Partitioned Persist Ordering

Yasas Seneviratne, Korakit Seeakhupt, Sihang Liu, and Samira Khan
University of Virginia

Abstract

Persistent Memory (PM) technologies enable program recovery to a consistent state in a case of failure. To ensure this crash-consistent behavior, programs need to enforce persist ordering by employing mechanisms, such as logging and checkpointing, which introduce additional data movement. The emerging near-data processing (NDP) architectures can effectively reduce this data movement overhead by partitioning the persistent programs and executing the crash consistency mechanisms in the NDP-enabled PM. However, a significant challenge lies in maintaining the persist ordering when execution has been partitioned between host CPU and NDP-enabled PM. In this work, we first propose Partitioned Persist Ordering (PPO) that ensures a correct persist ordering between CPU and NDP devices, as well as among multiple NDP devices. PPO guarantees high efficiency by reducing unnecessary synchronization among CPU and NDP devices. Based on PPO, we prototype an NDP system, NearPM, on an FPGA platform.

NearPM executes data-intensive operations in crash consistency mechanisms with correct ordering guarantees while the rest of the program runs on the CPU. We evaluate nine PM workloads, where each workload supports three crash consistency mechanisms—logging, checkpointing, and shadow paging. Overall, NearPM achieves $4.3 - 9.8 \times$ speedup in the NDP-offloaded operations and $1.22 - 1.35 \times$ speedup in end-to-end execution.

1. Introduction

Persistent memory (PM) technologies offer both high performance and data persistence. For example, Intel has released Optane PM [35] that shares the DDR interface with DRAMs and the industry is developing other PM devices [11, 71] for the upcoming Compute Express Link (CXL) standard [17]. These PM systems enable applications to perform direct accesses to PM, without going through the file system indirection, unlike conventional storage devices (e.g., HDD and SSD). Thus, PM-optimized applications can benefit from a faster data path. These opportunities have inspired research on developing and deploying PM [5, 7, 9, 13, 14, 21, 29, 46, 54, 56, 74, 87]. However, a new challenge occurs—without the file system maintaining the data persistence, applications need to manage the recovery of persistent data. In case of failures, such as system crashes and power outages, applications that directly access PM need to ensure that the persistent data is maintained under a consistent, recoverable state. This property is usually referred to as the crash consistency guarantee.

There have been a myriad of solutions to provide the crash consistency guarantee for PM-based applications. For example, the undo-logging approach makes a backup of the existing data to PM prior to updates [12, 13, 15, 16, 21, 30, 37, 52, 56]; the checkpointing method periodically makes a snapshot of PM data structures to keep a consistent, recoverable copy [6, 23, 28, 49, 66, 68, 78]; the shadow paging mechanism redirects writes to a shadow memory and swaps backward at commit [34, 63, 64, 84]. However, the crash consistency mechanisms come with a performance cost. First, the crash consistency guarantee requires writes to become persistent in a specific order. For example, the undo-logging mechanism backs up the to-be-updated persistent data before performing the update, introducing additional ordering constraints. Therefore, crash consistency mechanisms introduce additional stalls to the program execution, as they need to maintain a correct persist ordering [45, 52, 62, 67, 78, 87]. Second, these crash consistency mechanisms tend to create and keep extra copies of the original data [13, 15, 18, 34, 37, 45, 63] in order to recover in case of a failure. These data movements introduce additional memory bandwidth utilization. Combining the two performance bottlenecks, crash consistency mechanisms can place extra data-intensive operations on program’s critical path. In our study of ten common PM-based workloads (methodology in Section 6.1), we observe that some crash consistency mechanisms take a high overhead of 29.6% of the execution time on average.

A trend in computer architecture design, near-data processing (NDP), has the potential to mitigate the overhead of data movement in crash consistency mechanisms. By bringing computation closer to data, NDP can mitigate the memory movement between memory and processor (e.g., CPU) [22, 24, 25, 32, 50, 51, 58]. In particular, as the new CXL standard [17] is around the corner, more opportunities for processing closer to PM devices are being opened up. Our FPGA-based NDP prototype achieves $6.97 \times$, $4.26 \times$, and $9.76 \times$ speedup in logging, checkpointing, and shadow-paging over the software baselines. However, when adapting NDP to existing crash consistency mechanisms, it is not trivial to ensure that the NDP follows a correct persist ordering, without breaking the crash consistency guarantees.

We identify that there are two major challenges in maintaining persist ordering. The first challenge arises from running crash consist operations near memory in parallel with CPU execution. When program execution is partitioned between the CPU and the near-PM NDP units, the PM program still needs to guarantee a correct ordering of persists. For example, when the undo-log operation is offloaded to NDP, the memory operations on both CPU and NDP must be coordinated, such that the undo log becomes persistent by NDP before the CPU performance an in-place update. Naively, one can introduce

---

1The implementation of NearPM is open-sourced at [4].
additional CPU-NDP synchronizations to keep this order but synchronization may offset the performance gains from NDP.

Second, the program execution is not only partitioned between CPU and NDP, but may also lie across multiple NDP devices. For example, two interleaved PM devices may both contain NDP units, each holding a fraction of the persistent data. Thus, an offloaded operation may get partitioned on both NDP-enabled PM devices. However, the execution flows on both devices can be out of sync. Back to the undo log example, if one device has the update committed but the other is still backing up data to the log, a failure may cause the recovery procedure to keep the updates in one device but roll back updates on another, leading to inconsistencies after recovery. Likewise, it is not hard to design a naive solution that frequently synchronizes multiple NDP-enabled PM devices at the cost of performance degradation—a design against the key motivation for integrating NDP. In this work, we aim to design a new persist ordering model for such partitioned execution among CPU and multiple NDP devices.

To overcome the challenges in maintaining persist ordering for partitioned execution among CPU and NDP devices, we propose Partitioned Persist Ordering (PPO) that defines the persistence in NDP-enabled PM systems. Next, we describe the key insights of PPO.

**Persist ordering between CPU and NDP device.** Synchronizing after each offloaded operation preserves the original persist ordering, but we observe that maintaining such a strict ordering is often unnecessary. Execution in NDP units can in fact manage their local memory, without being visible to the CPU. For example, NDP-executed undo-log operations can be persisted to NDP-managed memory, as logged data is not used unless during failure-recovery. Therefore, the persist ordering within an NDP unit is not affected by the CPU. Only when data dependency between NDP and CPU exists, NDP needs to be ordered with the CPU. Back to the undo-logging example, in-place updates from the CPU may not persist before the offloaded logging operation that backs up the original data has persisted to the NDP-managed location. Otherwise, the CPU and NDP device do not need to follow a strict ordering. As long as the NDP hardware system can resolve such data dependency, the software execution on the CPU no longer needs to be blocked by the completion of data persistence on the NDP; instead, it can continue the program execution. For example, several logging operations on independent PM addresses can be processed in parallel by the NDP unit. Therefore, in PPO, partitioned execution on NDP that does not persist to CPU-visible memory is no longer constrained by the original persistence on the CPU but only needs to be ordered when there exist data dependencies with the CPU (more details in Section 3.1).

**Persist ordering among multiple NDP devices.** When a PM system consists of multiple PM devices, the key challenge lies in keeping the partitioned execution on NDP devices at the same pace, to present a consistent persistent state to the recovery procedure. Thus, one can naively synchronize multiple devices at recovery-critical points, such as committing a transaction, to make sure writes have been persisted by this synchronization event. However, we find that the synchronization can be delayed, without affecting the failure-recovery. If an NDP operation only persists to its separate PM region, the CPU would not change its persistence state. Even though multiple NDP devices may not stay in sync, as long as they preserve the persistence state that enables recovery, synchronization does not need to happen immediately. In the undo-logging example, if NDP devices delay the synchronization after committing the transaction, the persistent state can still recover: if a failure happens during commit but before synchronization, from each NDP’s execution status, the devices can together restore to a consistent state using the logged data or keep the in-place updates, given that CPU-NDP ordering ensures a consistent copy has been persisted prior to in-place updates; if a failure happens after the delayed synchronization, then the NDP devices can simply keep the committed in-place updates as they have reached a consistent state. In summary, these two key insights preserve a persist ordering and enable the software to provide the crash consistency guarantees (more details in Section 3.2).

On top of PPO, we further prototype an NDP-enabled system on an FPGA platform. We place the FPGA as a PCIe device and ensure data coherence using a software-based approach by modifying the Linux kernel. On top of the FPGA platform, we implement two PM devices, each containing its own NDP units, and evaluate ten PM-optimized workloads, where each workload has implementations for logging, checkpointing, and shadow paging.

The main contributions of this work are the following:

- Near-data processing (NDP) can mitigate data-intensive operations in crash consistency mechanisms for PM system, but introduces new challenges in maintaining a correct and efficient persist ordering. We propose Partitioned Persist Ordering (PPO) that ensures correctness and performance when part of the execution is partitioned among NDP devices.

- On top of PPO, we prototype an NDP-enabled PM system, NearPM, using an FPGA platform. We adapt ten PM workloads to our NearPM system, each workload is implemented in three commonly-used crash consistency mechanisms, logging, checkpointing, and shadow paging.

- Our experiments on ten PM workloads show 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 a CPU-based baseline system. In terms of end-to-end performance, it achieves 1.35×, 1.22×, and 1.33× speedup over the baseline, respectively, in programs based on the three crash consistency mechanisms.
Figure 2: Procedures in crash consistency mechanisms.

2. Background and Motivation

2.1. Crash Consistency and PM Programming

Persistent memory technologies (PM) feature high performance and byte-addressability while providing the additional benefits of data persistence and load-store direct access to persistent data bypassing the file system. The currently commercialized Optane PM [35] from Intel is a memory module that shares the memory bus with DRAM modules; other upcoming PM variations also service as PCIe devices [8, 11, 71] by leveraging new technologies such as the Compute Express Link (CXL) [17].

Compared to conventional storage, PM systems enable programs to perform direct access to persistent data, bypassing the file system indirections. The direct access to persistent data reduces the overhead on the data path but at the same time ships the burden of managing data recovery to the programs. We refer to the ability to restore persistent data in case of a failure (e.g., 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- and redo-logging logging [12, 13, 15, 34, 37, 45, 52, 78, 87], checkpointing [23, 49], and shadow paging [18, 63]. Next, we describe several commonly-used crash consistency mechanisms.

The logging mechanism maintains the persistent data in a separate location (e.g., 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 update; it deletes the log only after completing the updates. Similarly, redo-logging redirects each update to a separate location and only applies the updates in-place only after the updates have become persistent in a separate copy. This way, the redo-logged data can still be used to recover the original place in case of a failure. Different from logging, checkpointing (Figure 2b) maintains a coarse-grained snapshot of the location prior to updates. Another commonly-used coarse-grained method is shadow paging, which redirects the update to a new copy of the PM page and then switches the version of a whole page (Figure 2c).

These mechanisms are necessary for data recoverability but introduce performance overheads. Figure 1a shows the overhead of each of these mechanisms, where logging (undo and/or redo, depending on the implementation), checkpointing, and shadow paging mechanisms take up 25.7%, 23.1%, and 29.6% of the execution time, respectively (detailed methodology in Section 6.1). The main overhead in those mechanisms is due to intensive data movement when maintaining a copy of the data. Figure 1b, 1c, and 1d further break down the crash consistency overheads in each mechanism, where 68.9%, 60.4%, and 70.5% of the overhead come from data movement. Therefore, if we can accelerate this data movement between CPU and PM, there is a huge opportunity for performance improvement.

2.2. Opportunities for Near-data Processing (NDP)

In traditional systems, the CPU is the only processing unit that manipulates data. Even for simple operations, such as creating a copy of data in the 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 this data movement overhead, researchers have introduced the paradigm of near-data processing (NDP) that places computation closer to the data [1, 2, 22, 24, 25, 32, 50, 51, 58, 61, 75, 76, 89]. As underlying operations that maintain crash consistency guarantees exhibit intensive data movement between CPU and PM, NDP has the potential to mitigate the crash consistency overhead. The existing and upcoming PM devices also have the capability to host processing elements. For example, PCIe-based PM devices can easily integrate processing logic [3, 8]; even the more compact PM DIMMs, such as Optane DIMMs, already integrate controllers for data-intensive tasks [82], which has the potential to be extended for NDP. Figure 3b demonstrates an NDP unit inside the PM device can execute data-intensive
2.3. Challenges of Persist Ordering in NDP Systems

Even though NDP has the potential of reducing the crash consistency overhead, maintaining the original crash consistency guarantee becomes a new challenge. The crash consistency guarantee relies on the persist ordering provisioned by the PM system. For example, the undo-logging mechanism in Figure 2a requires the log to be created prior to the update, as indicated by the numbered steps. Likewise, a checkpoint of the old data needs to be persisted before the upcoming updates (Figure 2b); updates to the shadow copy must be persisted before switching the version of the page (Figure 2c).

When offloading computation to the NDP-enabled PM device, program execution becomes partitioned between the CPU and the PM device. Figure 4a shows undo-logging in a conventional CPU-centric system where each step is strictly ordered for crash consistency. Figure 4b offloads the undo-logging operation to an NDP unit while other steps remain executed on the CPU. Even though data-intensive operations in the logging step become faster, such partitioned execution breaks the ordering guarantee—the CPU can compute the updated persistent in PM, while the NDP unit is creating an undo log. In case of a failure, as indicated by the red line in Figure 4b, the program then needs to spawn a recovery procedure. As the update was not committed before failure, the program attempts to read from (as indicated by the rf edge) the undo log for recovery. Due to an incorrect ordering, the undo log may contain the already updated data, leading to an inconsistent recovery.

In addition to issues with CPU-NDP ordering, such partitioned execution can also introduce ordering issues among different NDP devices. Figure 4c shows a scenario where two PM devices interleave. As such, a PM object can span both devices. Likewise, the NDP units (NDP0 and NDP1) on both devices will operate on this partitioned object. However, without ordering the execution on both NDP devices, the offloaded execution may not follow the same pace on both devices. In case of a failure, as indicated by the red line, one PM device (NDP0) has committed the update, but another (NDP1) has not. As a result, during failure-recovery, NDP0 maintains the in-place updates as they were committed prior to failure, while NDP1 attempts to read from (rf) to the old copy in the log. Therefore, after recovery, the PM object can end up with inconsistent data—half from the original version and half from the updated version.

We summarize that the central challenge is to extend the persist ordering from a single execution device, i.e., the CPU, to multiple partitioned devices. Therefore, in this work, we aim to define persist ordering in such partitioned systems and enable the software system to implement efficient crash consistency mechanisms.

3. Partitioned Persist Ordering

In this work, we present Partitioned Persist Ordering (PPO) that ensures persistence ordering across the CPU and near-data processing units in PM. In this section, we define PPO in two practical scenarios—ordering between CPU and the NDP device in PM, and among NDP devices.

3.1. Persistent Ordering between CPU and NDP

The major challenge demonstrated in Figure 4a lies in maintaining the persist ordering between the CPU and the NDP unit. To enable a correct persist ordering, a naive solution is to synchronize between CPU and the NDP unit actively. As Figure 5b illustrates, such a naive solution, the CPU execution needs to wait till the partitioned execution on NDP has completed. Though faster than the CPU-only baseline in Figure 5a, the frequent synchronization offsets the performance benefits.

However, we observe that maintaining such a strict persist ordering is not always necessary. The partitioned execution on NDP does not always share the same memory with the CPU. In the example of Figure 3b, the NDP operation reads from CPU’s memory but copies and persists it to a NDP-managed memory, i.e., an undo log. Therefore, the persist
ordering between the offloaded execution boils down to data dependency between CPU and NDP—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 on 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 NDP-managed memory. Based on this observation, we see the opportunity to overcome the strict ordering between CPU and NDP units to exploit parallelization.

Therefore, we define a more relaxed persist ordering in PPO. First, we define two types of memories:

- **CPU-visible memory** is the memory space that is accessible by the CPU.
- **NDP-managed memory** is the memory space that is only accessible by the NDP unit.

Then, we define basic memory and NDP operations:

- **OP**: an operation, which can either be a memory access or an NDP operation.
- **M**: a memory access to memory address \( x \), where \( w_1 \) and \( w_2 \) are read and write access, respectively.
- **N**: an NDP request, where \( M_{N,x} \) is a memory access within \( N \) that accesses address \( x \).

Lastly, we define the following edges:

- \( OP_1 \xrightarrow{po} OP_2 \): memory operation \( OP_1 \) is program ordered before \( OP_2 \).
- \( OP_1 \preceq_p OP_2 \): memory operation \( OP_2 \) may not persist before \( OP_1 \).

**Persist to CPU memory.** An NDP-issued operation that persists to the CPU-visible memory follows a strict persist ordering, i.e., \( M_{x} \xrightarrow{po} N \rightarrow M_{x} \preceq_p N \) and \( N \xrightarrow{po} M_{x} \rightarrow N \preceq_p M_{x} \). This strict persist order on CPU-visible memory guarantees that, even under partitioned execution at an NDP unit, the original persist ordering still holds.

**Persist to NDP-managed memory.** An NDP operation that persists to NDP-managed memory only needs order according to data dependencies with memory operations from CPU. If \( N \) only persists within its NDP-managed memory, for a read \( M_{N,x} \in N, M_{N,x} \xrightarrow{po} M_{x} \rightarrow M_{N,x} \preceq_p M_{x} \) and \( M_{x} \xrightarrow{po} M_{N,x} \rightarrow M_{x} \preceq_p M_{N,x} \). However, other memory accesses, say \( M_{N,y} \in N \), where \( x \cap y = \emptyset \), do not need to be strictly ordered with \( M_{N,x} \). This relaxed persist ordering on NDP-managed memory guarantees parallelism within NDP units, as the memory persisted in these NDP units are not visible to the CPU, without affecting CPU’s persistency.

![Figure 6: An undo-logging example in multi-device partitioned execution.](image-url)

### 3.2. Persistent Ordering among Multiple NDP Devices

In addition to CPU and NDP ordering, partitioned execution presents another ordering challenge among multiple devices because persistent objects can be interleaved among multiple devices. 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 multiple NDP devices to make sure both 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 completed. Thus, this naive solution avoids the unrecoverable scenario in Section 2.3—the recovery program either both recovers to 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 stalling for synchronization.

Although synchronization seems necessary to guarantee a correct persist ordering, we observe that the NDP-managed memory is not visible to the CPU, unless failure-recovery happens. In the example of Figure 6c, if we relax this persist ordering and let the synchronization among devices delayed, the recovery program can still read from consistent data as long as the NDP-managed data 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 does not lead to additional stalling on CPU...
or NDP devices.

Follow to the definitions in Section 3.1, we additionally define the followings:

- \( N_{a,i} \): the i-th NDP operation on NDP device \( a \).
- \( S \): an synchronization event among all NDP devices.
- \( OP_i \xrightarrow{rf} OP_j \): memory operation \( OP_i \) reads from data persisted by \( OP_j \).
- \( F \): a system failure event; and, an operation \( OP \) that happens before a failure \( F \) is denoted as \( OP \xrightarrow{hb} F \).

**Synchronization among NDP devices.** If an NDP device \( a \) synchronizes with \( b \) through a synchronization event \( S \), then any operation in NDP \( a \) and \( b \), say \( N_{a,i} \) and \( N_{b,j} \), after synchronization may not persist before synchronization has completed, i.e., \( S \xrightarrow{po} N_{a,i} \land S \xrightarrow{po} N_{b,j} \rightarrow S \leq_p N_{a,i} \land S \leq_p N_{b,j} \). On the other hand, before the synchronization, any operation in NDP \( a \) and \( b \), say \( N_{a,i} \) and \( N_{b,j} \), must have been persisted, i.e., \( \exists N_{a,i} \xrightarrow{po} S \land \exists N_{b,j} \xrightarrow{po} S \rightarrow N_{a,i} \leq_p S \land N_{b,j} \leq_p S \). Based on this guarantee, next we discuss the correctness of failure-recovery.

**Failure-recovery correctness guarantee.** When the failure happens before synchronization, i.e., \( S \xrightarrow{hb} F \), the recovery procedure on each NDP device reads from NDP-managed data for recovery. Say, on NDP device \( a \), \( M_{a,\text{recovery}} \xrightarrow{rf} N_a \), where \( N_a \) has logged the original copy. As PPO enforces persistent ordering between updates to NDP-managed data and the in-place update from the CPU (Section 3.1), \( M_{a,\text{recovery}} \) is guaranteed to read consistent data. When failure happens after synchronization, i.e., \( F \xrightarrow{hb} S \), because all prior memory operations have become persistent, the recovery procedure is also guaranteed to read consistent data. In conclusion, the delayed synchronization PPO ensures a consistent failure-recovery and thus guarantees crash consistency.

### 3.3. Implementation of PPO in Near-PM Processing

So far, we have discussed the challenges of maintaining the crash consistency guarantee when execution is partitioned among CPU and multiple interleaved PM devices, and how PPO overcomes these challenges. To evaluate PPO, in this work we prototyped a PPO-enabled system using an FPGA platform. As intensive operations on persistent data are processed near the PM device, we name our prototype NearPM. Next, we look at the design and implementation of our PPO system, NearPM.

### 4. NearPM Design

In this section, we describe the design of NearPM that provides ordering guarantees for NDP in PM programs.

#### 4.1. 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.
- **Dispatcher** decodes and issues requests to **NearPM units** (i.e., execution engines). The **Dispatcher** keeps track of all **NearPM units**. It issues a request as soon as there is an available **NearPM unit**.
- **Address mapping table** converts virtual addresses in the requests to physical addresses, as the parameters in program-issued NearPM requests are in virtual address (see Section 4.6 for details).
- **In-flight memory access table** keeps track of memory addresses being accessed by both NearPM units and the CPU, in order to handle accesses with conflicting addresses. In case a operation attempts to access an address that is being written to, the **Dispatcher** stalls this operation and keeps 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 NearPM devices. When NearPM starts the execution of an operation, NearPM clears the completion status of all NearPM devices. Upon receiving completion
status from other NearPM devices, NearPM updates the completion status accordingly. When all devices have completed execution, NearPM can progress to the next synchronization point.

4.2. 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 1) and then gets decoded by the Dispatcher (step 2). During decoding, the Dispatcher translates request operands from virtual to physical address through an Address Mapping Table (step 3). After translation, the Dispatcher checks the request’s physical address (step 4)—requests without address conflicts are immediately issued, but stall until the completion of the other conflicting request/access (details in Section 4.3). Next, NearPM resets the status bit in Multi-device handler (step 5). Then, NearPM unit receives the request and starts the execution immediately (step 6). Upon completion, NearPM notifies the Multi-device handler to update the status bit both locally and in other NearPM devices (step 7). When all the NearPM devices have completed execution, the Multi-device handler notifies the Dispatcher (step 8) such that new commands can be assigned 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 10). Like before, the Dispatcher also checks the CPU’s addresses for address conflicts before dispatching (step 11). It issues memory access immediately if there is no conflict from the In-flight Access Table (step 12). Otherwise, it stalls CPU’s access until the other access has completed.

4.3. Ordering Guarantee of A Single NearPM Device.

The implementation of NearPM follows PPO. In a single-device setup, it enforces a correct persist ordering between CPU and NearPM. When the PM program on the CPU accesses PM, NearPM ensures no address dependency (i.e., conflicting addresses) between the in-coming read/write from the host processor and any of the pending and/or in-flight offloaded NearPM requests. The dependency detection is handled by the Dispatcher (Section 4.1).

Besides ordering with CPU, multiple NearPM units within a device can execute offloaded requests in parallel. Required by PPO, parallel execution only applies to requests without read-write dependency. Like the CPU-NearPM dependency handling, the Dispatcher also ensures that any NearPM units would not concurrently persist to a common conflicting address but instead persist in the order they were issued from the program.

4.4. Ordering Guarantee of Multiple NearPM Devices

**NearPM request issuing.** When the program issues a NearPM request, each request comes with a per-CPU-thread sequence number to each command to be later used for ordering. Similar to issuing memory requests in interleaved memory devices, the memory controller sends the NearPM request to all interleaved NearPM devices according to their address ranges. Once the request reaches NearPM device, NearPM sets Multi-device handler status and executes them independently.

**Correctness guarantees.** PPO enables delayed synchronization, as the recovery procedure may still read from consistent data to restore the persistent state. Thus, synchronization is not on the critical path. NearPM takes the approach in Figure 10 to coordinate the completion of requests in NearPM upon synchronization. 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. When NearPM starts execution, it waits for both the completion status from other NearPM devices (step 1) and the completion status of local execution (step 2) before progressing to the next synchronization point.

Figure 11 shows the state machine each Multi-device handler uses to track the completion of remote commands when there are two devices mounted on the system. The state machine starts at the All Complete state and will change its state based on the input signals Receive command, Receive local complete or Receive remote complete. When NearPM receives the command and starts the execution, the state ma-
machine change to the *Executing* state. After receiving command complete signals from all devices it will return back to the *All Complete* state. Each NearPM device has a dedicated state machine to track the completion of its command.

### 4.5. Discussion on Corner-cases

Hardware implementations have physical limitations because of only having finite resources (e.g., limited buffer capacity). In this section, we discuss how we overcome several key limitations in NearPM implementation.

**Execution between CPU and NearPM devices.** As PPO buffers and stalls a CPU memory access that has data dependency with in-flight NearPM requests, in case the buffer is full, any incoming requests to the same addresses will be stalled. However, accesses to the remaining addresses are not blocked, without interfering other threads on the CPU.

**Execution among NearPM devices.** As explained in Section 4.4, a NearPM device needs to wait for the completion of others to synchronize before moving to the next synchronization point. However, the completion status of the local NearPM device has no external dependency on the completion status of other devices. Thus, all NearPM devices will eventually complete the local execution without running into a deadlock.

### 4.6. Address Translation

Address translation has always been a challenge in NDP systems [10, 22, 24, 25, 32, 33, 50, 51, 51, 61, 75, 76, 85, 89] as structures such as TLB are placed in the host processor. Fortunately, PM libraries (e.g., [37]) 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 [79, 80, 88]. Therefore, NearPM keeps the translation of the base address for each pool and perform address translation without going through the CPU.

Figure 12 shows the address translation procedure in NearPM. When the program creates a PM pool, NearPM stores both the virtual base address (step ①) and physical base address (step ②) of the pool in the *Address Mapping Table* indexed by the pool ID. During execution, NearPM looks up the pool ID of the incoming request (step ③) and translates its virtual base address to the physical address, and finally adds on the offset to complete the translation (step ④).

**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 replies on the base address of the pool still applies to multi-device scenarios.

### 4.7. Recovery

#### 4.7.1. Persistence domain

PM hardware systems employ extended persistence domains (e.g., ADR [60] and eADR [70]) 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 [82]). 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.

#### 4.7.2. 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 which 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. NearPM Implementation

#### 5.1. Hardware Implementation

We use the Xilinx Virtex UltraScale+ VCU118 evaluation platform [86] to implement NearPM. The development board is attached to a PCIe 3.0 × 8 slot, with a bandwidth of 8...
Table 1: Evaluated crash consistency mechanisms.

| Crash consistency mechanism   | Common operations                                      |
|------------------------------|--------------------------------------------------------|
| Logging (undo) [12, 13, 15, 16, 21, 30, 37, 52, 56] | allocate, generate metadata, generate log, get log ID, allocate, copy data, delete log, commit |
| Logging (redo) [27, 59, 78, 83] | copy data, delete log, commit                          |
| Logging (undo+redo) [15, 37]   | allocate, generate metadata, generate log, get log ID, allocate, copy data, delete log, commit |
| Checkpointing [6, 23, 28, 49, 66, 68, 78] | allocate, generate metadata, copy data                 |
| Shadow paging [34, 63, 64, 84]  | allocate, copy data, switch page                       |

Table 2: System for evaluation.

| System Configuration |                     |
|----------------------|----------------------|
| CPU                  | AMD Zen 2, 2 GHz, 8 cores |
| DRAM                 | 4 × 16 GB DDR4        |
| FPGA                 | Xilinx UltraScale+ VCU118 (Section 5) |
| 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 |

6. Evaluation

6.1. Methodology

System configuration. We evaluate PPO on the prototype of NearPM (implementation in Section 5) in a testbed described with a system configurations in Table 2. In the rest of the evaluation, we focus on a system with the ADR support [77] where a combination of CLWB (or other flush instructions) and SFENCE is needed to ensure data’s persistence. Recently, an extension to the ADR support, eADR, uses batteries to backup CPU’s caches to eliminate the need for cache flush/write-back. In Section 6.2.7, we also evaluate NearPM in an emulated eADR [43] system, where flushes/write-backs are deprecated.

Workloads. Table 3 lists the workloads and their inputs. TPCC and TATP are PM transactions from a prior work [30]; btree, rbtree, skiplist, and hashmap are key-value stores from PMDK [37] library; Redis and Memcached are real-world workloads. PmemKV [39] 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:

- **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 interleaved NearPM devices with delayed synchronization.

6.2. Evaluation Results

In this section, we evaluate applications (listed in Table 3) in the configurations mentioned in Section 6.1.

6.2.1. Micro-benchmark. We first evaluate a micro-benchmark that copies persistent data of variable sizes. Figure 15 shows the speedup from NearPM. As the size of data in-
creases, the speedup also increases: from $1.13 \times$ when the size is 64 B to $5.57 \times$ when copying 16 kB of data. This speedup is comparable to prior FPGA-based NDP prototypes [31, 55, 65].

6.2.2. Speedup in crash consistency operations. Figure 13 shows the speedup from PPO in crash consistency code regions of each workload. On average, PPO achieves $6.9 \times$, $4.3 \times$, and $9.8 \times$ speedup for logging, checkpointing, and shadow paging, respectively. We notice that TATP has a low speedup of $1.23 \times$ 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.

6.2.3. End-to-end speedup. We evaluate the end-to-end performance for all four configurations in Section 6.1. NearPM SD achieves $1.29 \times$, $1.15 \times$, and $1.28 \times$ average speedup for logging, checkpointing, and shadow paging as presented by the first bar in the graphs in Figure 14. This result shows the performance PPO achieved only by effective handling of ordering between the CPU and NDP. NearPM MD SW-sync achieves $1.21 \times$, $1.14 \times$, and $1.23 \times$ average speedup for logging, checkpointing, and shadow paging as presented by the second bar in the graphs in Figure 14 second row. Its speedup is lower compared to NearPM SD, due to the synchronization overhead. By reducing the synchronization overhead, NearPM MD achieves $1.35 \times$, $1.22 \times$, and $1.33 \times$ speedup on average in the three crash consistency mechanisms, as presented by the third bar in Figure 14.

6.2.4. CPU-NearPM parallelism. One benefit of NearPM is the parallel execution between the NDP units and the CPU, i.e., CPU and NearPM can execute at the same time for a certain fraction of the program. Figure 16 presents the average percentage of execution that is parallelizable (the lower stack). On average, logging, checkpointing, and shadow paging have 20.01%, 17.25%, and 24.68% of the execution parallelizable, respectively.

6.2.5. Performance of variable numbers of NearPM units. This experiment evaluates the performance with 1, 2, and 4 NearPM units. Figure 17 shows that the 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.

6.2.6. Multithreaded performance. This experiment evaluates the performance when the application on the host CPU is multithreaded. We take the two real-world workloads, Redis and Memcached, that can scale the number of clients and the backend handlers from 1 to 16 threads. Figure 18 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. 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.

### 6.2.7. Performance in an eADR system.

We also evaluate NearPM under eADR [43], by deprecating cache flush/write-back primitives in the software, following Intel’s approach [44]. Figure 19 compares the average speedup of NearPM under ADR and eADR over the CPU baseline (which always runs ADR). Overall, with an eADR system, the speedup from NearPM reduces only 5.65%, 3.86%, and 3.93% in the three crash consistency mechanisms. Even though eADR reduces the writeback cost, the programs still perform intensive data movements.

### 7. Discussion

**PPO for different crash consistency mechanisms.** We have demonstrated that multiple crash consistency mechanisms can benefit from PPO. These mechanisms all use versioning of persistent data for recovery, which lead to data-intensive operations that are good fit for near-PM processing. However, other mechanisms may still benefit from NDP. For example, checksum-based crash consistency may offload the checksum computation to the PM device and use PPO to order the persistence of its checksum updates.

**Scalability.** Though in Section 6, we evaluated our prototype of two NearPM devices, due to limitations in our FPGA platform, PPO is scalable as synchronizations among devices is off the critical path. We expect future PM systems to have more PM devices, such as memory modules or CXL devices, where having such a scalable persist ordering is critical to performance.

**Expected performance in commercial NDP systems.** Our NearPM prototype shows comparable performance as prior work that also prototype NDP systems [31, 55, 65]—we achieve $7 - 9 \times$ 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 runs at a higher clock speed.

### 8. Related Work

**Near-data processing.** NDP aims to reduce memory movement in the conventional CPU-centric systems [1, 2, 22, 24, 25, 32, 33, 50, 51, 61, 75, 76, 89]. For example, RowClone [72] accelerates bulk data movement in DRAM, Boroumand et al. [10] targets Google workloads with near-data accelerators, and TETRIS [26] accelerates neural networks. Different from our work, they target computation rather than persist ordering. Our proposal PPO, on the other hand, can cooperate with those computation-focused methods to provide persistence guarantees. There have also been works that bring processing to SSDs. For example, CompoundFS [69] accelerates file system IO operations in SSD and Almanac [81] retains SSD history logs using an in-SSD logic. Nonetheless, they target conventional storage systems, different from 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 [67] and followup research continue to optimize the performance of persistency models. For example, DPO [53] and HOPS [62], PMEM-Spec [47], and Themis [73] 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, PPO, extends persistence to NDP-enabled PM systems.

**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 [36]. There are also databases and key-value stores based on PMDK that maintain crash consistency, such as Redis [42], MongoDB [41], RocksDB [38], and Memcached [56]. Atlas [12] and SFR [30] 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, 48]. DudeTM [57] and SoftWrAP [27] use shadow memory to maintain redo logs before applying them to PM. These mechanisms tend to maintain additional copies of data for recovery. Therefore, our NearPM, can be applied to mitigate their crash consistency overhead.

### 9. Conclusions

In this work, we propose Partitioned Persist Ordering (PPO) for PM systems that partitions data-intensive crash consistency operations to NDP-enabled PM devices during execution. On top of PPO, we further prototype an NDP system, NearPM using an FPGA platform, which processes crash consistency operations inside the PM device. We evaluate ten PM workloads, where each workload has three versions that use logging, checkpointing, and shadow paging for crash consistency. Overall, NearPM achieves $4.3 - 9.8 \times$ speedup in the NDP-offloaded operations and $1.22 - 1.35 \times$ speedup in end-to-end
execution.

References

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015.
[2] J. Ahn, S. Yoo, O. Mutlu, and K. 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] M. Ahn, A. Chang, D. Lee, J. Gim, J. Kim, J. Jung, O. Rebbolz, V. Pham, K. Malladi, and Y. S. Ki, “Enabling cxl memory expansion for in-memory database management systems,” in Data Management on New Hardware, 2022, pp. 1–5.
[4] Anonymous, “Source code of this submission,” https://github.com/anonprjct/NearPM.
[5] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, “Bztree: A high-performance latch-free range index for non-volatile memory,” Proceedings of the VLDB Endowment, 2018.
[6] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. 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.
[7] J. 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.
[8] A. Bhardwaj, T. Thornley, V. Pawar, R. Achermann, G. Zellweger, and R. Stutsman, “Cache-coherent accelerators for persistent memory crash consistency,” 2022.
[9] N. 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.
[10] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Knusela, A. Knies, P. 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.
[11] R. A. Brunner, “CXL and the tiered-memory future of servers,” https://www.lenovoxperience.com/newsDetail/283y044hzgcdv7snkrmmx9ovpqfuasemy9h9k7ai26487or, 2021.
[12] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging locks for non-volatile memory consistency,” in Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA), 2014.
[13] A. Chatzistergiou, M. Cintra, and S. D. Viglas, “REWIND: Recovery write-ahead system for in-memory non-volatile data-structures,” PVLDB, 2015.
[14] S. Chen and Q. Jin, “Persistent B+Trees in non-volatile main memory,” in VLDB, 2015.
[15] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, “From ARIES to MARS: Transaction support for next-generation, solid-state drives,” in SOCP, 2013.
[16] 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 ASPLOS, 2011.
[17] Compute Express Link, “CXL 2.0 specification,” https://www.computeexpresslink.org/spec-landing, 2021.
[18] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better I/O through byte-addressable, persistent memory,” in SOSP, 2009.
[19] B. Cooper, “Ycsb,” https://github.com/brianfrankcooper/YCSB, 2019.
[20] X. Dong, N. Muralimanohar, N. Iouppi, R. Kaufmann, and Y. 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] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, “System software for persistent memory,” in EuroSys, 2014.
[22] I. Fernandez, R. Quislant, E. Gutiérrez, O. Plata, C. Giannoula, M. Alser, J. Gómez-Luna, and O. Mutlu, “Natsa: A near-data processing accelerator for time series analysis,” in IEEE 38th International Conference on Computer Design (ICCD), 2020.
[23] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan, “Phoenix: Memory speed hpc i/o with nvm,” in IEEE 23rd International Conference on High Performance Computing (HiPC). IEEE, 2016.
[24] M. Gao, G. Ayers, and C. Kozyrakis, “Practical near-data processing for in-memory analytics frameworks,” in International Conference on Parallel and Distributed Processing (IPDPS), 2015.

[25] M. Gao and C. Kozyrakis, “Hpl: Efficient and flexible reconfigurable logic for near-data processing,” in IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016.

[26] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. 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 (ASPLSOs), 2017.

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

[28] E. Giles, K. Doshi, and P. Varman, “Continuous checkpointing of tmp transactions in nvm,” in ISMM, 2017.

[29] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, “Single machine graph analytics on massive datasets using intel optane dcp persistent memory,” arXiv, 2019.

[30] V. Goge, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, “Persistence for synchronization-free regions,” in PLDI, 2018.

[31] J. Gómez-Luna, I. E. Haji, J. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, “Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture,” arXiv, 2021.

[32] K. Hsieh, E. Brahmia, G. Kim, N. Chatterjee, M. O’Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, “Transparent offloading and mapping (TOM) enabling programmer-transparent near-data processing in GUP systems,” in ACM/IEEE 33rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[33] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, “Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation,” in IEEE 34th International Conference on Computer Design (ICCD), 2016.

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

[35] Intel, “Intel Optane DC persistent memory,” https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2018.

[36] Intel, “The libpmemobj library,” https://www.intel.com/content/www/us/en/architecture-and-technology/parallel-architecture/pmem.html.

[37] Intel, “pmem-rocksdb,” https://github.com/pmem/pmem-rocksdb.

[38] ——, “pmemkv,” https://github.com/pmem/pmemkv.

[39] ——, “pmemkv-bench,” https://github.com/pmem/pmemkv-bench.

[40] ——, “PMSE - Persistent memory storage engine for MongoDB,” https://github.com/pmem/pmem-mongodb.

[41] ——, “Redis,” https://github.com/pmem/redis/tree/3.2-nvm.

[42] ——, “eADIR: New opportunities for persistent memory applications,” https://www.intel.com/content/www/us/en/developer/articles/technical/eadir-new-opportunities-for-persistent-memory-applications.html.

[43] ——, “PMDK change log,” https://docs.pmem.io/persistent-memory/changelog/pmdk.

[44] J. Izraelevitz, T. Kelly, and A. Kolli, “Failure-atomic persistent memory updates via JUSTDO logging,” in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLSOs), 2015.

[45] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, “Basic performance measurements of the Intel Optane DC persistent memory module,” arXiv, 2019.

[46] J. Jeong and C. Jung, “PMEM-Spec: Persistent memory speculation (strict persistence can trump relaxed persistence),” in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLSOs), 2021.

[47] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, “Efficient persistent memory for multicores,” in MICRO, 2015.

[48] S. Kannan, A. Gavrilovska, K. Schwank, and D. Milojicic, “Optimizing checkpoints using NVM as virtual memory,” in IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS), 2013.

[49] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, “Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory,” in ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[50] J. S. Kim, D. S. Cali, H. Xiong, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, “Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies,” BMC genomics, 2018.

[51] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, “High-performance transactions for persistent memories,” in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLSOs), 2016.

[52] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, “Delegated persistent ordering,” in MICRO, 2016.

[53] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, “Recipe: Converting concurrent DRAM indexes to persistent-memory indexes,” in Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), 2019.

[54] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin et al., “Hardware architecture and software stack for pm based on commercial dram technology: Industrial product,” in 2021 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021, pp. 43–56.

[55] Lenovo, “Memcached-pmem,” https://github.com/lenovo/memcached-pmem, 2018.

[56] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, and J. Ren, “DudeTM: Building durable transactions with decoupling for persistent memory,” in ASPLOS, 2017.

[57] E. Lockerman, A. Feldmann, M. Bakhshaliour, A. Stanescu, S. Gupta, D. Sanchez, and N. 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 (ASPLSOs), 2020.

[58] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, “ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging,” TODS, vol. 17, no. 1, pp. 94–162, 1992.

[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] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, “A modern primer on processing in memory,” arXiv, 2020.

[61] S. Nallì, 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 (ASPLSOs), 2016.

[62] S. Ni, J. Zhao, D. Bittman, and E. Miller, “Reducing NVM writes with optimized shadow paging,” in USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2018.

[63] S. 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.

[64] A. Olgun, J. Gómez-Luna, K. Kanellopoulos, B. Salami, H. Hassan, O. Ergin, and O. Mutlu, “PiDRAM: A holistic end-to-end FPGA-based framework for processing-in-DRAM,” arXiv, 2021.

[65] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, “Fast crash recovery in RAMCloud,” in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP), 2011.

[66] S. Pelley, P. M. Chen, and T. F. Wenisch, “Memory persistence,” in ISA, 2014.

[67] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, “ThyNVM: Enabling software-transparent crash consistency in persistent memory systems,” in 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015.

[68] J. Ren, J. Zhang, and S. Kannan, “CompoundFS: Compounding I/O operations in firmware file systems,” in 22nd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2020.

[69] A. Rudoff, “Persistent memory programming without all that cache flushing,” https://www.youtube.com/watch?v=ulOfC6Kuoc.

[70] A. Rudoff, C. Douglas, and T. Kasanicky, “Persistent memory in CXL,” 2021.
[72] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. 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.

[73] S. M. Shahri, S. Armin Vakil Ghalani, and A. Kolli, “(almost) fenceless persist ordering,” in 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020.

[74] A. Shanbhag, N. Tatbul, D. Cohen, and S. 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.

[75] G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gómez-Luna, S. Stuijk, O. Mutlu, and H. 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.

[76] G. Singh, J. Gómez-Luna, G. Mariani, G. F. Oliveira, S. Corda, S. Stuijk, O. Mutlu, and H. Corporaal, “Napel: Near-memory computing application performance prediction via ensemble learning,” in 2019 56th ACM/IEEE Design Automation Conference (DAC), 2019.

[77] Storage Networking Industry Initiative (SNIA), “NVDIMM messaging and FAQ,” https://www.snia.org/sites/default/files/NVDIMM%20Messaging%20and%20FAQ%20Jan%2020143.pdf.

[78] H. Vólos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in ASPLOS, 2011.

[79] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck, “Hardware supported persistent object address translation,” in Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2017.

[80] T. Wang, S. Sambasivam, and J. Tuck, “Hardware supported permission checks on persistent objects for performance and programmability,” in ISCA, 2018.

[81] X. Wang, Y. Yuan, Y. Zhou, C. C. Coats, and J. Huang, “Project Almanac: A time-traveling solid-state drive,” in Proceedings of the fourteenth EuroSys Conference 2019 (EuroSys), 2019.

[82] 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.

[83] M. Wu and W. Zwaenepoel, “eNVy: A non-volatile, main memory storage system,” in ASPLOS, 1994.

[84] Z. Wu, K. Lu, A. Nisbet, W. Zhang, and M. Liu, “PMThreads: Persistent memory threads harnessing versioned shadow copies,” in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2020.

[85] S. L. Xi, A. Augusta, M. Athanassoulis, and S. Idreos, “Beyond the wall: Near-data processing for databases,” in Proceedings of the 11th International Workshop on Data Management on New Hardware, 2015.

[86] Xilinx, “Xilinx Virtex UltraScale+ FPGA VC1118 evaluation kit,” https://www.xilinx.com/products/boards-and-kits/vc1118.html.

[87] J. Xu and S. Swanson, “NOVA: A log-structured file system for hybrid volatile/non-volatile main memories,” in Proceedings of the 44th USENIX Conference on File and Storage Technologies (FAST), 2016.

[88] C. Ye, Y. Xu, X. Shen, X. Liao, H. Jin, and Y. 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.

[89] J. Zhan, O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie, “OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures,” in MICRO, 2016.