CodePM: Parity-based Crash Consistency for Log-Free Persistent Transactional Memory

Guanglei Xu, Yuchong Hu, *Member, IEEE*, Dan Feng, *Fellow, IEEE*, Wenpeng He, Junyuan Huang

***Abstract*—Emerging persistent memory (PM) can provide large persistent capacity with performance comparable to DRAM in modern memory systems. Persistent transactional memory (PTM) needs to ensure data consistency after unexpected power loss or crashes. Therefore, crash consistency strategies such as persistent logging are still required. However, the additional overhead introduced by these strategies, such as significant extra writes on PM, can lead to system performance degradation. In this paper, we propose CodePM, a fault-tolerant PM transactional library that utilizes parity-based crash consistency to remove logging overhead while guaranteeing the correct state of data. CodePM reuses the decoding capability of parity to detect and recover inconsistent objects. To ensure consistency without logs when updating, CodePM employs fine-grained memory fences to carefully align potential inconsistency with the repairability of parity. To detect inconsistency without logs when recovering, CodePM utilizes optimistic speculative scanning recovery by reusing checksum and parity, which supports instant recovery with transient degraded reliability. Moreover, we study the memory fence blocking effects and further augment CodePM with pipelined encoding and persistent writing to hide update latency. We implemented CodePM on Pangolin, the state-of-the-art parity-based PTM for fault-tolerance. Evaluation results with real-world workloads on Intel Optane DCPMM show that CodePM can achieve up to 3.4x higher throughput than Pangolin.**

***Index Terms*—Crash consistency, persistent memory (PM), erasure coding.**

I. Introduction

PERSISTENT memory (PM) is a promising new memory technology to provide memory-like performance and disk-like capacity for persistent storage systems [1]. System applications can directly perform `load/store` instructions to byte-addressable PM. While data on PM can be stored persistently, the data correctness is affected by crash inconsistency. Unexpected crashes may cause data inconsistency due to the volatile cache hierarchy and out-of-order execution.

To ensure crash consistency of PM, existing persistent transactional memory (PTM) [2] commonly uses write-ahead-logging (WAL) during updates on PM. WAL persists logs on PM before in-place updates, ensuring that the system can recover to a consistent state after crashes. But logs significantly increase write operations on PM, which usually suffers from limited write performance [1], leading to system performance degradation. Prior studies [2]–[4] have proposed strategies like decoupled logging to reduce logging overhead. However, these methods still require extra persistent writes for crash consistency or are limited to specific data structures or systems.

Some previous studies [5], [6] utilize system-level reliability redundancy (e.g., parity) to contribute to crash consistency, but they either focus on reducing development efforts or still require additional PM writes during updating. While redundancy can repair data corruption caused by software or hardware errors, it cannot ensure PM crash consistency, so logs are still required [7]. This is because parity redundancy and logs, as two distinct components, are designed to address different aspects of data correctness issues. The disk-based system TRAID [8] observed the functionality redundancy between the underlying RAID system and the upper-layer database logs. It then reduced some of the upper-layer logs by reusing the underlying parity blocks. This design inspired us to observe the overlap between crash consistency logs and fault-tolerant parity redundancy on PM. Our key finding is that parity can play a dual function, facilitating both corruption repair and crash recovery. By decoding parity alongside other consistent data, it is possible to restore inconsistent data to its prior consistent state, an effect analogous to that of undo-logging. Consequently, this approach potentially obviates the need for logging traditionally required for guaranteeing PM data crash consistency, thus significantly improving overall system performance.

However, it is still challenging to provide both crash consistency and fault-tolerance only by parity redundancy: 1) *Multiple inconsistency*. In the transaction update workflow, in-place updates are executed not only for the targeted data column but also for all related parity columns. Consequently, an unexpected system crash might lead to inconsistencies across data and parity columns, rendering the issue beyond the repairability of parity redundancy. 2) *Inconsistency detection without logs*. In addition to logging the content of updated regions, logs also record the locations of these regions to fully detect and recover inconsistency. Previous studies that utilized system redundancy to eliminate logging overhead still require additional persistent writes to record the potential inconsistency locations, such as Kamino-TX [6] using additional intent logs, and Romulus [9] employing additional persistent flags. Therefore, it remains a challenge to detect and repair inconsistencies without introducing extra overhead in updates.

This work is supported in part by the National Natural Science Foundation of China (No. 62272185 and No. 61821003); in part by the Shenzhen Science and Technology Program (JCYJ20220530161006015); and in part by the Key Research and Development Program of Hubei Province (No. 2021BAA189). *(Corresponding author: Dan Feng.)*

Guanglei Xu, Dan Feng, Wenpeng He and Junyuan Huang are with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: grayxu@hust.edu.cn; dfeng@hust.edu.cn; wenpenghe@hust.edu.cn; jyhuang@hust.edu.cn).

Yuchong Hu is with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China, and also with and Shenzhen Research Institute of Huazhong University of Science and Technology, Shenzhen 518057, China (e-mail: yuchonghu@hust.edu.cn).

In this paper, we propose CodePM, a fault-tolerant PM transaction framework, which reuses parity redundancy for crash consistency to remove logs. To address the first challenge (i.e *multiple inconsistency*), we propose fine-grained memory fence instructions to apply stricter write orders, thereby preventing concurrent updates to multiple PM areas in a single transaction. It guarantees that the repairability of parity aligns with the range of potential inconsistency by separating data updates and parity updates. To address the second challenge (i.e. *inconsistency detection without logs*) without extra update overhead, we propose speculative recovery as an optimistic strategy to effectively scan the entire PM space and recover to consistent states. This approach reuses checksums verification and parity redundancy decoding to enable the comprehensive detection and recovery from potential inconsistency. Speculative recovery can also support instant recovery as a background scanning process. This approach allows applications to immediately provide foreground services after crashes, albeit with temporarily reduced reliability guarantees. We also improve scanning throughput through inter-stripe parallelization and software prefetching.

While parity-based consistency of CodePM can effectively reduce PM write traffic by removing logs, it introduces stricter fence ordering in updates, potentially diminishing instruction parallelism and negatively impacting system performance [10]. We first study the blocking effects of memory fences in asynchronous memory I/O protocols and then discover that, even under the same consistency guarantees, different instruction organization strategies can result in varied performance. Driven by findings through experiments, we propose pipeline encoding and persistent writing, which implement early flushing and encoding to alleviate the blocking effects of memory fences and hide update latency. This strategy enables the CPU to execute foreground computations while background in-flight write requests are concurrently committed to PM, enhancing instruction parallelism.

To illustrate the efficiency of CodePM, we built it atop the current state-of-the-art fault-tolerant PTM, Pangolin [7], and evaluate its performance using a series of PMDK benchmarks [11]. Additionally, we executed real-world YCSB [12] workloads on a transactional key-value store implemented by CodePM. Evaluation results reveal that CodePM markedly reduces write traffic to PM media and improves both the latency and throughput of update transactions.

In summary, this paper makes the following contributions:

- We analyze the write traffic issue in existing fault-tolerant PTM and, for the first time, propose parity-based crash consistency, thereby removing log overhead. (Section II)
- To ensure crash consistency without logs, we limit concurrent updates to multiple PM areas by fine-grained fence. Then, we repurpose the system reliability components to implement log-free crash recovery. (Section III)
- We further study memory fence blocking effects and propose pipelined encoding and persistent writing to hide update latency by exploiting instruction parallelism. (Section IV)
- We implement CodePM and evaluate its performance. The results show that CodePM significantly outperforms the state-of-the-art fault-tolerant PTM in several benchmarks. (Section V & VI)

II. Background and Motivations

A. Persistent Memory

Persistent memory (PM) can provide performance comparable to DRAM with large capacity for persistent storage systems [1]. Memory-intensive systems can leverage PM as an alternative to DRAM via asynchronous memory access protocols, including DDR-T and Compute Express Link (CXL) [13]. Furthermore, PM data persistence can diminish system recovery overhead, such as memory caching system warm-up time after crashes. Intel Optane DCPMM, as the first commercially available PM device, has been widely researched and deployed. Although the Intel Optane series has been shut down, industrial research on PM remains active and ongoing, including Samsung memory-sematic SSD [14].

Despite the many advantages that PM brings to memory system design, it also leads to other design challenges due to its unique characteristics [1], [15]. Existing PM devices use multi-layer structures containing SRAM or DRAM as write-back caches to hide the slower media latency. However, numerous studies have demonstrated the low write performance of PM devices, which significantly deteriorates under high pressure [1], [15]. Meanwhile, excessive write requests can lead to heightened competition for resources like memory controllers, thereby delaying all memory I/O requests [16]. Moreover, numerous write requests also diminish the naturally limited write endurance of PM. As a result, many systems [17], [18] focus on reducing PM writes to improve system performance and extend the lifetime of PM.

The persistence of data on PM reduces the recovery cost after unplanned crashes or shutdowns, but poses new challenges to maintain the correctness of data [1], [2]. To ensure crash consistency, the x86 architecture offers a series of persistence-related instructions, such as `CLWB` and `CLFLUSHOPT`, which can flush data in cacheline to PM. Besides, NTSTORE can bypass the cacheline and write data directly into PM through the line fill buffer. Due to CPU volatile cacheline and instruction hardware reordering mechanism, weak-ordered persistence-related instructions may be executed out of order. Therefore, memory fence instructions, like `SFENCE`, are used to ensure the correct order of persist instructions. For example, the write operations that are before memory fences can be globally visible to those after.

Existing PM systems [2], [11] construct mechanisms such as write-ahead-logging (WAL) to ensure the crash consistency during non-atomic writes. For example, undo-logging first copies the old data before updating it to PM. This log is then used to detect and recover potential inconsistency after unplanned system crashes. Copy-on-write (CoW) represents an alternative strategy that addresses the challenge of non-atomic in-place writes by executing modifications on a separate copy. However, the overhead of copying data in CoW presents challenges to its adoption as a complete replacement for the WAL strategy. While some studies [2], [3] propose strategies to reduce log overhead, they are limited to small-range updates, or still require a certain amount of additional PM writes.

Persistent transactional memory (PTM) offers a transactional interface that simplifies integration with upper-layer systems originally designed for traditional DRAM, overseeing all operations on PM to ensure crash consistency [5], [7], [19]. This interface streamlines the transition from DRAM to PM systems, reducing development efforts and ensuring optimal performance through centralized management. Intel PMDK Libpmemobj [11], an open-source C library, manages PM address space — mapped via `mmap()` — into PM pools and maintains persistent metadata at the head of each pool to identify internal persistent objects. It utilizes the WAL strategy for crash consistent updates in upper-layer applications with a transaction interface consisting of C macros.

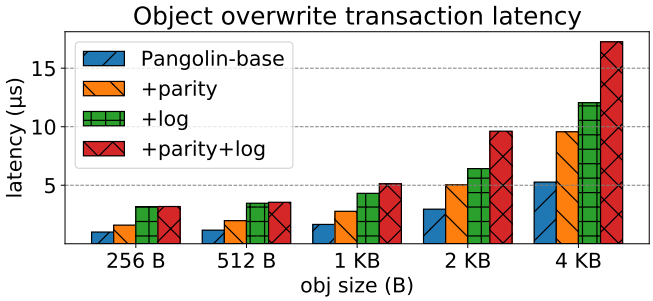


Fig. 1. Logging and parity overhead of object random overwrite transactions in Pangolin.

B. Fault-Tolerant Persistent Memory

Data errors also affect the data correctness on PM [5], [7], [20], [21]. Memory errors in modern production systems have also been extensively studied, as uncorrectable memory errors may lead to serious errors in systems. Recent studies have demonstrated that the reliability of PM devices is lower than that of DRAM [5], [7], [20]. The media of PM may experience random errors, such as bit flips, some of which are beyond the capabilities of ECC modules. Furthermore, errors above the hardware level, such as PM device firmware, kernel errors, CPU errors and software scrubbing, can result in incorrect write requests being sent to PM media. These errors from higher levels cannot be detected by device-level ECC. Although existing commercial products are equipped with on-chip ECC modules, the mean time between failures (MTBF) of PM devices is still only 200 hours [22], which is on par with hard disks. What's worse, the default interleaved configuration mode in PM address space will further amplify the risks of data error, similar to RAID 0 of disks.

For fault-tolerant memory systems, the cost of using replication is relatively high compared to encoded parity [23]. Meanwhile, some studies [24] have demonstrated that employing erasure coding can provide better reliability compared to replication. Reed-Solomon codes [25], the most commonly used erasure coding scheme, are composed of multiple stripes with n columns, including p parity columns and $n - p$ data columns, enabling the recovery of up to p erroneous data columns by decoding any $n - p$ columns. Note that XOR parity represents a specific instance of erasure coding when p is equal to 1. Erasure coded parity redundancy is widely used in distributed systems and local systems [8], [23].

Previous fault-tolerant PTMs [5], [7], [20], including Pangolin, employ system-level parity redundancy on PM to provide system-level fault-tolerance. To detect potential data corruption, Pangolin keeps checksum fields alongside objects for error detection. It then aggregates multiple objects with checksums into groups to form a complete coding stripe, and generates XOR parity to enable the repair of detected corruption. When the application issues update requests, Pangolin allocates a dedicated DRAM buffer as a shadow copy for each updated datum, redirecting application read/write requests to this buffer. Additionally, new checksums and parity are calculated in this DRAM buffer. This design prevents applications from directly modifying PM addresses, thereby preventing crash inconsistency caused by issues such as random cache eviction. Finally, during the transaction commit, since in-place updates on PM is not atomic, all updated fields in the DRAM buffer are persisted back to PM with additional replicated redo logs. The rare instance where crash inconsistency and corruption occur simultaneously at the same offset is not considered [7]. Therefore, inconsistent parity can be reconstructed using data and logs. Pangolin is built on top of Libpmemobj [11], thus applications or benchmarks upon the Libpmemobj C macro transaction interface can be quickly migrated to Pangolin.

C. Motivation

Both parity and logging aim to ensure data correctness. WAL records update operations on PM in advance, facilitating data recovery through log replay following a crash. Parity offers fault tolerance by recovering errors at both hardware and software levels. Previous studies [5], [7], [20] apply both mechanisms separately to address distinct issues of data correctness. Consequently, ensuring crash consistency and fault tolerance during in-place updates requires multiple PM write operations for both data and parity. This increased write activity can drastically affect the system's performance due to PM's limited write bandwidth. Furthermore, heavy PM write loads can lead to excessive contention, degrading the performance of all memory I/O operations [16].

To study the impact of logs and parity on system performance, we conducted a PMDK object overwrite transaction micro-benchmark by removing logs and parity of Pangolin [7]. Figure 1 shows the latency of overwriting varying object sizes with different strategies. Pangolin-base represents neither logs nor parity, while the others are variants with logging or parity protection. It can be observed that both logs and parity increase the average latency of overwrite transactions. Since logs in Pangolin are replicated, adding only logs increases the latency by over 2.5 times. In the case of overwriting 4 KB objects, the combination of parity and logs increases the latency by over 3 times. Extensive writes in Pangolin markedly reduce system performance, particularly with numerous log-related operations.

Some studies have addressed extensive log writes in PTM, proposing system redundancy for crash consistency. Yet, these

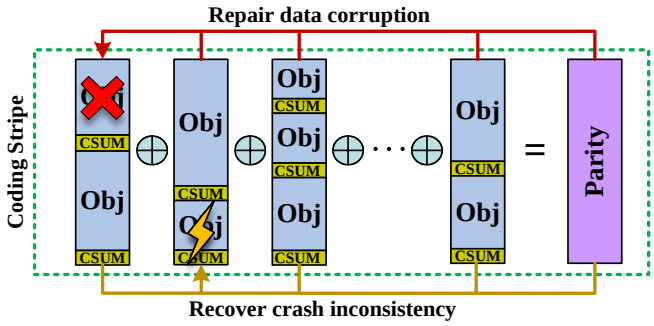


Fig. 2. Parity serves for both corruption repair and crash recovery. For example, one encoded column of parity redundancy can be used to repair the data corruption in the first data column or recover the crash inconsistency in the second data column, as long as they do not occur at the same offset.

approaches [6], [9] still necessitate additional writes on PM during updates to ensure crash consistency. The disk array system TRAID [8] identifies the redundancy overlap between the parity of the underlying RAID system and the logs of upper-layer databases, thereby reducing a part of logs by RAID parity. Driven by this insight, we have found that if crash inconsistency is also considered a type of data error, then the logging functionality within the all-or-nothing guarantees of PTM can be seen as a variant of transient replication. This motivates us to rethink *how to ensure crash consistency only through existing parities*, thus reducing performance degradation from logs. We observed that decoding computation of the parity together with other objects in the same stripe can retrieve an inconsistent object at any position, as shown in Figure 2. In other words, the repairability of parity can recover crash-inconsistent data to the latest consistent state, similar to undo-logging. The protection strategy in this paper, similar to Pangolin's, does not address data errors and crash inconsistencies that occur simultaneously at the same offset. As a result, employing parity can be utilized separately for both types of correctness guarantees simultaneously.

However parity and logs cannot simply substitute each other's functionality due to multiple in-place writes within the update workflow. It is still challenging to guarantee data correctness with only parity redundancy: 1) *Multiple inconsistency*: Multiple in-place updates on PM in existing transactional update workflows can result in potential multiple inconsistencies, which cannot be recovered by parity. 2) *Inconsistency detection without logs*: Implementing inconsistency detection and recovery without introducing extra overhead to the update workflow remains challenging without logs.

III. CodePM Design

We propose CodePM, a log-free fault-tolerant PM transaction framework with parity-based crash consistency. We carefully design the update workflow to remove logs without compromising consistency (Section III-A). We further design optimistic inconsistency detection based on speculative scan recovery without logs (Section III-B). Finally, we discuss the reliability of CodePM compared to log-based strategies (Section III-C).

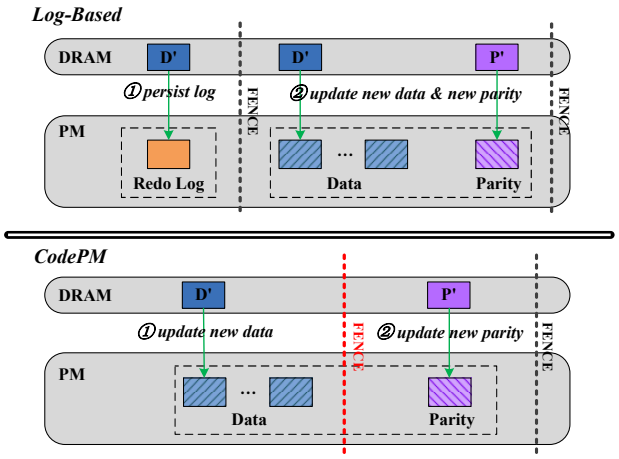


Fig. 3. Log-free update workflow in CodePM. The red extra SFENCE is added to prevent data and parity concurrent updates.

A. Log-Free Consistent Update

Challenge: To ensure crash consistency during updates, the write-ahead-logging strategy introduces additional write overhead. It seems that using parity redundancy as a replacement strategy is both natural and straightforward. However, the update workflows in previous works [5], [7], protected by logging, allowed for in-place writing back to new data and corresponding new parity of any size and any region within a single transaction. Suppose that in the log-free update workflow, both data and parity become inconsistent due to partial updates caused by crashes. Assuming a crash occurs in such a simple log-free update workflow, due to non-atomic writes, partial updates would lead to inconsistencies in both data and parity. In this case, recovering data to a consistent state becomes impossible, as the parity involved in decoding is also not in the correct state. Therefore, the repairability of parity does not align with the existing update workflow in the absence of logs. Thus, simply removing logs undermines the crash consistency during the update process, and affects the fault-tolerance of other objects on the same offset.

Design: To narrow down the range of potential inconsistency in the log-free update workflow, we propose fine-grained memory fences to enforce stricter ordering for both data and parity in-place updates. Figure 3 illustrates a single data block update using a single XOR parity system. In log-based strategies like Pangolin, persisting the log precedes the simultaneous updates to data and parity. This approach, even for small-scale updates, necessitates concurrent modifications across various columns, increasing the risk of potential inconsistencies. Therefore, the sequence of updates determines the range of crash inconsistency. To reduce the number of columns being updated simultaneously, we must impose additional order requirements on concurrent updates. This limitation ensures that potential crash inconsistency is confined within a single column, aligning with the repair capabilities of parity. The log-free update strategy in CodePM, as illustrated in Figure 3, introduces an extra memory fence following data (object and its checksum) updates. This enforces an order that prevents simultaneous updates to both data and parity.

Consequently, any inconsistencies are restricted to a single column. If a transaction commit requires updating multiple PM data areas with the same offset on a stripe, CodePM must insert additional memory fences to ensure that concurrent updates do not occur across multiple data columns, thus preventing multiple inconsistencies. In systems employing multiple parities, the increased repair capability can relax the ordering requirements to some extent. Specifically, in a system with p parities, no more than p data columns with the same offset can be updated simultaneously.

Contrary to intuition, the fine-grained fence design usually does not introduce additional memory fences during the update workflow. Prior studies have shown that excessive use of memory fences can harm system performance [10], while the fine-grained fence design appears to trade more fences for fewer PM writes in the update workflow. Although stricter ordering may increase the need for extra memory fences for data and parity updates, the absence of logging eliminates the need for a fence dedicated to persistent logs. When multiple data updates occur at the same offset, additional memory fences are required to prevent concurrent data updates; otherwise, multiple inconsistencies may arise. But updating the same offset across multiple data areas simultaneously within a single transaction is less likely, especially in PM systems where the address space is significantly larger than in traditional memory systems, further decreasing the probability. Consistent with prior research [5], [7], [26], CodePM ignores the rare case of simultaneous inconsistency and corruption. Hence, even with the extension to multiple parities (e.g., Reed-Solomon codes [25]), broken parities across several columns can be reconstructed by re-encoding the consistent data. CodePM can avoid the use of fine-grained memory fences for concurrent updates of multiple parities. Therefore, compared to the log-based strategy, CodePM's log-free updates exhibit a relatively stable number of memory fences.

Concurrent transaction updates. Pangolin and Libpmemobj do not strictly support concurrent updates. Concurrently modifying a shared object may cause data inconsistency if one transaction crashes. But Pangolin allows multiple threads to execute transactions concurrently as long as they do not modify on the same object simultaneously [7], [11]. On the other hand, parity columns protect multiple data columns, therefore it is possible for parity columns to be modified by multiple transactions simultaneously, which can lead to concurrency safety issues. For efficient thread synchronization, Pangolin proposes *parity range-locks* to ensure concurrency safety by combining two update methods: 1) For small updates, take shared ownership of a range lock and use the atomic XOR instruction provided by modern CPUs to perform atomic parity updates. 2) For large updates, take exclusive ownership of a range lock and use vectorized instructions to perform ordered updates. The design of CodePM can also be extended within this framework to support efficient concurrent updates. However, RS codes for multi-parity are not supported by atomic instructions. To address this, CodePM adopt a strategy that first computes the delta parity for each column using SIMD within the DRAM buffer. Subsequently, CodePM update the existing old parity on PM through 8 B atomic XOR operations with delta parity. For better system efficiency, compared to Pangolin which uses 8 KB as the object size threshold, CodePM selects 256 B as the new threshold by comparing the performance of the two concurrent parity update strategies. Moreover, Since CodePM does not allow concurrent updates on multiple data columns at the same offset, the scope of exclusive locking must extend from parity to data columns. This larger lock can also provide strict support for concurrent data updates. If Pangolin is to support concurrent data updates, a similar strategy needs to be introduced. Although a larger lock scope may result in increased lock contention, the low probability of these events makes their impact on performance acceptable. Our experiments also indicate that the primary factor limiting multi-thread scalability is the write traffic on PM. We plan to improve the performance of locks in our future work.

B. Log-Free Crash Recovery

Challenge: While the log-free approach can reduce the overhead of the update critical path, it introduces new challenges for crash recovery. Write-ahead logging enables recovery after unplanned crashes by scanning all logging entries to identify potential inconsistency and recovering inconsistent objects through replaying or rolling back. In contrast, while the parity-based strategy uses fine-grained memory fences to limit the range of crash inconsistency during updates and matches the repair capabilities of parity, comprehensively detecting potential crash inconsistency without using logs remains challenging. In order to detect the consistency state without explicit logs, previous studies have relied on updating additional flags on PM during updates [6], [9]. This approach contradicts our target of reducing PM writes to improve service performance. The goals of crash recovery in CodePM are the following: 1) It can repair inconsistent data or parity after crashes to ensure the atomicity of transactions. 2) Even in the rare case of unrecoverable errors (simultaneous corruption and inconsistency), CodePM must comprehensively detect them to prevent silent and erroneous recovery.

Design: To avoid extra update overhead, we propose *speculative recovery*, which reuses checksums and parity, two redundancies that are originally used for corruptions, to help detect the consistency state of data and parity during crash recovery. *Speculative recovery* is an optimistic strategy that performs a complete scan on data and parity space after crashes, reusing checksum verification and parity to determine the consistency state of the data and parity. By leveraging existing system components, speculative recovery avoids additional update overhead.

Specifically, the process of speculative recovery consists of two parts. The first part is *checksum reuse*. CodePM first reads a complete stripe on PM into the dram buffer. Since the commit phase concurrently updates both checksums and objects that cross-verify each other on the data column, checksum verification can also detect crash inconsistencies on the data column. However, as shown in Figure 2, the limitation of checksum reuse lies in the fact that the parity protects both the data and its checksums, resulting in a lack of checksums for detecting inconsistencies in the parity

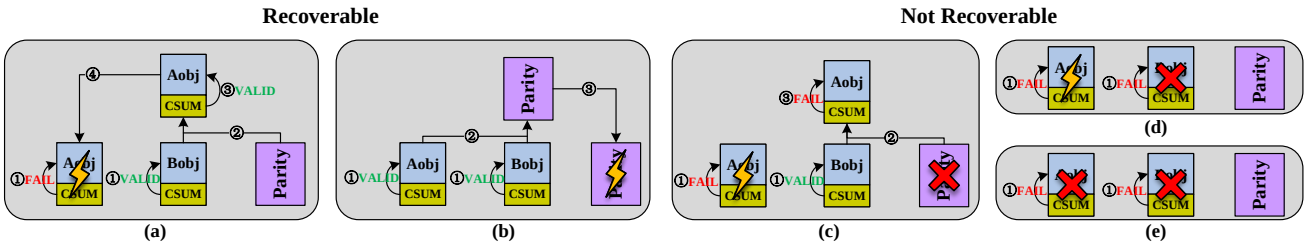


Fig. 4. Log-free crash recovery in different data error scenarios. (a) inconsistency on data, (b) inconsistency on parity, (c) inconsistency and corruption on data and parity respectively at the same offset, (d) & (e) multiple erroneous data at the same offset.

itself. Therefore, speculative recovery needs to be combined with the second part, *parity decoding reuse*. First, if all the data in the DRAM buffer passes the checksum verification, CodePM uses this consistent data to re-encode and obtain new parity. It then compares the new parity with the old parity. If they are the same, it indicates that the parity on PM is consistent. Otherwise, the parity on PM needs to be updated with the newly re-encoded parity. Because the fine-grained memory fence in the commit phase ensures that multiple inconsistencies do not occur, if inconsistent data is detected in the DRAM buffer, parity and other consistent data are used to decode and obtain the new data. If the decoded data fails the checksum verification, it indicates corruption on parity. This suggests simultaneous cross-column errors, which exceeds the repair capabilities of CodePM. CodePM needs to throw a warning to the application. Speculative recovery traverses and scans all the stripes on PM through these two parts to obtain global data state information.

Figure 4 presents the recovery results of CodePM for different data error scenarios during scanning:

Inconsistency on data. Figure 4a illustrates the process where, upon detecting a checksum verification failure on object A (phase 1), new object A is recovered by decoding consistent object B and parity (phase 2). Next, the recovered object A undergoes checksum verification (phase 3). If successful, it is written back to the original location of object A (phase 4).

Inconsistency on parity. Figure 4b shows the case of inconsistent parity. Using the consistent data that passed the checksum verification (phase 1), the new parity is re-encoded (phase 2). Comparing the two parities determines if the stored parity is consistent. If not, the new parity updates the inconsistent parity (phase 3).

Inconsistency and corruption on data and parity respectively at the same offset. Figure 4c presents an example of both erroneous data and parity. First, similar to phase 1 in Figure 4a, an inconsistent object A is detected through checksum verification. But in phase 2, the new object A derived from decoding computation cannot pass the checksum verification (phase 3), indicating that the parity is also inconsistent.

Multiple erroneous data at the same offset. Figure 4d and e illustrate scenarios of multiple erroneous data, such as simultaneous corruptions or both inconsistency and corruption. When errors span multiple columns, checksum verification during phase 1 can detect these as unrecoverable.

Multi-parity speculative recovery. CodePM extends speculative recovery to multi-parity systems by multi-path decoding. Multi-parity speculative recovery differs by offering multiple decoding paths when inconsistencies are detected. Phase 2 in Figure 4(a) will involve multiple paths. For instance, in a system with two parity columns, if an data inconsistency is found, decoding can proceed using either p_1 or p_2, along with other consistent data columns. Due to the uncertain consistency of the parities, each newly decoded data along every path must be verified with the decoded checksum. If verification fails, it indicates that the parity column used in that decoding path is inconsistent and requires updating. For load balance, we employ a simple round-robin approach to select the decoding path. Regardless of the number of parities, CodePM will perform the same re-encoding to determine if these parities match and consistent.

Instant Recovery. Speculative recovery scanning, due to its time-consuming nature, can obstruct and delay online foreground services, undermining the benefits of data persistence. To achieve instant recovery, we propose to deploy speculative recovery as a background task, accepting transient degraded reliability as a trade-off. CodePM distinguishes itself from systems that necessitate full-space scans for maintaining service continuity, such as those involved in recovering volatile indexes [27]. CodePM lacks only the consistency state of the global space compared to PMDK Libpmemobj and Pangolin. However, thanks to persistent checksums and parity redundancy, CodePM can continue services within milliseconds of reading persistent metadata with degraded reliability. Due to the incomplete scanning performed after CodePM's instant recovery, there may be inconsistent data or parity that requires repairing, leading to degraded reliability for data at the same stripe offset. In the degraded state, CodePM necessitates additional checksum verification for each read to ensure data correctness and determine if repairs are needed. This degraded state can only be recovered after speculative recovery through a complete scan. The speculative recovery process can amortize overhead by running in the background.

C. Reliability Analysis

CodePM offers fault-tolerance comparable to the state-of-the-art framework Pangolin, and ensures crash consistency for transactions. Although after instant recovery, the system is under a reliability-degraded state, it does not affect the ultimate reliability. The issue not addressed in Section III-B is the simultaneous inconsistency and corruption at the same offset. First, Pangolin cannot handle such a situation. For instance, if both the updated data and parity are inconsistent while

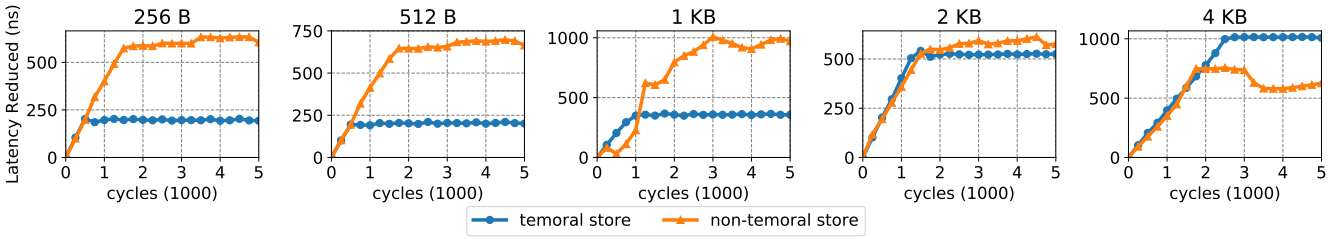


Fig. 5. The reduced latency achieved by moving the memory fence to after tasks that consume a specific number of CPU cycles.

other data at the same offset are corrupted, recovery becomes infeasible. Even though logging can recover the updated data, the corrupted data at the same offset cannot be recovered by the inconsistent parity. Similarly, although CodedPM limits multiple inconsistencies, it still fails if both corruption and inconsistencies occur simultaneously at the same offset and the number of their columns exceeds p. This scenario surpasses the recovery capability based on decoding.

However, previous research argued that in an large PM memory address space, the probability of random corruption and crash inconsistency occurring at the same offset is extremely small. As a result, they overlooks the possibility of simultaneous crash inconsistency and corruption [5], [7], [20]. This is because data corruption primarily stems from uniform small-probability errors (10^{-9} to 10^{-4}) [5], [28]. On the other hand, crash inconsistency arises from interruptions during non-atomic updates. The potential inconsistency is limited to the updated range at the moment of the crash, which is a very small fraction relative to the entire space.

IV. Pipelined CodePM

To delve deeper into the implications of memory fences in the log-free updates with the asynchronous memory I/O protocol, we designed experiments to observe how memory fence placement affects access to PM (Section IV-A). Driven by the findings from the experiments, we propose pipelined encoding and persistent writing by carefully adjusting the order of instructions to hide update latency (Section IV-B).

A. Memory Fence Blocking Effects

Although the log-free update significantly diminishes write traffic in the update critical path, it still has limitations stemming from memory fence effects. Fine-grained memory fences also diminish the parallelism between data and parity updates. The Intel DDR-T asynchronous memory protocol offers a set of asynchronous instructions to ensure the persistence of data on PM, such as `CLWB` and `CLFLUSHOPT`, which can asynchronously flush data from the CPU cache to PM media [1]. We refer to the data that has been flushed but not yet arrived at PM media as *in-flight writes*. Similar to `fsync()` in file systems, `SFENCE` acts as a memory fence to block subsequent memory writes until all previous in-flight writes are completed. As a result, strategically scheduling instructions while ensuring data persistence emerges as a critical determinant of PM system performance. For instance,

Algorithm 1 Measure memory fence effects

Require: $addr$, len, $cycles$

```
pos = 0
while pos < len do
    mov 0x00, [addr + pos]; // Temporal store
    clwb [addr + pos];
    // Line 3 and 4 can be replaced by nt-store (movnti)
    # Insert memory fence here
    busy_waiting(cycles)
    # Or insert memory fence here
    pos = pos + 64;
end while
```

recent research highlights that the frequency of `SFENCE` operations profoundly influences PM system concurrency [10]. Moreover, the lazy and batch flushing techniques have been proposed to diminish the average flush latency [29]. Although the log-free strategy in CodePM reduces PM write traffic, it does not decrease the number of memory barriers. Instead, it limits the concurrency of data and parity updates. It is worth mentioning that asynchronous memory I/O is still necessary for PM, as the memory access latency of PM is still higher than that of DRAM, and synchronous I/O protocols would result in more CPU cycles being wasted waiting for slow memory I/O completion. Therefore, not limited to the DDR-T protocol, other slow memory platforms like CXL also provide similar operations to guarantee consistency, such as global persistent flush (GPF).

Since the log-free strategy in CodePM relies on fine-grained memory fences to ensure crash consistency, simply reducing the number of fences is not feasible. To further study the impact of memory fences and flush instructions, we first design a micro-benchmark to understand how memory fences affect system performance. Algorithm 1 shows how we quantify the latency difference with respect to fence positions. This benchmark continuously performs persistent write operations to a random aligned address on PM, with each persistent write consisting of either an non-temporal store (e.g., `MOVNTI`) or a temporal store (e.g., `MOV`) followed by a flush (e.g., `CLWB`). After each persistent write is completed, we use `RDTSC` instruction to idle in a for-loop for a specified number of CPU cycles to simulate time-consuming tasks, before initiating the next persistent write. We observe the impact on the average latency by inserting the memory fences (e.g., `MFENCE`) associated with each write before or after the for-loop.

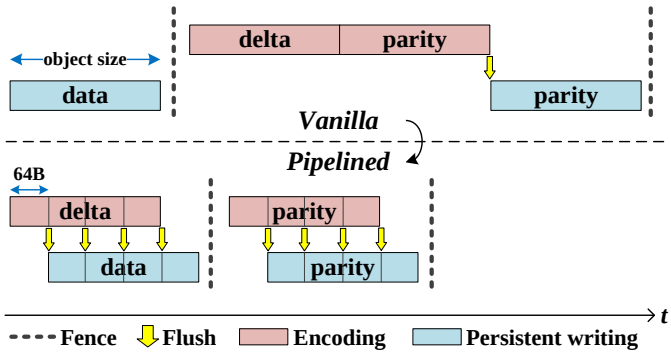


Fig. 6. Pipelined encoding and persistent writing. The red box represents foreground encoding, whereas the blue box represents background persistent writing.

Figure 5 shows the average results of 1 million executions for each, demonstrating that moving memory fences after the for-loop noticeably reduces the average latency. The extent of latency reduction is related to the object size, the waiting time within the for-loop, and the number of channels. For non-temporal stores, moving the memory fence to a later position linearly reduces latency until time-consuming tasks completely mask the latency of persistent writes, achieving peak efficiency. For temporal writes, latency reduction is more noticeable with smaller access sizes, yet fluctuations increase. This occurs because temporal writes necessitate pre-reading data into the cacheline, which is a slower process and is also influenced by the random cache policy. The insight behind the experiment results is that after executing asynchronous persistence instructions, data is written to PM in the background and the CPU can execute other instructions in the foreground. This benchmark reveals that the impact of memory fences on system performance is not only related to the number, but also depends on the number of in-flight writes that have to wait due to memory fences. Despite executing same tasks and ensuring the same level of crash consistency, the arrangement of persistence-related instructions still markedly influences instruction execution concurrency, thereby affecting system performance. This effect is particularly significant in fault-tolerant systems with parity redundancy, as the additional compute-intensive tasks heighten CPU resource demands, amplifying the role of instruction execution concurrency on overall system performance.

B. Pipelined Encoding and Persistent Writing

Challenge: In Pangolin and Pavise [5], [7], new data is initially buffered and encoded in DRAM to generate the corresponding parity. Subsequently, both the new data and parity are persisted on PM. However, *encoding before flushing* in this manner results in the memory fence for the persistent write operation having to wait for all just-issued asynchronous persistent write instructions for data and parity to complete. As a result, a notable blocking overhead arises and may further increase with the size of the updated object. This increased number of in-flight writes leads to wasted CPU cycles waiting. This finding prompts us to further improve the system's overall performance by reorganizing the default positions of flush and memory fence instructions in log-free updates.

Design: To hide update latency, we propose *pipelined encoding and persistent writing* strategy to minimize the number of in-flight write requests that need to be wait due to memory fences. Simply delaying memory fences is insufficient to effectively extend the flight window of PM write requests, resulting in limited optimization potential. Therefore, we approach our pipeline strategy from two perspectives. First, our approach seeks to prioritize independent computational tasks, and delay the memory fences until post-completion, thereby postponing the window's end point. Second, the data no longer required for subsequent computation should be flushed as early as possible, thereby moving the window's start point forward.

Figure 6 illustrates the differences between the vanilla encoding before flushing and the pipelined approach. To prioritize independent computational tasks, we first collect and apply updates to data columns that do not need computation. This is followed by the calculation of delta data and delta parity in the DRAM buffer. A memory fence is then used to ensure the persistence of data writes, and the delta parity then facilitates persistent updates to the parity columns with another memory fence. Through this design, we shift a part of the computational operations, previously executed post-memory fence, to the window preceding the fence. To initiate early data flushing, we first observe that the object size and flushing granularity (64 B) are typically distinct [7], [30], which offers an opportunity to pipeline the update process by early cacheline flushing. The flush granularity is cacheline size (64 B), and there is no interdependency between different 64 B units within an object. Consequently, after the encoding of a preceding 64 B unit is completed, it can be immediately flushed to PM, without the need to wait for the entire object's encoding to be completed.

The idea of pipelining is prevalent, such as CPU instruction pipelines and network communications. Nevertheless, previous studies, such as batch flushing [29], have overlooked the potential of leveraging asynchronous memory I/O to construct pipelines. We leverage early flushing and encoding to extend the time interval between issuing a write request to PM media and the subsequent blocking by a memory fence, reducing the queue of pending write requests ahead of the memory fence. The parity-based PTM introduces additional computational encoding tasks compared to traditional I/O-intensive systems. Hence, the pipeline optimization strategy manifests its effectiveness by allowing CPU encoding tasks to proceed in the foreground while simultaneously performing background flushes of pending writes to PM. In summary, the pipeline strategy efficiently leverages asynchronous memory I/O protocols to boost the instruction-execution parallelism.

V. Implementation

Pangolin architecture. Pangolin [7] is a transactional programming library designed for PM to provide fault-tolerance. It is built upon Libpmemobj transactional library of PMDK [7] by introducing object-granularity checksums and RAID-4 style XOR parity to prevent data corruption. Since Pangolin is

derived from PMDK Libpmemobj [11], it supports a series of PMDK benchmarks. Pangolin creates multiple 16 GB zones within a specified PM pool on PMDK, with each zone divided into several columns. Metadata is persistent at the head of the related structures. The computation-based determination of parity addresses, using the logical memory address relationship between parity and data, eliminates the need for additional persistent metadata in Pangolin. During the transaction, Pangolin creates a dedicated DRAM micro-buffer for each modified data region to facilitate subsequent operations. The micro-buffer is reused within a buffer pool, minimizing the overhead associated with frequent allocations and deallocations. Throughout the transaction process, applications can only obtain pointers to the DRAM buffer, and are not allowed direct access to PM addresses. These write requests intended for PM are accumulated and then collectively committed at the end of the transaction. During the commit phase, Pangolin ensures crash consistency for subsequent PM updates by first writing replicated redo-logging. These logs serve the purpose of recording the changes made during the transaction. The purpose of these logs is to document data modifications within the transaction, without logging parity changes. Subsequently, new checksums and new parity are calculated and written back to PM with new data. When a new object is written, it is appended to the initialized zero-value tail region, thereby obviating the need for logging. Logging becomes necessary only during object updates.

Modifications of CodePM. To demonstrate the effectiveness of CodePM, we extended Pangolin to implement log-free and pipelined variants. We mainly modified Pangolin commit process after the ends of transactions. The original Pangolin commit process computes new parity and new checksums on the DRAM buffer, and then flushes modified areas back to PM address with replicated redo-logging. As mentioned in Section III and IV, our modifications in Pangolin include: 1) eliminating PM writes for creating and cleaning logs in updates, 2) inserting extra memory fences between persistent writes of data and parity, 3) integrating the encoding and persistent writing processes that were originally separate through the pipeline strategy. CodePM keeps the same the policies for checksum and parity computation and verification in Pangolin. CodePM incurs no extra overhead for persistent metadata management. We have extended both systems to include support for multiple parities using RS codes.

Pangolin offers a set of transaction APIs for applications to invoke. `pgl_open()` and `pgl_tx_add_range()` are used to create and designate DRAM micro-buffers corresponding to updated objects, while `pgl_commit()` serve as the transaction commit point for redo-logging and the update of checksums, data and parity. Our modifications focus on the commit process to reduce log overhead. We remove logs for object update transactions and eliminate log reclamation overhead after transaction completion. To ensure crash consistency, we separate the updates for new data and new parity, and use additional memory fences to control their sequencing. To improve execution efficiency, we extended the Intel ISA-L library [31], a SIMD-optimized low-level assembly library, to insert persistence-related instructions during encoding. In scenarios requiring extra buffers, the existing object buffer pool is re-utilized to reduce the memory allocation and deallocation overhead.

Speculative recovery performance. The optimistic speculative recovery prevents extra overhead of updates but incurs higher loads for recovery. While maintaining consistency-related metadata on PM can alleviate the load during recovery [6], CodePM emphasizes minimizing additional update overhead to maximize update performance. We observed that during speculative recovery, most of the time is spent on reading PM and re-encoding. Only inconsistent data or parity needs to be overwritten. Previous studies have showed that both the read performance of PM and CPU computation resource are relatively sufficient and scalable [1]. While CodePM requires a lot of extra computation compared to existing full-space scanning recovery [27], since there is no dependency between stripes during recovery, we can distribute the recovery task across different cores to achieve better scalability. Considering the slower read speed of PM relative to encoding throughput, and the fact that the targets of upcoming scans are predetermined, recovery throughput can be further enhanced through software prefetching. Ultimately, CodePM, with inter-stripe parallelization and software prefetching, can achieve speeds comparable to systems relying solely on scanning reads for crash recovery.

VI. Evaluation

A. System Setup

Hardware configurations. Our hardware testbed is a Linux 4.15.0 server with Intel Xeon Gold 6240 @ 2.60 GHz with 18 physical cores. Each CPU socket has 6 channels of 192 GB DRAM and 128 GB Intel Optane DCPMM with AppDirect mode. Except where noted, we conducted all experiments across all 6 memory channels. Guided by prior studies [1], [15], we limited our evaluations to a single CPU socket to avoid NUMA access penalties. To ensure stable hardware performance across multiple experiments, we also disabled the CPU dynamic boost frequency and C-state.

System configurations. Pangolin [7] is developed based on Intel PMDK Libpmemobj [11]. We implement two variants of CodePM with proposed optimizations to study the impact of different strategies. CodePM removes logs from Pangolin with fine-grained memory fence, while CodePM-P deploys the pipelined strategy to exploit instruction parallelism. CodePM and Pangolin maintain alignment on transaction atomicity, data correctness, and vectorized acceleration instructions. Note that the baseline, Pangolin, across all benchmarks, refers to Pangolin-MLPC with Metadata replication, Log replication, Parity, and Checksum. We additionally removed all logs in Pangolin as another baseline (Pangolin-nolog). Pangolin-nolog only uses one memory fence after a complete transaction has ended, so it cannot guarantee the crash consistency of data and parity. By default, a stripe in the encoding scheme consists of 10 columns, including 1 parity column. Read verification is disabled. All benchmarks are compiled with `O3` optimization. We used IPMCTL [32] to record read/write traffic of PM.

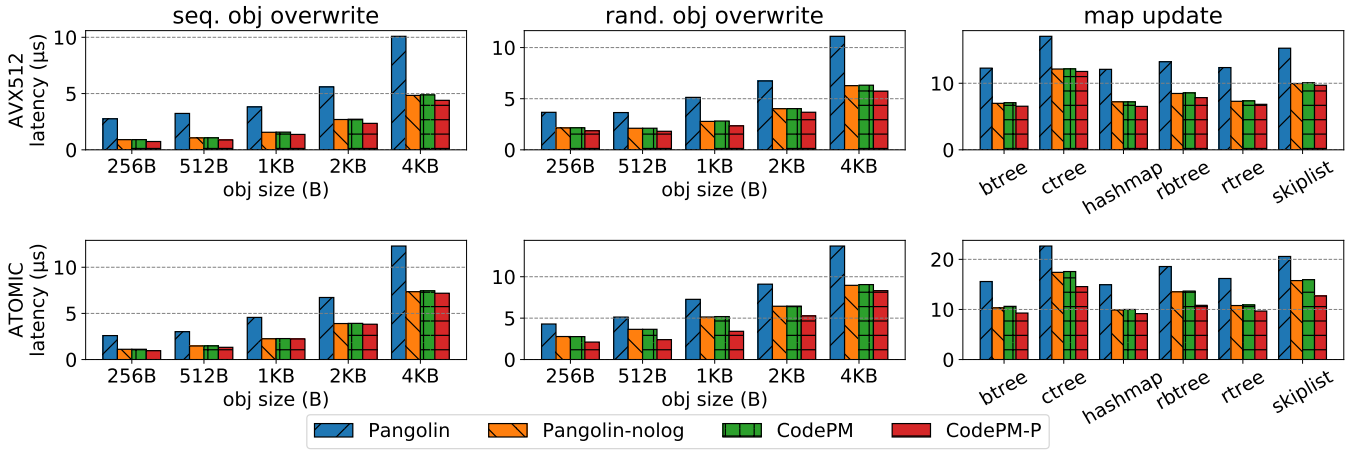


Fig. 7. PMDK transaction benchmark performance.

B. Micro-Benchmark Results and Analysis

1) Object overwrite transactions. Figure 7 measures the PMDK transaction performance. In the object overwrite transactions of the PMDK benchmark, 1 million objects of a specified size are created on PM. Subsequently, these objects are then modified using the library's transaction interface, with transactions being committed at the end. In Pangolin, two encoding strategies are provided for transaction concurrency: SIMD vectorized instructions with locks and atomic XOR instructions. In AVX512 SIMD coding, it is observed that CodePM offers a substantial latency reduction by 55.8–67% across various object sizes, similar to Pangolin-nolog. This improvement stems from the significant log overhead in Pangolin, which includes writing two replicated logs and cleaning them to zero. Relative to Pangolin-nolog, CodePM incurs a slight increase in latency, attributable to diminished parallelism in handling data and parity updates. CodePM-P decreases latency by 9.9–18.4% compared to CodePM by exploiting instruction parallelism. This performance improvement is especially significant for small object sizes, where the optimization benefits are accentuated due to a higher frequency of memory fences and diminished parallelism associated with such objects. In object random overwrite transactions, the logging overhead of sequential writing is minor when compared to the cost associated with the random writes of data and parity. Consequently, with smaller object sizes, the performance gains realized through CodePM are less pronounced, leading to a diminished latency reduction ranging from 41.2–43.1%. Moreover, CodePM-P, in comparison to CodePM, can further decrease latency by an additional 9.2–13.5%.

On the other hand, in atomic coding, CodePM's latency reduction ratio is 41.0–57.0% in sequential object overwrites, and 35.5–36.3% in object random overwrites. Compared to CodePM, CodePM-P demonstrates a latency reduction ranging from 14.2% to 23.8% for smaller data sizes (256 B). This improvement surpasses that of AVX512 coding due to the low efficiency of atomic coding, which enhances the latency hiding effect in the pipeline strategy of CodePM-P. This indicates that the effect of the pipeline strategy is highly correlated with the computation latency. Similar to AVX512 coding, the optimization of CodePM-P deteriorates with large object sizes, especially during sequential writes. In Pangolin, a threshold of 8KB was set for switching from SIMD to atomic mode, based on the performance comparison between Libpmemobj replication and Pangolin's atomic mode. Our experiments showed that with AVX512 vectorization, AVX512 mode surpassed atomic mode for objects larger than 256 bytes. Thus, a 256B threshold for switching parity computation strategies was established for better performance in the following experiments.

2) Map update transactions. We insert and randomly update 1 million 256 B objects into various data structures with PMDK benchmarks, including btree, ctree (crit-bit tree), hashmap, rbtree (red-black tree), rtree (radix tree), and skiplist. CodePM mitigates average latency across them by 28.7–42.4%. Furthermore, CodePM-P offers additional optimizations in average latency by 3.7–9.5%. Compared with object overwrites, these data structures involve more complex internal operations, such as indexing before updating. A single transaction may trigger multiple persistent writes, such as updating counters in hashmaps. Consequently, separating data and parity updating in log-free updates may adversely affect cache efficiency, leading to less significant enhancements.

3) Performance under different system configurations. Figure 8 illustrates the object random overwrite transactions performance under different encoding parameters and numbers of PM channels. First, similar to previous research findings [5], we observe that the width of stripes does not significantly affect the performance. This can be attributed to the fact that the stripe width does not alter the number of PM I/O operations or computation tasks. Second, more memory channels directly enhances the PM write bandwidth, facilitating the distribution of large IO requests across multiple channels and improving average latency. Nonetheless, CodePM's optimization primarily targets reducing writes, leading to diminishing its benefit with multiple channels. But since the utilization of multiple channels for most I/O operations is limited, CodePM maintains high efficiency. The benchmark results of 2 parities indicate that the latency reduced by CodePM drops to 41.4–60.0%. This is due to the decreased proportion of log writes in the

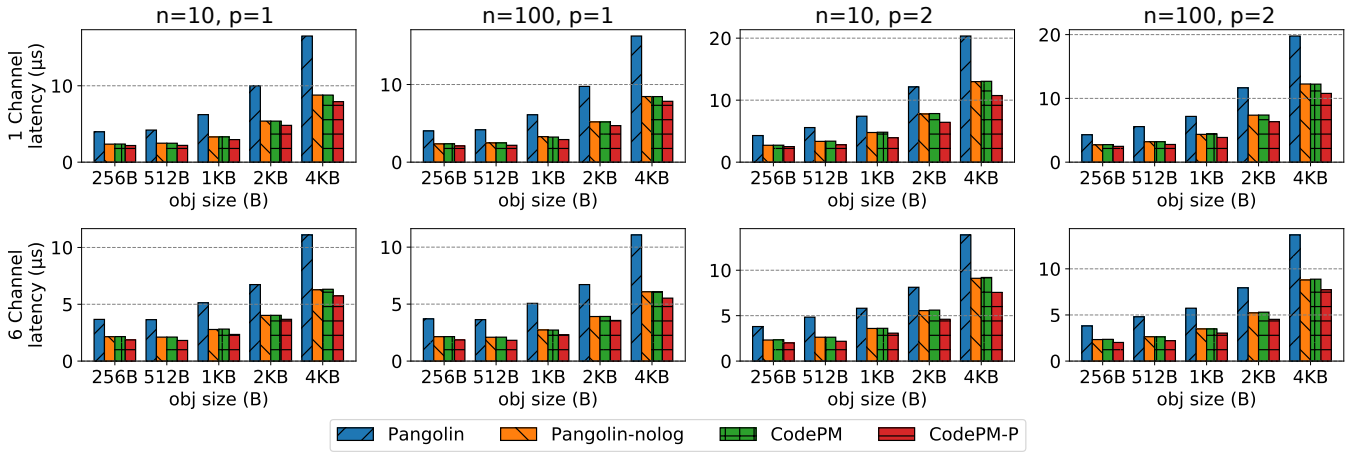


Fig. 8. PMDK object random overwrite transaction benchmark performance under different system configurations.

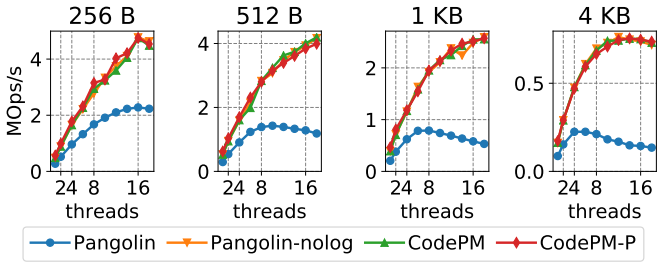


Fig. 9. Object random overwrite transaction scalability.

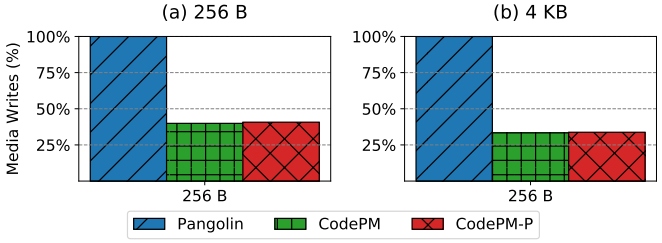


Fig. 10. The PM write traffic of object overwrite transactions.

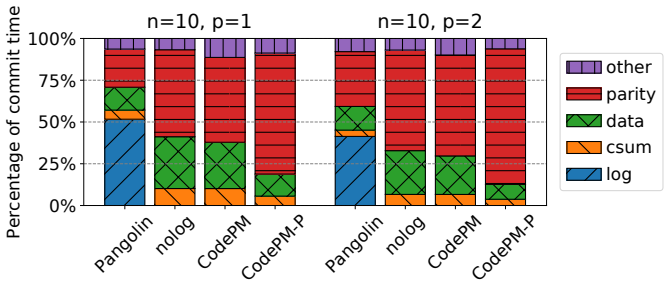


Fig. 11. Overhead breakdown of object overwrite transaction commits.

total write operations during the update process Meanwhile, more parity columns elevates both the computational tasks and PM writes. This increases the optimization space of the pipeline strategy, allowing more tasks to be efficiently hidden. As a result, in comparison to CodePM, the latency reduction realized by CodePM-P is enhanced to 14.5–21.2%. In summary, our proposed strategies significantly optimize performance across diverse system configurations.

4) Scalability. Figure 9 presents the object overwrite transaction scalability in SIMD coding with varying object sizes and the number of threads. Comparing peak throughput, CodePM achieves a 208–335% performance improvement over Pangolin by eliminating logging. This significant enhancement is primarily due to alleviating the PM write bandwidth bottleneck, a critical factor that is exacerbated by higher levels of concurrency. The removal of logging reduces write traffic, which in turn, substantially improves throughput performance. Moreover, due to the limitations imposed by write bandwidth, Pangolin's throughput performance quickly reaches the saturation point and subsequently declines. At this juncture, CodePM is able to deliver a performance that is up to 541% of Pangolin's throughput. The rapid degradation of log-based systems is further exacerbated by the load imbalance across multiple memory channels. This results in a performance bottleneck during log writing, especially under high concurrency conditions. At lower write loads, CodePM-P exhibits marginally superior performance compared to CodePM. However, their peak throughput remains comparable, which can be ascribed to their analogous write traffic.

C. Write Traffic Analysis

Figure 10 shows the normalized write traffic observed on the PM media for each aligned object overwrite transaction request. CodePM generates about 43% of write traffic compared to Pangolin at 256 B and about 33% at 4 KB. Pangolin maintains replicated redo-logging, which leads to 2 writes for logs, 2 writes for data and parity, and 2 writes for cleaning logs. In contrast, CodePM only needs to update data and parity, saving 2/3 of write traffic, consistent with the results at 4 KB. On the other hand, the ratio of write traffic reduced by CodePM at 256 B decreases. The reason is that log cleanup writes are spatial locality friendly under low write loads. Therefore, a part of writes only touches the on-chip write buffer of PM, rather than directly accessing the PM media. In

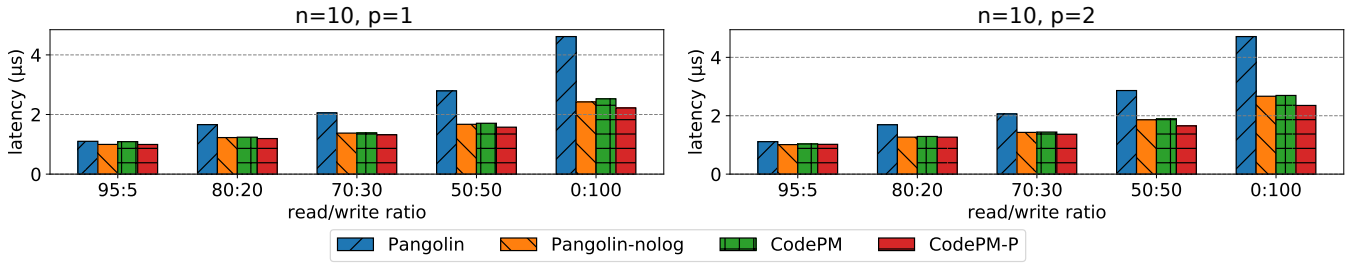


Fig. 12. YCSB benchmark performance on a transactional hashmap.

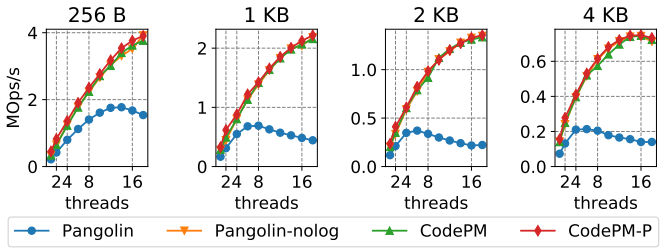


Fig. 13. YCSB update transaction scalability.

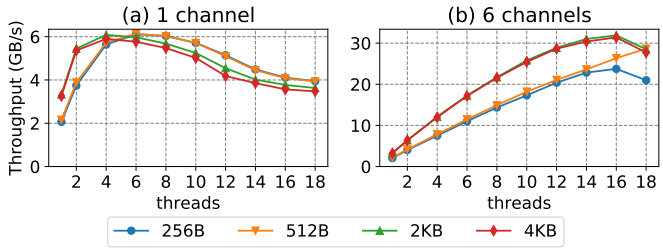


Fig. 14. The speculative recovery scanning throughput.

addition, CodePM-P has slightly higher write traffic compared to CodePM. This may be attributed to the pipeline strategy causing more dispersed write requests. A small portion of 64 B writes cannot be merged into the 256 B granularity of the DCPMM XPLine, resulting in slight write amplification [15].

D. Overhead Breakdown Analysis

Figure 11 shows the overhead breakdown from different operations during the commit phase of random 4 KB object overwrite transactions. We utilized the `perf` tool with the frame pointer register to observe the overhead introduced by different system components. In Pangolin, the processes of log writing and cleaning significantly contribute to the duration of the transaction commit phase. As a result, CodePM substantially improves the overall system efficiency by eliminating logging. Without logs, the parity overhead in CodePM, attributed to parity reading, computation, and updating, constitutes approximately half of the total overhead. Updating parity is slower than updating data because the data update operation itself does not require reading from PM, whereas updating parity first necessitates reading from the PM. However, in the pipeline data version of CodePM, the data overhead is markedly reduced to approximately 10%. The reason for this reduction is that the data update's fence blocking effect is delayed to include the parity updates, thereby aligning the overhead with only the foreground duration of asynchronous data flushes, rather than the entire duration of writing to PM.

E. Macro-Benchmark Results and Analysis

To further evaluate with real-world workloads, we used YCSB [12] to generate key-value workloads with real-world patterns. We first implemented a simple transactional key-value storage engine using `hashmap-tx` in Libpmemobj to handle requests generated by YCSB. We used the default settings of YCSB to generate workloads in the zipf distribution.

Figure 12 presents the average latency of 256 B objects for varying ratios of read and write operations. The optimization strategies employed by CodePM are specifically designed to reduce the overhead associated with write operations. As a result, the performance improvements of CodePM's optimizations become more pronounced as the write ratio increases, demonstrating enhancements ranging from 1.1–45.2% as the update rate escalates from 5% to 95%. Furthermore, CodePM-P, compared to CodePM, achieves a latency reduction of 4.6–14.7%. In the typical write-intensive scenario of YCSB `workloada`, with a read/write ratio of 50:50, CodePM and CodePM-P reduce the KVS latency by 38.9% and 45.3%, respectively, when compared to Pangolin. As the number of parity columns increases, the optimization impact of CodePM does not decline as significantly as that observed with object overwrite transactions. Nevertheless, CodePM-P still offers larger optimization benefits, ranging between 5.6–16.9%.

As shown in Figure 13, the throughput scalability of KVS under different strategies resembles the patterns we observed in the PMDK object overwrite benchmark. This similarity is largely attributable to CodePM's ability to markedly reduce PM writes for logging, thereby enhancing throughput by 225–350%. In contrast, Pangolin experiences a rapid saturation, followed by performance degradation.

F. Speculative Recovery Performance and Analysis

Although CodePM's instant recovery can quickly restart services after crashes, its reliability is still degraded during the recovery scanning window. To explore the recovery time of CodePM, Figure 14 shows the scalability of recovery scanning throughput under different hardware configurations. In a single channel, the throughput can exceed 6 GB/s, close to the

device's performance limit. Meanwhile, with 6 channels, the throughput can exceed 30 GB/s at peak. This means that recovery from the transient degraded state takes less than 30 seconds on our platform. The results indicate that log-free recovery can restore reliability with small background time investment after instant recovery.

VII. Related Works

Persistent Transactional Memory. Since x86 atomic write instructions are limited to 8 B, previous studies introduce undo or redo logging before in-place updates for PM crash consistency [11]. To mitigate the logging overhead on PM, various methods have been proposed. MorLog [17], ClobberNVM [18], and JustDo [33] reduce log writes by eliminating redundancy between log entries. DUDETM [3], TIMESTONE [34], and ASAP [35] reduce logging within the critical path using techniques decoupled logging, asynchronous logging, and group commit. These studies improve update performance specifically by optimizing the logging itself. On the other hand, Romulus [9] utilizes extra system redundancy to reduce logging overhead. It maintains primary-backup replicas to ensures crash consistency through synchronized replicas, thus eliminating the need for logging overhead. CodePM distinguishes itself from these works by utilizing additional system redundancy to achieve both crash consistency and fault tolerance.

Fault-Tolerant PTM. Memory errors arise due to device defects or write disturbance [7], [21]. To address this, several mechanisms have been developed to provide error correction, such as ECC [36] and RS-Code [37]. However, these methods focus only on protection at the device level while ignore system-level faults. [7]. Consequently, a series of works have provided system-level fault-tolerance for crash-consistent PTM by replication or parity. *1) Replication*: Libpmemobj-R and TENET ensure fault-tolerance through replicated PM pools and replicated logs on SSDs respectively [11], [26]. Kamino-Tx [6] uses chain-replicas to provide fault-tolerance and crash consistency, but still needs extra writes for inconsistency detection. Meanwhile, these replication strategies introduce higher space overhead. *2) Parity*: NOVA-Fortis [20] and Pangolin [7] use XOR parity and checksum to provide fault-tolerance with lower space overhead. TVARAK [38] and Vilamb [19] reduce the parity update overhead through novel hardware controllers and asynchronous parity updates respectively, but they introduce issues related to hardware modifications or reliability degradation. Pavise [5] co-designs crash consistency and fault-tolerance with minimal development efforts but underperforms compared to Pangolin. CodePM identified a functional overlap between two types of system redundancy and proposed parity-based consistency to fully eliminate logging overhead, thereby improving system performance.

VIII. Conclusion

Existing fault-tolerant PM systems suffer from significant writes due to logs for crash consistency. This paper proposes CodePM, a fault-tolerant transaction framework for PM that utilizes parity-based crash consistency to remove logs during updates. CodePM reuses the decoding capability of parity to simultaneously recover crash inconsistency. To guarantee data correctness, CodePM utilizes fine-grained fence and speculative recovery in updates and recovery. Additionally, we propose pipelined encoding and writing to hide update latency. Evaluation results show that CodePM achieves up to 3.4x throughput compared to the log-based strategy.

References

[1] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, 2020, pp. 169–182.

[2] C. Ye, Y. Xu, X. Shen, Y. Sha, X. Liao, H. Jin, and Y. Solihin, "Specpmt: Speculative logging for resolving crash consistency overhead of persistent memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023, pp. 762–777.

[3] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 329–343.

[4] T. David, A. Dragojevic, R. Guerraoui, and I. Zablotchi, "Log-free concurrent data structures," in *Proceedings of USENIX Annual Technical Conference*, 2018, pp. 373–386.

[5] H. J. Qiu, S. Liu, X. Song, S. Khan, and G. Pekhimenko, "Pavise: Integrating fault tolerance support for persistent memory applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2022, pp. 109–123.

[6] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 499–512.

[7] L. Zhang and S. Swanson, "Pangolin: A fault-tolerant persistent memory programming library," in *Proceedings of USENIX Annual Technical Conference*, 2019, pp. 897–912.

[8] P. Shang, S. Serish, and J. Wang, "Traid: Exploiting temporal redundancy and spatial redundancy to boost transaction processing systems performance," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 517–529, 2012.

[9] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 271–282.

[10] Y. Xu, C. Ye, Y. Solihin, and X. Shen, "Ffccd: Fence-free crash-consistent concurrent defragmentation for persistent memory," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 274–288.

[11] Intel, "Persistent memory development kit (pmdk)," 2018. [Online]. Available: https://pmem.io/pmdk/

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 143–154.

[13] M. Kwon, J. Jang, H. Choi, S. Lee, and M. Jung, "Failure tolerant training with persistent memory disaggregation over cxl," in *IEEE Micro*, vol. 43, 2023, pp. 66–75.

[14] Samsung, "Memory-semantic ssd," 2022. [Online]. Available: https://samsungmsl.com/ms-ssd/

[15] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, "Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering," in *Proceedings of the 17th European Conference on Computer Systems*, 2022, pp. 488–505.

[16] J. Yi, B. Dong, M. Dong, R. Tong, and H. Chen, "Mtˆ2: Memory bandwidth regulation on hybrid nvm/dram platforms," in *Proceedings of the 20th USENIX Conference on File and Storage Technologies*, 2022, pp. 199–216.

[17] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye, "Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 2020, pp. 610–623.

[18] Y. Xu, J. Izraelevitz, and S. Swanson, "Clobber-nvm: Log less, re-execute more," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 346–359.

[19] R. Kateja, A. Pavlo, and G. R. Ganger, "Vilamb: Low overhead asynchronous redundancy for direct access nvm," in *arXiv:2004.09619*, 2020.

[20] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 478–496.

[21] R. Wu, Z. Shen, Z. Yang, and J. Shu, "Mitigating write disturbance in non-volatile memory via coupling machine learning with out-of-place updates," in *Proceedings of IEEE International Symposium on High-Performance Computer Architecture*, 2024, pp. 1184–1198.

[22] Intel, "Intel optane persistent memory 100 series 512gb module product specifications," 2019. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/190351/intel-optane-dc-persistent-memory-512gb-module.html

[23] Y. Hu, L. Cheng, Q. Yao, P. P. C. Lee, W. Wang, and W. Chen, "Exploiting combined locality for wide-stripe erasure coding in distributed storage," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, 2021, pp. 233–248.

[24] S. Kadekodi, S. Silas, D. Clausen, and A. Merchant, "Practical design considerations for wide locally recoverable codes lrcs," in *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, 2023, pp. 1–16.

[25] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[26] R. M. Krishnan, D. Zhou, W.-H. Kim, S. Kannan, S. Kashyap, and C. Min, "Tenet: Memory safe and fault tolerant persistent transactional memory," in *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, 2023, pp. 247–264.

[27] L. Benson, H. Makait, and T. Rabl, "Viper: An efficient hybrid pmem-dram key-value store," in *Proceedings of the VLDB Endowment*, vol. 14, 2021, pp. 1544–1556.

[28] P. J. Nair, C. Chou, B. Rajendran, and M. K. Qureshi, "Reducing read latency of phase change memory via early read and turbo read," in *IEEE 21st International Symposium on High Performance Computer Architecture*, 2015, pp. 309–319.

[29] J. Wang, Y. Lu, Q. Wang, M. Xie, K. Huang, and J. Shu, "Pacman: An efficient compaction approach for log-structured key-value store on persistent memory," in *Proceedings of USENIX Annual Technical Conference*, 2022, pp. 773–788.

[30] J. Yang, Y. Yue, and KV. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020, pp. 191–208.

[31] Intel, "Intel intelligent storage acceleration library," 2023. [Online]. Available: https://github.com/intel/isa-l

[32] ——, "Ipmctl user guide," 2020. [Online]. Available: https://docs.pmem.io/ipmctl-user-guide/

[33] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 427–442.

[34] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable transactional memory can scale with timestone," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 335–349.

[35] A. Abulila, I. E. Hajj, M. Jung, and N. S. Kim, "Asap: Architecture support for asynchronous persistence," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 306–319.

[36] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 397–408.

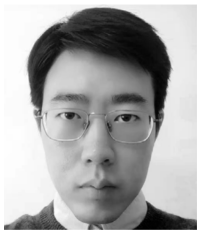
[37] S. Jeong, S. Kang, and J.-S. Yang, "Pair: Pin-aligned in-dram ecc architecture using expandability of reed-solomon code," in *Proceedings of the 57th ACM/IEEE Design Automation Conference*, 2020, pp. 1–6.

[38] R. Kateja, N. Beckmann, and G. R. Ganger, "Tvarak: Software-managed hardware offload for redundancy in direct-access nvm storage," in *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, 2020, pp. 624–637.

Guanglei Xu received the BE degree in computer science from the Huazhong University of Science and Technology (HUST), China, in 2020. He is currently working toward the Ph.D degree majoring in computer science at HUST. His current research interests include persistent memory and data reliability.

Yuchong Hu (Member, IEEE) received the BS degree in computer science and technology from the School of the Gifted Young, University of Science and Technology of China, Anhui (USTC), China, in 2005, and the PhD degree in computer science and technology from USTC, in 2010. He is currently a professor with the School of Computer Science and Technology, Huazhong University of Science and Technology. His research interests include improving data reliability (e.g., erasure coding) and Big Data storage systems. He has published 12 papers as the first/corresponding author in conferences FAST, INFOCOM, SoCC, and journals ACM-TOS, IEEE JSAC. He also published more than 40 articles in major journals and conferences, including IEEE TC, ATC, DSN.

Dan Feng (Fellow, IEEE) received the B.E., M.E., and Ph.D. degrees in computer science and technology from Huazhong University of Science and Technology (HUST), Wuhan, China, in 1991, 1994, and 1997, respectively. She is a Professor and the Dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, Non-Volatile memory technology, distributed file system, and massive storage system. She has more than 200 publications in major journals and international conferences, including IEEE TC, IEEE TPDS, IEEE TCAD, ACM-TOS, FAST, USENIX ATC, ISCA, EuroSys, HPDC, SC, DAC, et al. She has served as the reviewer of multiple journals, including IEEE TC, IEEE TPDS, et al, and the program committees of multiple international conferences, including FAST, SC, MSST, SRDS, et al.



Wenpeng He received the B.E. degree from Northeastern University, Shenyang, China, in 2017. He is currently a Ph.D student at Huazhong University of Science and Technology, Wuhan, China. His current research interests include trusted computing, non-volatile memory, secure and reliable memory system.

Junyuan Huang is a Ph.D. student advised by Prof. Yuchong Hu at Huazhong University of Science and Technology (HUST). He obtained the B.E. degree in Computer Science from HUST in 2024. His current research interests include memory disaggregation, system reliability.