with CPU. NDP accesses to these memory addresses thus do not need to order with memory accesses from CPU. Maintaining correct execution between CPU and NDP fundamentally depends on two invariants. The first invariant concerns the read-write dependency that ensures that execution on CPU or NDP always accesses data in the intended order that was defined by the program. The second invariant concerns persistence, as a correct recovery relies on enforcing persist ordering between writes.

Invariant 1: read-write ordering. In memory addresses shared between CPU and NDP, reads and writes issued by an NDP procedure are strictly ordered with the CPU. Let us define any read or write from the CPU as $M_{x,CPU}$: $M_{x,CPU} \in \{R_{x,CPU}, W_{x,CPU}\}$, where x is shared between CPU and NDP, i.e., $x \in NDP \bigwedge x \in CPU$. Likewise, we define $M_{y,NDP}$ that can be either read or write to memory address y that is shared between both CPU and NDP. Then, $\forall M_{y,NDP} \in N, M_{x,CPU} \xrightarrow{po} N \Rightarrow M_{x,CPU} \xrightarrow{hb} M_{y,NDP}$, and $N \xrightarrow{po} M_{x,CPU} \Rightarrow M_{y,NDP} \xrightarrow{hb} M_{x,CPU}$. In essence, within these shared addresses, memory accesses from NDP follow the program order with respect to the CPU.

Whereas, addresses that are only managed by NDP only need to follow the program order within an NDP procedure. Let $M_{a,NDP} \in \{R_{a,NDP}, W_{a,NDP}\}$, where $a \in NDP \bigwedge a \notin CPU$ and $M_{b,NDP} \in \{R_{b,NDP}, W_{b,NDP}\}$, where $b \in NDP \bigwedge b \notin CPU$ be two memory accesses issued by an NDP procedure N to addresses only managed by NDP. Then, $M_{a,NDP} \xrightarrow{po} M_{b,NDP} \rightarrow M_{a,NDP} \xrightarrow{hb} M_{b,NDP}$.

Invariant 2: persistence. Like before, we discuss both NDP-CPU shared memory and NDP-managed memory. For memory addresses x and y that are shared between CPU and NDP, the persist ordering follows the program order as well: $\forall W_{y,NDP} \in N, W_{x,CPU} \xrightarrow{po} N \Rightarrow W_{x,CPU} \leq_p W_{y,NDP}$, and $N \xrightarrow{po} W_{x,CPU} \Rightarrow W_{y,NDP} \leq_p W_{x,CPU}$.

Writes to NDP-managed memory that is not shared with CPU, say z (e.g., logs, checkpoints, and shadow copies) follow relaxed persist ordering. Writes from NDP can delay their persistence, as the CPU cannot access these addresses: $\forall W_{z,NDP} \in N, N \xrightarrow{po} M_{x,CPU} \Rightarrow W_{z,NDP} \not\leq_p M_{x,CPU}$.

4.2 Multiple-Device Synchronization

In addition to the definitions in Section 4.1, we define the following:

- F: an event of system failure.
- N_A: an NDP procedure executed on NDP device A, which may be interrupted by a failure F. Read or write accesses to memory address x that are issued by N_A are denoted as $R^A_{x,NDP}$ and $W^A_{x,NDP}$.
- $R^A_{x,NDP} \xrightarrow{rf} W^A_{x,NDP}$: NDP read $R^A_{x,NDP}$ accesses memory address x that was persisted by a write $W^A_{x,NDP}$ before a failure F, to recover an interrupted NDP procedure N_A.
- S: a synchronization event that enforces completion and persistence of memory accesses among all NDP devices.

When executing a procedure on a shared object between two NDP devices A and B, if device A synchronizes with B using a synchronization event S, then any memory access from the executions of N_A and N_B, may not persist before synchronization is complete, i.e., $S \xrightarrow{po} W_{x,NDP_A} \wedge S \xrightarrow{po} W_{y,NDP_B} \Rightarrow S \leq_p W_{x,NDP_A} \wedge S \leq_p W^B_{y,NDP}$.

Invariant 3: Persist before synchronization. Before synchronization, any memory access from NDP procedures N_A and N_B must be persisted, i.e., $W^A_{x,NDP} \xrightarrow{po} S \wedge W^B_{y,NDP} \xrightarrow{po} S \Rightarrow W^A_{x,NDP} \leq_p S \wedge W^B_{y,NDP} \leq_p S$. Based on this guarantee, we next discuss the correctness of failure-recovery.

Invariant 4: Failure-recovery. When the failure happens before synchronization, i.e., $F \xrightarrow{hb} S$, the recovery procedure on each NDP device reads from data that has been persisted before failure for recovery. Say, on NDP device A, $R^A_{x,NDP} \xrightarrow{rf} W^A_{x,NDP}$, where $W^A_{x,NDP}$ has persisted data for recovery. As PPO enforces persist ordering between writes from NDP and CPU, $R^A_{x,NDP}$ is guaranteed to read consistent data. And the same guarantee applies to device B. When a failure happens after synchronization, i.e., $S \xrightarrow{hb} F$, because all prior memory operations have become persistent, the recovery procedure also reads consistent data.

5 NearPM Hardware Design

5.1 Architecture of NearPM

NearPM is placed inside the PM controller of the PM device, with direct access to the PM storage medium. This enables NearPM to access PM with higher bandwidth and lower latency than the host processor. NearPM consists of the following major components (Figure 8):

- **Host read/write queue** takes regular reads and writes from the host processor and accesses the PM media.
- **Request FIFO** takes requests issued by the host processor and keeps them until they are executed.

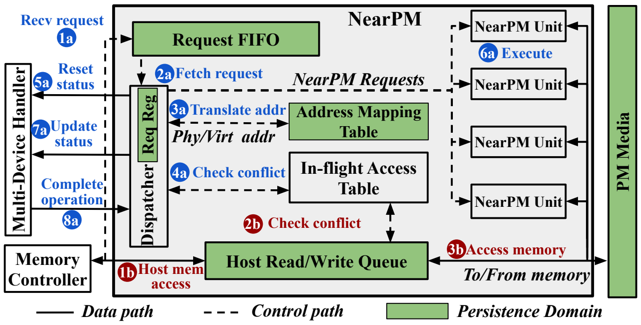


Figure 8. High-level architecture of a NearPM device.

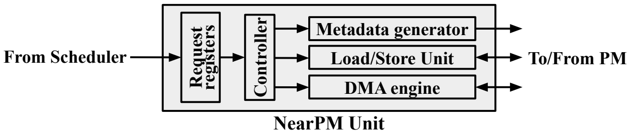


Figure 9. Components in each NearPM unit.

- **Dispatcher** decodes and issues requests to *NearPM units* (i.e., execution engines).
- **Address mapping table** converts virtual addresses in the requests to physical addresses, as the parameters NearPM commands are virtual addresses (details in Section 5.4).
- **In-flight memory access table** keeps track of memory addresses being accessed by the NearPM units in order to handle accesses with conflicting addresses. In case an operation attempts to access an address that is being written to, the *Dispatcher* stalls this operation and buffers it in the *Host read/write queue.*
- **NearPM units** are processing engines that manipulate data in PM and are controlled by the *Dispatcher*. Each *NearPM unit* has a request register that stores the request from the *Dispatcher*, a controller which converts requests into control signals, a metadata generator (e.g., metadata generation and log deletion), and a load/store unit for fine-grained data movement, and DMA engine for large data movement (e.g., data copy), as shown in Figure 9.
- **Multi-device handler** stores the status of other NearPM devices and coordinates among them. A command execution is complete when all devices have completed execution. It keeps track of all *NearPM units*, issuing a request as soon as one of them is available.

5.2 NearPM Execution Flow

In this section, we further introduce the execution flow of NearPM that handles program-offloaded crash consistency operations (i.e., NearPM requests) and services the regular memory accesses from the host processor.

NearPM request execution. Figure 8 shows the workflow (steps in blue). A NearPM request first enters the *Request FIFO* (step 1a) and then gets decoded by the *Dispatcher* (step 2a). During decoding, the *Dispatcher* translates request operands from virtual to physical address through an *Address Mapping Table* (step 3a). After translation, the *Dispatcher* checks the request's physical address (step 4a)—requests without address conflicts are immediately issued, but stall until the completion of the other conflicting request/access (details in Section 5.3.1). Next, NearPM resets the status bit in *Multi-device handler* (step 5a). Then, NearPM unit receives the request and starts the execution immediately (step 6a). Upon completion, NearPM notifies the *Multi-device handler* to update the status bit both locally and in other NearPM devices (step 7a). When all NearPM devices have completed execution, the *Multi-device handler* notifies the *Dispatcher* (step 8a) to assign new commands to the NearPM unit.

Host memory access. Figure 8 (steps in red) describes the execution for CPU's memory accesses. CPU's memory accesses enter the host read/write queue (step 1b). Like before, the *Dispatcher* also checks the CPU's accesses for address conflicts before dispatching (step 2b). It issues memory access immediately if there is no conflict from the *In-flight Access Table* (step 3b), Otherwise, it buffers CPU's access until the other access has completed.

5.3 Correctness Guarantees

5.3.1 CPU-NDP Ordering. NearPM implementation follows PPO. We first discuss the implementation that ensures CPU-NDP ordering.

Invariant 1 is ensured by the *Dispatcher* (Section 5.1). When NearPM dispatches a request to a NearPM unit for execution, it updates addresses in the *NearPM access table* (Figure 10) When the CPU accesses PM (Figure 10 step 1b), NearPM checks if there are ordering dependencies between inflight NearPM execution and incoming CPU memory accesses (step 2b). If a conflict is detected, NearPM buffers incoming memory access from the host (step 3b) until the conflicting access is completed by the NearPM units (step 4b). To avoid ordering invariants violations between NDP procedures in a single NearPM device, the *Dispatcher* checks the lookup table for conflicts between memory ranges accessed by the pending and in-flight requests (step 2a). If there is a conflict, the *Dispatcher* delays the issue of pending requests (step 3a) until the conflicting access has completed (step 4a).

Invariant 2 is ensured by writing back all updates to PM on the CPU side before invoking an NDP procedure. As there is no write caching in the NDP device, as soon as NDP issues a write access to PM, it enters the persistence domain.

5.3.2 NDP-NDP Ordering. PPO enables delayed synchronization because writes to data required for recovery do not need to complete immediately. Therefore, synchronization is not on the critical path. We take an approach described in Figure 11 to coordinate the completion of requests among devices. When the program issues a NearPM request that operates on a persistent object spanning multiple devices, similar to issuing memory requests in interleaved memory modules, the memory controller sends the NearPM request to all interleaved NearPM devices according to their address ranges. Each NearPM device has a *Multi-device handler* that keeps track of the status of each command in local NearPM execution logic as well as in other NearPM devices. After NearPM starts execution, it waits for the completion status from other NearPM devices (step 1) and its local execution (step 2). Finally, NearPM removes or resets the data required for recovery, which is not on the critical path of execution. In this way, the delayed synchronization mitigates the performance overheads.

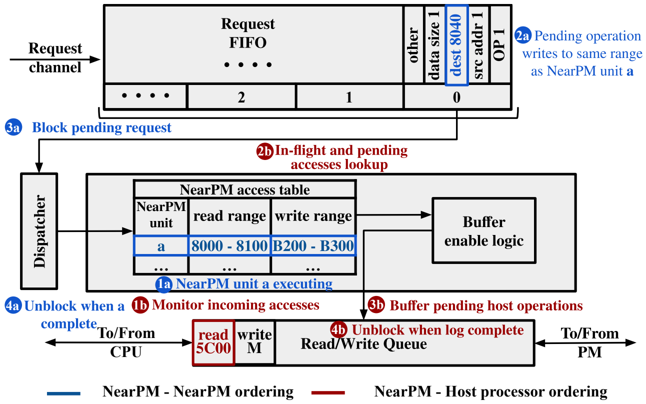


Figure 10. Ordering handling in a single NearPM device.

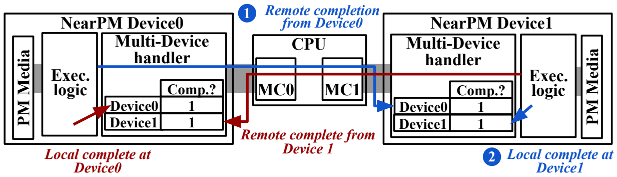


Figure 11. Hardware for cross-device synchronization.

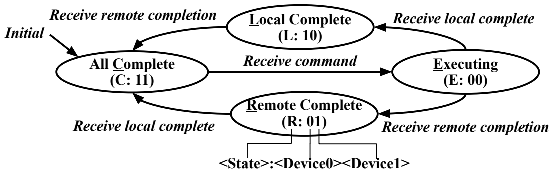


Figure 12. Synchronization state machine of partitioned execution on two devices.

PPO maintains a state machine to keep track of the synchronization status during partitioned execution to meet invariant 3. Figure 12 shows a state machine for a two-device setup. The state machine starts from the *All Complete (C)* state until a command that was duplicated to execute in two devices is received. Then the state machine changes its state to *Executing (E)* and keeps monitoring for *Receive local complete* or *Receive remote complete* signals from local execution or remote execution. After receiving the command complete signals from all devices, it will return back to the *All Complete* state. When both devices reach *All Complete* state, writes before this point have become persistent.

5.3.3 Recovery. Failure-recovery is another aspect of correctness. To satisfy Invariant 4, the NDP system keeps in-flight operations in a persistence domain and restores them after failure.

Persistence domain. PM hardware systems employ extended persistence domains (e.g., ADR [64] and eADR [75]) that include not only the PM devices but also buffers/caches in the processor. As NearPM executes the crash consistency operations in the PM module and services regular memory accesses from the host processor, these operations and memory access requests should also be placed in the persistence domain, in case they are not completed before failure. Figure 8 marks the hardware components of NearPM within the persistence domain in green: Request FIFO (2 kB), Address Look-Up Table in the Address translator (432 Bytes), In-flight request registers (256 Bytes) in the Dispatcher, and Host Read/Write Queue (4 kB). Those structures have a total capacity of 7 kB, much less than the buffers (tens of kBs) in existing Optane PM modules [86]). Thus, it is practical to use residual capacitors similar to existing Optane PM to write back structures in the persistence domain to a reserved PM location upon failure.

Recovery procedure. After the system is up again, the hardware of a NearPM device ensures that the results of in-flight NearPM requests and pending host memory accesses in the persistent domain are visible to the recovery program. In a case where there are multiple NearPM devices in the system, the recovery program needs to determine the progress made by each device prior to failure—the latest synchronization point that all NearPM devices reach before failure happens. The recovery procedure of NearPM hardware includes two steps: (1) NearPM loads the data from the reserved PM region back to the structures in the persistence domain. (2) NearPM replays the in-flight NearPM requests and host memory accesses until it reaches the latest synchronization point. Thus, the results of all in-flight operations prior to the synchronization point are visible in memory.

5.4 Address Translation

Address translation has always been a challenge in NDP systems [9, 22, 26, 26, 27, 35, 36, 52, 53, 53, 65, 80, 81, 89, 93] as structures such as TLB are in the host processor. Fortunately, PM libraries (e.g.[40]) usually allocate PM as pools and a memory access to the pool manifests as a base address plus an offset within the pool. Prior works have shown that as long as a pool's base address is translated, it is straightforward to also translate other memory addresses in the same pool using the offset value [83, 84, 92]. Therefore, NearPM keeps the translation of the base address for each pool and performs address translation without going through the CPU.

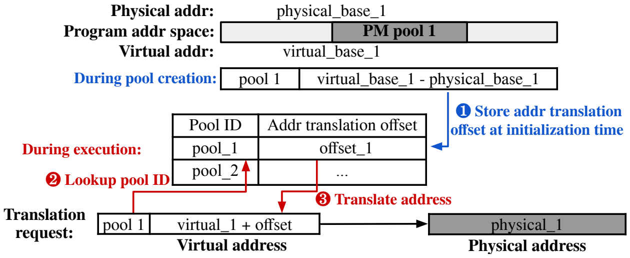


Figure 13. Address translation in NearPM.

Figure 13 shows the address translation procedure in NearPM. When the program creates a PM pool, NearPM first computes the offset between the virtual and physical addresses. The offset is then stored in the *Address Mapping Table* indexed by the pool ID (step ①). Because the addresses encoded in the command are from the virtual address space, to execute them, NearPM looks up the pool ID of the incoming request (step ②) and translates its virtual address to the physical address, by adding the offset to the incoming virtual address (step ③). When accommodating multi-threaded applications, in addition to the pool ID, thread ID is also used for indexing address translation offset.

Context switch handling. NearPM keeps the base address mapping for each PM pool. As each pool ID is unique in the system, even across a context switch, the pool-ID-indexed translation mapping still remains valid.

Multi-device support. A PM pool can span across multiple interleaved NearPM devices, where certain bits in the virtual address identifies which NearPM device the data locates. Based on these bits, each device contains a virtual-physical mapping for the base address that is mapped to its local device. Thus, the translation mechanism that relies on the base address of the pool still applies to multi-device scenarios.

6 NearPM Software Design

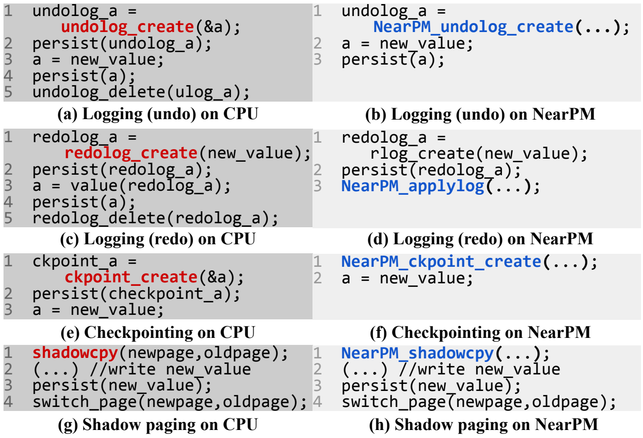
NearPM is a software-hardware co-design, including a software interface for developing programs and communication between the program and NearPM.

6.1 Software Interface

For the API to communicate with the NearPM hardware, there is a dedicated control path (Figure 8). This control path is memory-mapped using a separate address space from that of the PM. At the application level, the program needs to provide the device path of NearPM to `NearPM_init_device` which memory-maps this command path.

Table 2. NearPM software interface.

NearPM primitives	Arguments	Description
NearPM_undolg_create	pool_id, thread_id, old_data_ptr, size	Undo-logging: generate metadata and copy old data to an undo log
NearPM_applylog	pool_id, thread_id, redolog_ptr, size	Redo-logging: apply a redo log by copying data to the original location
NearPM_commit_log	pool_id, thread_id	Undo/redo-logging: delete and commit multiple logs in a PMDK transaction
NearPM_ckpoint_create	pool_id, thread_id, old_data_ptr, size	Checkpointing: generate metadata and copy existing data to a checkpoint before update
NearPM_shadowcpy	pool_id, thread_id, page_ptr, size	Shadow-copy: copy an existing page before update
NearPM_init_device	device_path	Device initialization: memory-maps NearPM's command interface



```
undolog_a =
    undolog_create(&a);
persist(undolog_a);
a = new_value;
persist(a);
undolog_delete(ulog_a);
```

(a) Logging (undo) on CPU

```
undolog_a =
    NearPM_undolog_create(...);
a = new_value;
persist(a);
```

(b) Logging (undo) on NearPM

```
redolog_a =
    redolog_create(new_value);
persist(redolog_a);
a = value(redolog_a);
persist(a);
redolog_delete(redolog_a);
```

(c) Logging (redo) on CPU

```
redolog_a =
    rlog_create(new_value);
persist(redolog_a);
NearPM_applylog(...);
```

(d) Logging (redo) on NearPM

```
ckpoint_a =
    ckpoint_create(&a);
persist(checkpoint_a);
a = new_value;
```

(e) Checkpointing on CPU

```
NearPM_ckpoint_create(...);
a = new_value;
```

(f) Checkpointing on NearPM

```
shadowcpy(newpage,oldpage);
(...) //write new_value
persist(new_value);
switch_page(newpage,oldpage);
```

(g) Shadow paging on CPU

```
NearPM_shadowcpy(...);
(...) //write new_value
persist(new_value);
switch_page(newpage,oldpage);
```

(h) Shadow paging on NearPM

Figure 14. Examples that demonstrate the use of NearPM's software interface.

The PM program running on the host processor uses a software API to issue commands. Table 2 lists the primitive functions and their parameters, which can be directly called in PM programs or libraries. In our evaluation, we implemented the APIs in the PMDK [40] library. The API is agnostic whether or not an operation is single or multi-device. The address range of the command operand is monitored by the memory controller; a command is duplicated if the operand object is shared among multiple devices. Figure 14 demonstrates code examples that show the use of these calls. Each primitive function in the API corresponds to the crash consistency mechanisms in Table 1. Besides the listed primitive functions, our prototyping system can also be extended to test and develop other crash consistency mechanisms.

6.2 Recovery

When recovering back from a system failure the software is responsible for initializing the hardware recovery procedure. The recovery program sends the command for system-wide recovery and the NearPM devices will individually run their own recovery procedures following the hardware steps discussed in Section 5.3.3 After completion, the PM program can start executing from the last consistent program point.

Table 3. System for evaluation.

System Configuration	
CPU	AMD Zen 2, 2 GHz, 8 cores
DRAM	4×16 GB DDR4
FPGA	Xilinx UltraScale+ VCU118 (Section 7)
PM	2 GB, Emulated with on-FPGA DRAM
NearPM	4 NearPM units, 32 entry request FIFO
Software System	
OS	Ubuntu 20.04, Linux kernel v5.3.0
Environment	gcc/g++-9.2, PMDK-1.9

7 NearPM Implementation

We use the Xilinx Virtex UltraScale+ VCU118 evaluation platform [90] to implement NearPM. The development board is attached to a PCIe 3.0 × 8 slot, with a bandwidth of 8 GB/s. We use 2 GB of the onboard DRAM to emulate PM on a NearPM device. In our evaluation, the access latency to the emulated PM is 436 ns, similar to real evaluations on Intel's Optane DCPMM [48]. NearPM connects to the CPU's memory controller via the PCIe bus. Constrained by the FPGA platform, we implement two NearPM devices on the same FPGA, with each device having four NearPM Units running at 300MHz. The emulated NearPM devices on the FPGA are mapped to a contiguous memory region. NearPM can only support interleaving which will result in a contiguous block in a given device. Scatter-gather operations are not supported. Each NearPM device contains four NearPM units, connected through an internal AXI bus of 4 GB/s bandwidth.

The DRAM on the FPGA board is mapped to the CPU's physical memory space in the write-back cacheable mode and is directly accessible through load-store instructions. However, the Linux kernel maps FPGA's memory as non-cacheable by default. Thus, we manipulate the memory type range register (MTRR) at the boot time to enable writeback caching. Thus, our implementation is a software-implemented coherent memory. In the upcoming CXL [16] systems, we expect even better performance.

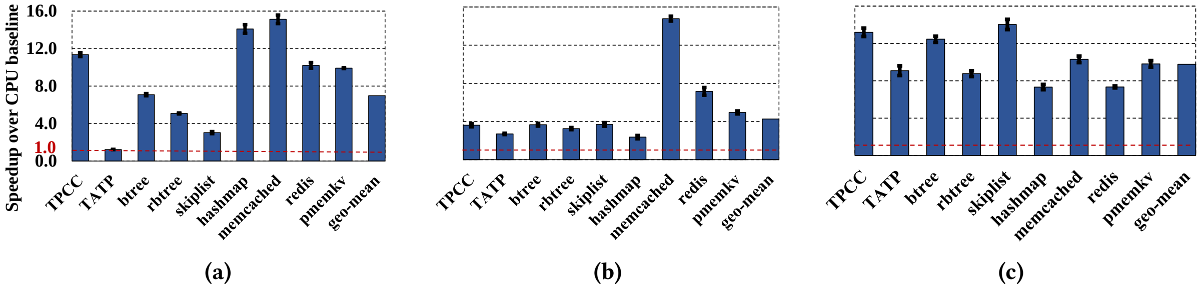


Figure 15. Speedup in code regions for crash consistency in (a) logging, (b) checkpointing, and (c) shadow paging.

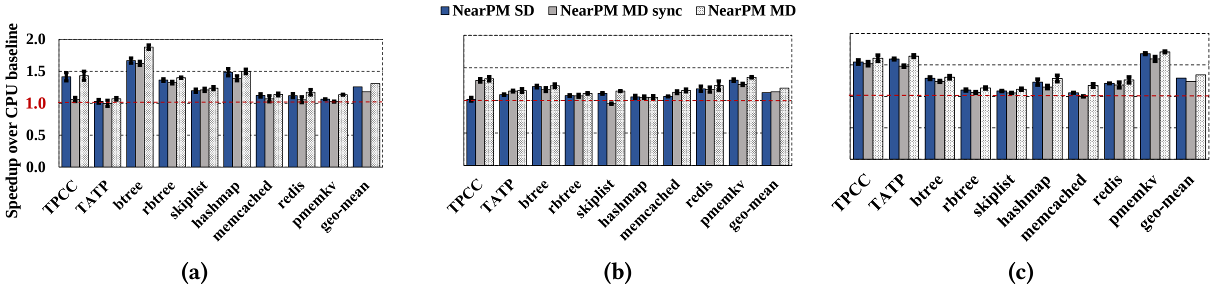


Figure 16. End-to-end speedup in (a) logging, (b) checkpointing, and (c) shadow paging.

Table 4. Workloads for evaluation.

Workload	Input
TPCC, TATP [32]	Process TPCC/TATP transactions
btree, rbtree, skiplist, hashmap [40]	Insert random key-values (value size is 64 B)
memcached [58]	100% write requst from YCSB [18]
redis [45]	100% write requst from YCSB [18]
pmemkv [42]	Input from PmemKV-bench [43]

8 Evaluation

8.1 Methodology

System configuration. We evaluate PPO on the prototype of NearPM (implementation in Section 7) in a testbed described with the system configurations in Table 3.

Workloads. Table 4 lists the workloads and their inputs. TPCC and TATP are PM transactions from a prior work [32]; btree, rbtree, skiplist, and hashmap are key-value stores from PMDK [40] library; Redis and Memcached are real-world workloads. PmemKV [42] is a key-value store that uses a B+ tree as the backend. For each workload, we evaluate three crash consistency implementations:

- **Logging:** The performance of each program's original crash consistency support based on undo/redo logging.
- **Checkpointing:** The performance of a modified crash consistency support based on checkpointing.
- **Shadow paging:** The performance of a modified crash consistency support based on shadow paging.

Note that both checkpointing and shadow paging operate at 4 kB page granularity.

Comparison points. We evaluate four configurations, where all experiments are evaluated 10 times. The error bars (standard deviation) are included in the figures.

- **Baseline** executes only on the CPU.
- **NearPM SD** offloads crash consistency operations to a single NearPM device.
- **NearPM MD SW-sync** offloads crash consistency operations to two NearPM devices and synchronizes using a CPU-polling, software mechanism.
- **NearPM MD** offloads crash consistency operations to two NearPM devices with delayed synchronization.

8.2 Speedup Evaluation

In this section, we evaluate applications (listed in Table 4) in the configurations mentioned in Section 8.1. We first demonstrate the benefit of NDP using a microbenchmark and demonstrate the parallelism in real applications. Then, we present the speedup of these applications of both crash consistency code regions and the whole programs.

8.2.1 Micro-benchmark. We evaluate NearPM with a micro-benchmark that copies persistent data. Figure 17 shows the speedup from NearPM. As data size increases, the speedup also increases: from 1.13× when the size is 64 B to 5.57× when copying 16 kB of data. This micro-benchmark does not introduce operation-level parallelism. Thus, the

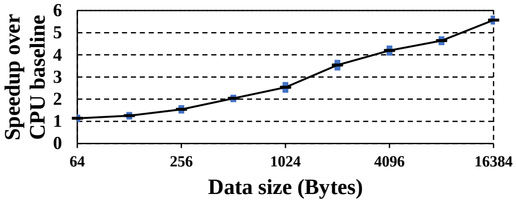


Figure 17. Data movement speedup of NearPM over CPU.

speedup is a result of the proximity to memory when copying data with NearPM. This speedup is comparable to prior FPGA-based NDP prototypes [33, 57, 70].

8.2.2 CPU-NearPM parallelism. We evaluate the benefit of parallelism between CPU and NDP devices, i.e., CPU and NearPM may execute at the same time for a certain fraction of the program. Figure 18 presents the average percentage of execution that is parallelizable between the CPU and NearPM of workloads in Table 4. On average, logging, checkpointing, and shadow paging have 20.01%, 17.25%, and 24.68% of the execution parallelizable, respectively.

8.2.3 Speedup in crash consistency operations. Figure 15 shows the speedup within code regions that maintain crash consistency. On average PPO achieves 6.9×, 4.3×, and 9.8× speedup for logging, checkpointing, and shadow paging, respectively. We notice that TATP has a low speedup of 1.23× in undo-logging. The main reason is that TATP has only one NearPM operation that performs logging and commits immediately afterward. Thus, it does not benefit from parallelism in NearPM execution.

8.2.4 Whole-application speedup. We then present the whole-application performance in Figure 16. NearPM SD achieves 1.29×, 1.15×, and 1.28× average speedup for logging, checkpointing, and shadow paging, respectively. This result shows the performance PPO achieved by effective handling of ordering between the CPU and NDP. NearPM MD SW-sync achieves 1.21×, 1.14×, and 1.23× average speedup for logging, checkpointing, and shadow paging, respectively. Due to the synchronization overhead, its speedup is lower compared to NearPM SD. By reducing the synchronization overhead, NearPM MD achieves 1.35×, 1.22×, and 1.33× speedup on average in the three crash consistency mechanisms, respectively.

8.3 Scalability Evaluation

Next, we evaluate the scalability of NearPM. First, we demonstrate the multithreading performance of two realistic workloads, Memcached [58], and Redis [45]. Then, we present the impact of the number of NearPM units on performance, by sweeping the number of units.

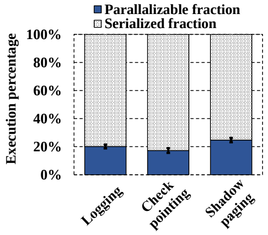


Figure 18. Percentage of parallel execution between CPU and NearPM.

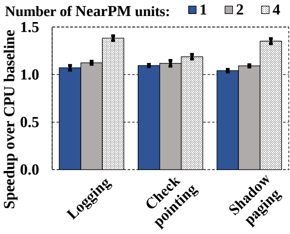


Figure 19. End-to-end performance with variable # NearPM Units.

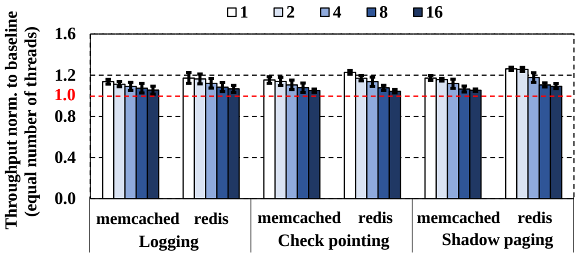


Figure 20. Throughput with multiple threads.

8.3.1 Multi-threaded Performance. This experiment evaluates the performance when the application on the host CPU is multithreaded. We scale Redis and Memcached with 1 to 16 threads, for both the number of clients and the backend handlers. Memcached access different memory pools for each thread, while Redis shares the same pool amongst multiple threads. Figure 20 presents the speedup over the CPU-based baseline with the same number of threads. As the number of threads increases, the speedup from NearPM reduces but still outperforms the baseline. This trend still holds even when we consider standard error for multiple runs as shown in black in Figure 20. The main reason for the slowdown is that the number of execution units in NearPM is limited to four due to the limitation of our FPGA platform. We expect commercialized systems to integrate more units for intensive workloads.

8.3.2 Sensitivity study on the numbers of NearPM units. This experiment compares the performance with 1, 2, and 4 NearPM units. Figure 19 shows that the average speedup over the CPU-based baseline increases with more NearPM units, as the offloaded program contains parallelizable operations, such as copying multiple cache lines in a page can happen in parallel.

9 Discussion

Scalability. Though in Section 8, we evaluated our prototype of two NearPM devices, due to limitations in our FPGA platform, PPO is scalable as synchronizations among devices are off the critical path. Scalability is critical to performance with CXL-supported systems.

Expected performance in commercial NDP systems. Our NearPM prototype shows comparable performance as prior work that also prototype NDP systems [33, 57, 70]—we achieve 7−9× speed when evaluating the offloaded crash consistency operations (i.e., accelerable computation kernels). In commercialized implementations, we expect better performance as the NearPM device can allow for more processing units that operate at a higher clock frequency.

Opportunity with CXL.. As CXL is around the corner, future-generation PM systems are expected to be a CXL-based instead of occupying DIMM slots. Though evaluated using a PCIe FPGA, NearPM is independent of the interconnect technology and can largely benefit from the hardware-based coherence support from CXL. In the current design, the software handles coherence by explicitly writing updated data back to NearPM devices. With CXL, we expect a much lower communication and synchronization overhead between NearPM devices and the host CPU.

Security considerations. NearPM target performance optimization using NDP while ensuring correctness. Thus, security in multi-tenant scenarios is not the focus. Nonetheless, to support multi-tenancy, the existing address-translation mechanism can be extended to support boundary checking by storing the pool size alongside the address translation offset. We expect future work to build upon our prototype.

NUMA support. NUMA systems are common in datacenters. Although our evaluation platform is single-socket, the design of NearPM fundamentally supports NUMA systems. The major challenge in NUMA system is that accesses to memory devices that belong to a different socket can experience a longer latency, increasing the variability of memory accesses. NearPM guarantees a correct ordering of PM accesses and NDP-offloaded operations. Therefore, NearPM is NUMA-safe.

10 Related Work

Near-data processing. NDP aims to reduce memory movement in the conventional CPU-centric systems [1, 2, 22, 24–27, 35, 36, 52, 53, 65, 80, 81, 93]. For example, RowClone [77] accelerates bulk data movement in DRAM and TETRIS [28] accelerates neural networks. There have also been works that bring processing to SSDs. For example, CompoundFS [74] accelerates file system IO operations in SSD and Almanac [85] retains SSD history logs using an in-SSD logic. However, they target conventional storage systems instead of PM systems that directly manage persistent data.

Hardware support for memory persistency. The memory persistency model ensures the order in which writes become persistent. Pelly et al. first propose memory persistency [72] and followup works continue to optimize the performance of persistency models. For example, DPO [55] and HOPS [66], PMEM-Spec [49], and Themis[78] provide efficient persistency models by reducing the cost of blocking due to data persistence. However, those works target CPU-centric systems. In comparison, our work, NearPM, extends persistence to NDP.

Crash consistency mechanisms. There are a number of previous works that provide solutions for crash consistency. Intel's PMDK library provides transactions using a combination of undo and redo logs [39]. There are also databases and key-value stores based on PMDK that maintain crash consistency, such as Redis [45], MongoDB [44], RocksDB [41], and Memcached [58]. Atlas [11] and SFR [32] convert code regions marked by synchronization primitives to undo-log-based transactions. Checkpointing creates a copy of the updated persistent memory to enable recovery [20, 50]. DudeTM [59] and SoftWrAP [29] use shadow memory to maintain redo logs before applying them to PM. These mechanisms tend to maintain additional copies of data for recovery. Therefore, NearPM, can be applied to mitigate their crash consistency overhead. Recently works also use software testing techniques to detect bugs in crash-consistent programs [19, 34, 60, 61, 67]. These techniques can also be applied to a NearPM system to detect misuse of offloaded crash consistency primitives.

11 Conclusions

In this work, we propose NearPM, an accelerator for crash consistency mechanisms in PM-based storage-class applications. To realize the full potential of acceleration, we move the ordering handling between CPU and NearPM near memory, using Partitioned Persist Ordering (PPO). We prototype NearPM on an FPGA platform and evaluate nine PM workloads, where each workload has three versions that use logging, checkpointing, and shadow paging for crash consistency. Overall, NearPM achieves 4.3 − 9.8× speedup in the NDP-offloaded operations and 1.22 − 1.35× speedup in end-to-end execution of the whole applications.

Acknowledgement

We thank the anonymous reviewers and the shepherd, Prof. Marc Shapiro, for their valuable feedback. This work is supported by the SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP) and the National Science Foundation (NSF).

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[3] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Data Management on New Hardware*, pages 1–5. 2022.

[4] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 2018.

[5] Katelin A Bailey, Peter Hornyack, Luis Ceze, Steven D Gribble, and Henry M Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[6] Jeff Barr. Now available – Amazon EC2 high memory instances with 6, 9, and 12 TB of memory, perfect for SAP HANA. https://aws.amazon.com/blogs/aws/now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfect-for-sap-hana/, 2018.

[7] Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. Cache-coherent accelerators for persistent memory crash consistency. 2022.

[8] Nan Boden. Available first on Google cloud: Intel Optane DC persistent memory. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory, 2018.

[9] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[10] Richard A. Brunner. CXL and the tiered-memory future of servers. https://www.lenovoxperience.com/newsDetail/283yi044hzgcdv7snkrmmx9ovpq6aesmy9u9k7ai2648j7or, 2021.

[11] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014.

[12] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 2015.

[13] Shimin Chen and Qin Jin. Persistent B+-Trees in non-volatile main memory. In *VLDB*, 2015.

[14] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *SOSP*, 2013.

[15] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[16] Compute Express Link. CXL 2.0 specification. https://www.computeexpresslink.org/spec-landing, 2021.

[17] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[18] Brian Cooper. Ycsb. https://github.com/brianfrankcooper/YCSB, 2019.

[19] Bang Di, Jia-Wen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[20] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *Proceedings of the conference on high performance computing networking, storage and analysis (SC)*, 2009.

[21] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *EuroSys*, 2014.

[22] Ivan Fernandez, Ricardo Quislant, Eladio Gutiérrez, Oscar Plata, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, and Onur Mutlu. Natsa: A near-data processing accelerator for time series analysis. In *IEEE 38th International Conference on Computer Design (ICCD)*, 2020.

[23] Pradeep Fernando, Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. Phoenix: Memory speed hpc i/o with nvm. In *IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016.

[24] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-memory compute using off-the-shelf DRAMs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[25] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. FracDRAM: Fractional values in off-the-shelf DRAM. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[26] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[27] Mingyu Gao and Christos Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[28] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3D memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[29] E. R. Giles, K. Doshi, and P. Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[30] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous checkpointing of htm transactions in nvm. In *ISMM*, 2017.

[31] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using Intel Optane DC persistent memory. *arXiv*, 2019.

[32] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *PLDI*, 2018.

[33] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *arXiv*, 2021.

[34] Hamed Gorjiara, Guoqing Harry Xu, and Brian 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.

[35] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent offloading and mapping (TOM) enabling programmer-transparent near-data processing in GPU systems. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[36] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *IEEE 34th International Conference on Computer Design (ICCD)*, 2016.

[37] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

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

persistent-memory.html.
[39] Intel. The libpmemobj library.
[40] Intel. Persistent memory programming. https://pmem.io/.
[41] Intel. pmem-rocksdb.
[42] Intel. pmemkv. https://github.com/pmem/pmemkv, 2018.
[43] Intel. pmemkv-bench. https://github.com/pmem/pmemkv-bench, 2018.
[44] Intel. PMSE - Persistent memory storage engine for MongoDB. https://github.com/pmem/pmem-rocksdb, 2018.
[45] Intel. Redis. https://github.com/pmem/redis/tree/3.2-nvml, 2018.
[46] Intel. Quick start guide: Configure Intel Optane™ DC persistent memory modules on Linux. https://software.intel.com/en-us/articles/quick-start-guide-configure-intel-optane-dc-persistent-memory-on-linux, 2019.
[47] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
[48] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. *arXic*, 2019.
[49] Jungi Jeong and Changhee 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.
[50] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *MICRO*, 2015.
[51] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using NVM as virtual memory. In *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.
[52] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.
[53] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies. *BMC genomics*, 2018.
[54] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
[55] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *MICRO*, 2016.
[56] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.
[57] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. Hardware architecture and software stack for pim based on commercial dram technology: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56. IEEE, 2021.
[58] Lenovo. Memcached-pmem. https://github.com/lenovo/memcached-pmem, 2018.
[59] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *ASPLOS*, 2017.
[60] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira 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.
[61] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *ASPLOS*, 2019.
[62] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. Livia: Data-centric computing throughout the memory hierarchy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
[63] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1):94–162, 1992.
[64] David Mulnix. Intel Xeon Processor D product family technical overview. https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview.
[65] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory. *arXiv*, 2020.
[66] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly 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.
[67] Ian Neal, Andrew Quinn, and Baris 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.
[68] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.
[69] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan 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.
[70] Ataberk Olgun, Juan Gómez-Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oguz Ergin, and Onur Mutlu. PiDRAM: A holistic end-to-end FPGA-based framework for processing-in-DRAM. *arXiv*, 2021.
[71] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.
[72] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ISCA*, 2014.
[73] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.
[74] Yujie Ren, Jian Zhang, and Sudarsun Kannan. CompoundFS: Compounding I/O operations in firmware file systems. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2020.
[75] Andy Rudoff. Persistent memory programming without all that cache flushing. https://www.youtube.com/watch?v=uLOjC6Kuokc.
[76] Andy Rudoff, Chet Douglas, and Tiffany Kasanicky. Persistent memory in CXL, 2021.

[77] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[78] Sara Mahdizadeh Shahri, Seyed Armin Vakil Ghahani, and Aasheesh Kolli. (almost) fence-less persist ordering. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[79] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. Large-scale in-memory analytics on Intel® Optane™ DC persistent memory. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, 2020.

[80] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. Nero: A near high-bandwidth memory stencil accelerator for weather prediction modeling. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020.

[81] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. Napel: Near-memory computing application performance prediction via ensemble learning. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019.

[82] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memeory. In *ASPLOS*, 2011.

[83] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. Hardware supported persistent object address translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[84] Tiancong Wang, Sakthikumaran Sambasivam, and James Tuck. Hardware supported permission checks on persistent objects for performance and programmability. In *ISCA*, 2018.

[85] Xiaohao Wang, Yifan Yuan, You Zhou, Chance C. Coats, and Jian Huang. Project Almanac: A time-traveling solid-state drive. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)*, 2019.

[86] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao. Characterizing and modeling non-volatile memory systems. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[87] Michael Wu and Willy Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *ASPLOS*, 1994.

[88] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. PMThreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*,, 2020.

[89] Sam Likun Xi, Aurelia Augusta, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, 2015.

[90] Xilinx. Xilinx Virtex UltraScale+ FPGA VCU118 evaluation kit. https://www.xilinx.com/products/boards-and-kits/vcu118.html.

[91] Jian Xu and Steven 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.

[92] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, and Yan Solihin. Supporting legacy libraries on non-volatile memory: a user-transparent approach. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021.

[93] Jia Zhan, Onur Kayiran, Gabriel H. Loh, Chita R. Das, and Yuan Xie. OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures. In *MICRO*, 2016.

A Artifact Appendix

A.1 Abstract

This artifact document consists of the description of how to reproduce the major results of the NearPM accelerator evaluation results in Section 8. The evaluation is performed on a Xilinx Virtex UltraScale+ VCU118 Evaluation Platform, where the on-board DRAM is emulated as PM, accessible by both the host CPU and NDP units in NearPM.

A.2 Description & Requirements

A.2.1 How to access. The source code of the software and the FPGA implementation are hosted on two GitHub repositories:

- https://github.com/Systems-ShiftLab/NearPMSW
- https://github.com/Systems-ShiftLab/NearPMHW

A.2.2 Hardware dependencies.

- CPU: AMD Ryzen 7 3700X CPU
- Memory: 64 GB DDR4
- FPGA: Xilinx Virtex UltraScale+ VCU118 evaluation platform

The hardware build process uses the Vivado toolchain (version 2018.2 in our experiments). A licensed Vivado version is required to create the bitstreams for the experiments in this document. The toolchain compiles and deploys the bitstreams.

A.2.3 Software dependencies. The software dependencies include both kernel modifications and PM workloads. First, the kernel of the system must be modified and recompiled to enable caching for the emulated PM. The instructions for the required kernel change are provided in `NearPMHW/kernelchange.pdf`. In our setup, the evaluation was done using Linux kernel version 5.4.0. The workloads in the software repository `NearPMSW` have their own dependencies, which are included in our repository.

A.3 Set-up

Clone the NearPMHW repository and create the Vivado project as `NearPMHW/NearPM`:

```
$ git clone https://github.com/Systems-ShiftLab/NearPMHW
$ cd NearPMHW
$ vivado -mode batch -source build.tcl
```

Next, launch Vivado in GUI mode.

```
$ vivado
```

Click on *Open Project* and open the newly created NearPM project. After the project loading is completed you will find out five designs on the *Sources* tab. To build any of the designs: First, right-click on the design and select “Set as Top”. Next, click “Generate Bitstream” in “Flow Navigator” tab and follow the prompt.

This process will take approximately one hour. After the bitstream is generated, program the FPGA using the "Hardware Manager" (bottom left of the Vivado window).

First, make sure that caching of the FPGA memory region is functional. Generate the bitstream for design5 in the project. Use the previously explained steps to generate the bitstream. After programming the device, restart the host and then run the following commands for testing.

```
$ source NearPMHW/setup.sh
$ git clone https://github.com/Systems-ShiftLab/NearPMSW
$ cd NearPMSW/pcache
$ make
$ sudo ./random-chase
```

If caching works properly, the access latency will be around 1–2 ns for smaller block sizes due to caching and keeps increasing as the program proceeds due to of cache misses when accessing larger blocks. You may terminate the test program after analyzing several data sizes.

A.4 Evaluation workflow

A.4.1 Major Claims. The artifact comes with scripts that reproduce the key results in the following figures:

- **Figure 15:** Speedup in code regions for crash consistency in (a) logging, (b) checkpointing, and (c) shadow paging.
- **Figure 16:** End-to-end speedup in (a) logging, (b) checkpointing, and (c) shadow paging.

A.4.2 Experiments. After setting up the environment, reproduce the following experiments.

Experiment (E1): *[Figure 15] [60 human-minutes + 2 compute-hour]: The experiment will reproduce the results in Figure 15.*

For this result to be regenerated the generated bitstream of design5 of the Vivado project must be loaded onto the FPGA. Restart the host and run the following commands.

```
$ cd NearPMSW
$ ./genfig15.sh
```

The output will present the results in a readable format.

Experiment (E2): *[Figure 16] [60 human-minutes + 2 compute-hour]: The experiment will reproduce the results in Figure 16.*

For this result to be regenerated the generated bitstream of design5 of the Vivado project must be first loaded onto the FPGA. Restart the host and run the following commands.

```
  $ source ./NearPMHW/setup.sh
```

Next, run the following commands to get the results for the NearPM MD-sync case.

```
$ cd NearPMSW
$ ./genMDsync.sh
```

Next, take the following steps on Vivado:

1. Open Vivado GUI.
2. Click on *Open Block Design* and select *design_5*.
3. Double click on the block *multi_thread_command_0*.
4. Change *Dimm0 End Addr* to `0xBFFFFFFF`.
5. Follow the steps in Appendix A.3 to generate bitstream and program the FPGA.

After restarting the host, run the following commands.

```
$ source NearPMHW/setup.sh
$ cd NearPMSW
$ ./genfig16.sh
```