Pavise: Integrating Fault Tolerance Support for Persistent Memory Applications

Han Jie Qiu
University of Toronto
Canada
hanjie.qiu@mail.utoronto.ca

Sihang Liu
University of Virginia
USA
sihangliu@virginia.edu

Xinyang Song
University of Toronto
Canada
xinyang.song@utoronto.ca

Samira Khan
University of Virginia
USA
samirakhan@virginia.edu

Gennady Pekhimenko
University of Toronto
Canada
pekhimenko@cs.toronto.edu

ABSTRACT

Persistent memory (PM) allows programmers to bypass the file system and efficiently manage persistent data directly. As a consequence, the application is now responsible for a non-trivial task—maintaining data crash consistency. In addition, it is highly desirable for today's production-grade storage systems to have fault tolerance to restore from data corruptions. Systems may provide fault tolerance through data redundancy. However, direct PM accesses bypass the system and make the data vulnerable to corruption. Without system-level support, it is the application's responsibility to maintain both crash consistency and fault tolerance, creating a demand for software tools to alleviate the burden from the application programmer.

Providing fault tolerance is challenging in the absence of system support because it is difficult to track data updates and efficiently update data along with its redundancy in a crash-consistent manner. Existing fault-tolerant mechanisms for PM applications either impose significant programming restrictions to the programmer or compromise on the level of protection they provide. This paper designs and implements *Pavise*[1], a software framework that provides protection for data within PM applications. Pavise uses a compiler pass to automatically track accesses to persistent data. It co-designs fault tolerance operations with the crash consistency mechanism to efficiently update data and its redundancy while maintaining the crash consistency guarantee. Pavise can be easily applied to existing PM applications with minimal manual effort and modest overheads. Our evaluation of common PM applications shows that Pavise achieves 83.2% (with ignore-list) and 70.9% (with conservative tracking) performance of the current state-of-the-art fault-tolerance software system, Pangolin. Because Pavise provides both application and library data with equally strong protection, Pavise can sustain a much higher error rate of 10^{-5} as compared to Pangolin's 10^{-7}.

[1]Pavise means a large European-style shield. Pavise source code is available at https://pavise.persistentmemory.org/.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PACT '22, October 10–12, 2022, Chicago,IL
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Computer systems organization** → **Processors and memory architectures**; **Redundancy**.

KEYWORDS

persistent memory, crash consistency, redundancy, fault tolerance

ACM Reference Format:
Han Jie Qiu, Sihang Liu, Xinyang Song, Samira Khan, and Gennady Pekhimenko. 2022. Pavise: Integrating Fault Tolerance Support for Persistent Memory Applications. In *Proceedings of PACT '22: International Conference on Parallel Architectures and Compilation Techniques (PACT) (PACT '22).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The persistent memory (PM) technology offers near DRAM-like performance and disk-like capacity and data persistence, such as Intel's Optane PM modules [23] and CXL-based PM [58]. These PM platforms allow fine-grained access to persistent data, therefore, exposing the high performance directly to the programmers, garnering a wide interest from both industry and academia [8, 17, 26, 28, 30, 40, 69, 71, 74].

To let programs better leverage the high-performance and byte-addressability features of PM, the Direct-Access (DAX) mode [66] is introduced to unify memory and storage. In DAX mode, the program maps a file backed by PM into its virtual address space, allowing byte-addressable access that bypasses the OS and file system. However, as data is persistent, the program is expected to recover to a consistent state and resume execution in case of a system crash or a power failure, which is known as the *crash consistency* guarantee.

Unfortunately, it is hard to provide crash consistency due to the presence of volatile cache hierarchy [10, 42, 43, 52]. Data must be persisted to PM in the correct order for the program to recover to a consistent state after failures. Writing an efficient and crash-consistent PM program is difficult and prone to errors [16, 42, 43, 52]. Therefore, PM libraries [26, 69] are developed to ease this burden for the programmer. The libraries will handle all the low-level direct accesses to PM while the programmer only needs to access PM through the library.

While maintaining crash consistency is important, persistent data residing in production-grade storage should also be protected against corruption. The ability to detect and correct errors in a system is

known as *fault tolerance*. Errors can appear for various reasons. On the hardware side, memory errors [20, 64, 68] can lead to incorrect data, CPU defects [11, 18] may write incorrect results, and firmware bugs may lead to lost or misdirected writes [27, 34]; while on the software side, various software bugs such as buggy kernels and file systems may corrupt data [1, 2, 32, 35, 55, 65, 78]. Errors that occur in the memory devices can be largely mitigated with techniques such as ECC [48, 76], Chipkill [9, 77], and wear-leveling [56]. However, errors that stem from the processor or software systems cannot be directly detected/corrected by memory-level protections, thus requiring stronger protections. In a conventional system, data residing in disks are protected by the system through data redundancy techniques [14, 46, 50, 53, 57, 72]. PM should receive the same protection as it is also persistent and is susceptible to various categories of data corruption [1, 2, 11, 18, 27, 32, 34, 35, 48, 62, 65, 78].

However, the existing file-system-level fault tolerance does not apply to PM under the DAX mode [72]. This is because reads and writes in DAX mode bypass the OS and file system, and the system cannot act on data updates (i.e., compute and update its data redundancy) without knowledge of those updates happening. Therefore, the burden of maintaining fault tolerance fully lies on the programmer, necessitating the creation of software tools to ease this burden.

Providing fault tolerance to PM programs in DAX mode has its own requirements and challenges. First, the data redundancy (e.g., checksum and parity) must be updated in a consistent manner along with the corresponding data, while the crash consistency guarantee is being maintained. Consistency only matters for data that become persistent, therefore, it is unnecessary to perform crash-consistent updates for every write that has yet been persisted, which will help reduce overhead. However, only periodically updating the redundancy can lead to scenarios where they become stale and unprotected. An ideal solution should decide the best timing and method of performing these updates, which is challenging given the complexities of the consistency guarantees and the performance trade-off.

Second, bypassing the system using the DAX mode means losing the ability to track which data are modified and require updates on redundancy. It is therefore necessary to identify and track dirty pieces of data. This can be done by leveraging the dirty bits in the existing memory paging system [35]. However, the system will then be dealing with data at page granularity, losing the benefit of fast fine-grained access that PM brings. Fine-grained tracking may be achieved by requiring the programmer to use a defined interface when developing the application. All PM accesses will be mediated by this interface. The required information such as the dirtiness and boundaries of data will be provided by the programmer through the programming interface and passed on to the fault tolerance mechanism. Unfortunately, such an approach requires significant manual effort and is not easily applicable to existing workloads [35].

Prior works have attempted to address these challenges. Hardware-based solutions, such as Tvarak [34], require dedicated hardware designs. Thus, those hardware-accelerated solutions are not considered in the scope of this work. On the other hand, existing fault-tolerant software solutions [35, 78] either restrict the programming model of the application and requires manual modifications to the code, or compromise the level of protection. Pangolin [78] is a PM transaction library that provides fault tolerance but requires users to use its API and programming model. For example, there have been attempts to adapt a PM-based Redis to Pangolin, but this modified more than 70% of the PM-related code [35]. Another prior work, Vilamb [35], does not restrict the programming model but performs asynchronous data redundancy updates to overcome the performance overhead. Thus, it can lead to time intervals in between updates in which the data redundancy is stale, and the system will be vulnerable to data corruption during these periods. Ideally, programmers should enjoy the full benefits of fault tolerance transparently, which motivates the creation of an automatic framework to provide fault tolerance in a plug-and-play fashion for existing systems. In this work, we introduce *Pavise*, a new software framework that automatically provides fault tolerance to PM applications.

PM access tracking. We observe that instructions that access PM can be effectively identified by a compiler pass, which can then be tracked via instrumentation. While the compiler does not have precise knowledge about whether an instruction will operate on volatile or persistent memory during runtime, it can insert extra code (i.e., if-statements on memory instructions' address) to check where the address belongs to during runtime. To guarantee all PM writes are tracked, all store instructions will need to be instrumented. To reduce the performance overhead of this conservative method, we make two optimizations. First, we make the distinction between application code and PM library code. As mentioned earlier, writing low-level PM code is error-prone and is often offloaded to PM libraries. Thus, applications tend to not access PM directly but instead using libraries. Therefore, it is sufficient to track PM updates within application code according to the use of library functions and PM primitives. Second, for sophisticated PM library code which may mix volatile and PM accesses, we conservatively track all memory instructions. To reduce the overhead, an *optional* ignore-list of functions that would never access PM may be provided by the programmer, wherein store instructions would not be instrumented. This technique allows us to track PM updates with minimal programmer's burden and without any OS/hardware changes. Based on this key insight, Pavise tracks PM updates at load/store granularity and stores checksums and RAID-4-style parity blocks for fault tolerance.

Efficient and crash consistent updates. As noted previously, it is challenging to efficiently update data redundancy while maintaining crash consistency. We observe that the correct persistent state of the data redundancy is also tightly coupled with crash consistency. Since both fault tolerance and crash-consistency-related operations aim to maintain a consistent and recoverable state of the PM region, any operation can be delayed until its effects are persistent (committed to PM). Leveraging this idea, Pavise atomically redirects all PM operations to a volatile *shadow copy* of the under-modification PM regions and commits all updates and their redundancy atomically to the original PM region via redo-logging. The programmer only needs to mark out the failure-atomic boundaries. This effectively reduces the overhead from providing fault tolerance but at the same time, maintains the protection and consistency guarantees.

We implement Pavise as a framework consisting of (i) a compiler pass, and (ii) a runtime library. We evaluate Pavise across seven PMDK benchmarks [26], and four real-life workloads including a PM-based Redis [60] and two different implementations of PM-based memcached [13]. We also perform an error injection experiment to evaluate Pavise's tolerance against errors and compare it with that of Pangolin. We found that Pavise has equivalent or a

better error-tolerance than Pangolin in our tested error rates (10^{-9} to 10^{-4}). In summary, this paper makes the following contributions.

- Providing fault tolerance to a PM application requires a redesign of its implementation. To ease the programmer's burden, we implement a compiler pass that automatically tracks PM-access instructions and a co-design of fault tolerance and crash consistency that provides efficient and recoverable redundancy for PM data.
- Based on these two key mechanisms, we implement Pavise, a framework that provides fault tolerance for PM applications with minimum programmer effort.
- Pavise achieves 83.2% (with ignore-list) and 70.9% (with conservative tracking) performance of the state-of-the-art fault-tolerant PM library, Pangolin [78], when comparing Pavise's directly converted applications with Pangolin's heavily optimized versions. We thoroughly test Pavise's fault tolerance by injecting errors. As Pavise covers not only the application data but also the underlying PM library with the *same level* of redundancy, it is able to sustain a high error rate of 10^{-5} as compared Pangolin's 10^{-7}.

2 BACKGROUND ON PERSISTENT MEMORY

Persistent Memory (PM) is a class of memory technology that offers near-DRAM access speed with disk-like capacity and persistence, such as Intel's recently-launched Optane DC Persistent Memory [23]. These PM modules sit alongside DRAM and can be accessed at byte granularity. These favorable characteristics have attracted wide interest in the industry and academic research [8, 17, 26, 28, 30, 40, 69, 71, 74].

There are mainly two ways to take advantage of PM. First, PM can be treated as a faster storage device and accessed through the file system interface (e.g., `read()` and `write()` system calls) [8, 12, 33, 71]. This method makes PM compatible with most existing programs but still keeps the intermediary of the OS and file systems. The second, more efficient way is to perform loads and stores from the user-space application, by directly accessing PM. This direct access to PM is enabled by the Direct-Access (DAX) support [66]. After mapping a PM-backed file (PM pool) to the program's virtual memory space, the program can bypass the OS and file system and gain the full benefits of both high performance and persistence. However, the cost is programmability. A PM pool is mapped into the program's virtual address space through `mmap`, wherein a PM object is located with a fat pointer—containing a base pointer to its PM pool and an offset within the pool. The base pointer which represents the start address of the mapping of the pool may be different across program executions, or within an execution if mapping is done multiple times. As such, pointer management in PM systems is more complex than normal volatile pointers [61]. This leads to the common use of the specialized libraries [7, 26, 69] that abstract away the low-level memory management (see Section 4.1 for details). Furthermore, programmers need to maintain crash consistency by themselves rather than relying on file system support. Crash consistency refers to the program's ability to recover in a consistent state and resume execution, in the case of a system crash or power failure. Software systems that manage persistent data are expected to provide the crash consistency guarantee.

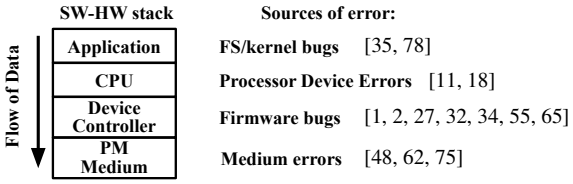

Figure 1: Sources of errors in the system stack.

Maintaining crash consistency is challenging due to CPU's caching [16, 17, 37, 41–43, 49, 54]. The order in which data becomes persistent may be different from the intended program order. As a result, without carefully manipulating the ordering, the state of the data at the point of a crash may not be recoverable. To enforce the order of persists, instructions such as `CLWB` and `SFENCE` from x86 [21] are used. `CLWB` allows the program to manually flush a cache line while `SFENCE` is a memory fence that orders `CLWB` instructions with respect to other `CLWB` and store instructions. For simplicity, we refer to a sequence of a `CLWB` and an `SFENCE` as a `persist_barrier` which ensures that a cache line is persisted before future writes.

To alleviate the programmer's burden on maintaining crash consistency, multiple PM libraries [4, 7, 15, 17, 19, 26, 69] have been developed. These libraries abstract away the low-level details of PM programming and the direct management of persistent memory regions. They manage PM on behalf of the programmer and provide higher-level APIs such as transactions [26, 69] to make crash consistency easier to maintain. Transactions utilize undo or redo logging to allow programmers to specify a piece of code to run atomically in case of a failure, thereby achieving crash consistency [4–6, 26, 29, 36, 69].

Even with various programming support for PM systems, prior works have shown that implementing a correct and efficient PM program (i) is difficult, (ii) requires expertise, and (iii) often involves trade-offs between performance and programmability [10, 16, 41–43, 51, 61]. The difficulty only exacerbates after adding the fault-tolerance requirement.

3 MOTIVATION

This section first introduces the fault tolerance guarantee in PM systems and then motivates the demand for a framework that automatically provides strong fault tolerance to PM systems in a crash-consistent manner.

3.1 Fault tolerance in PM systems

The crash consistency guarantee ensures *data recovery* in case of a failure, while the fault tolerance guarantee is another critical aspect that ensures *data correctness* in case of errors. Storage systems require fault tolerance to protect persistent data against errors. Figure 1 shows how data from the application travels through the system stack. Errors can be caused by both hardware and software and originate at multiple locations in the stack. On the software side, although writes bypass the file system under DAX-mode, a buggy kernel/FS may interfere with the application's address space which contains the PM region and corrupt data [35, 78]. On the hardware side, CPUs are susceptible to defects such as processor device errors and early life failures [11, 18], causing them to write corrupted data

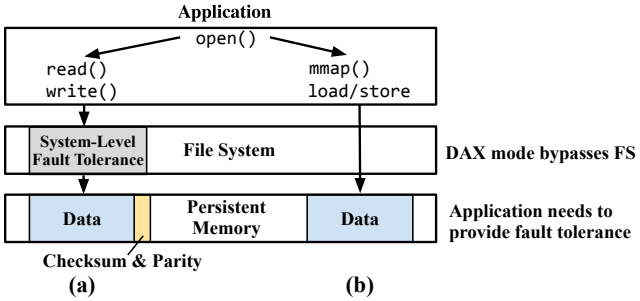

Figure 2: Application accesses PM via (a) FS and (b) DAX.

to PM; device controllers may have buggy firmware leading to lost or redirected writes [1, 2, 32, 34, 55, 65]. There have been reports where a potentially buggy firmware rendered corrupted data in Intel's Optane PM [27]. The PM storage media itself is also subject to errors and random bit flips, corrupting the data [48, 62, 75]. In this work, we follow the error model in which these various types of errors result in the outcome that corrupted data is read from or written to PM. The same error categorizations and modelling were done by prior work [35, 78]. Regardless of their sources, the errors manifest themselves as data belonging to the program being corrupted. In our fault tolerance evalations (Section 6.2.2), we simulate the errors as corruptions of data at random locations and different sizes within the program's PM address space. Techniques such as ECC [48, 76] and Chipkill [9, 77] can only mitigate errors that stem from the PM media. However, errors from the application, CPU, and firmware are not protected [34, 35, 78]. For example, a buggy firmware may write incorrect data to PM but ECC is calculated on top of such incorrect data.

For stronger protection, system-level checksums and data redundancy (e.g., parity) need to be maintained. Such techniques are widely used in various storage systems [14, 38, 46, 50, 53, 57, 67, 72, 79]. We believe PM systems deserve the same level of protection.

PM maintains data in a persistent state. Therefore, PM is no exception when it comes to the demand for fault tolerance [34, 35, 72, 78]. However, existing systems that provide protection for both traditional hard drives [57, 79] and PM [72] will not work under DAX-mode. Figure 2 shows two ways an application can access PM. Conventionally, when the application opens a PM file, it may access PM through standard system calls (e.g., POSIX) as shown in Figure 2a. The accesses will go through the file system, and if the file system supports fault tolerance as a feature, it will store system-level redundancy (e.g. block-based checksums and parity) for the application data. In the DAX mode, the application accesses PM directly, where loads and stores from the user-space application bypass the OS and the file system, as shown in Figure 2b. As the file system is unaware of any accesses, it is unable to mediate data transfers and provide real-time fault tolerance. Therefore, persistent data will be unprotected against errors that occur during runtime, when operating in DAX mode. This leaves the burden of maintaining fault tolerance fully on the programmer, which in turn creates the demand for software tools that can lessen or eliminate this burden.

Maintaining fault tolerance requires having the ability to perform error detection and error correction. The former can be achieved by storing checksums of data while the latter can be achieved by techniques such as RAID [53] (e.g., RAID-4 uses block-level striping with a parity disk). When a PM location is being updated, checksums will need to be updated as well to avoid being stale. And, they need to be verified at some point later when the program wishes to detect potential data corruptions before using the data. Similarly, data redundancy such as parity blocks used by RAID will need to be updated together with the data. For PM applications, the lack of file system support under DAX mode makes it necessary for the application to track the changes to persistent data and update the checksum/parity in a consistent manner, both of which are further complicated by the need for crash consistency. While a manually-crafted solution may solve the problem for one application, it is not a practical solution if we want to apply it to a multitude of workloads. Therefore, automatic solutions are preferred. In summary, the following requirements are needed for a fault-tolerant PM system to be widely adopted:

(1) Efficiently update dirty data along with its redundancy (checksum and parity) and maintain crash consistency.

(2) Provide the above features in a mostly automated way, with minimal programmer's effort.

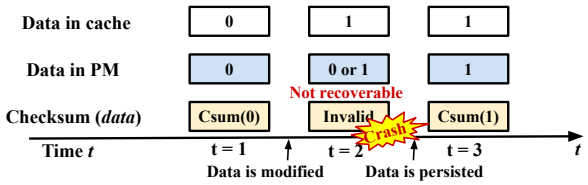

Figure 3: Challenge in providing crash consistency and fault tolerance. Checksum (`Csum()`) and data must be consistent.

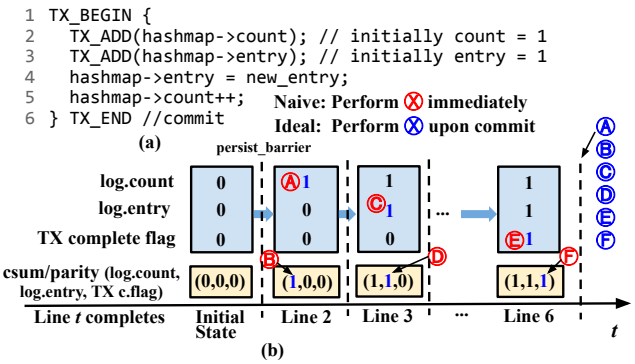

Figure 4: (a) Example code accessing PM using transactions (b) Timeline of PM data and their redundancy updates.

3.2 Challenges in Providing Fault Tolerance

This section discusses the challenges associated with satisfying the requirements for a fault-tolerant PM system.

Challenge 1: Crash consistency and performance overhead. We find that the redundancy must be consistent with data at all times for correct recovery. Figure 3 demonstrates the procedure of updating a persistent location (parity is omitted for simplicity). If a crash happens at $t = 2$, the data may have been updated but the checksum

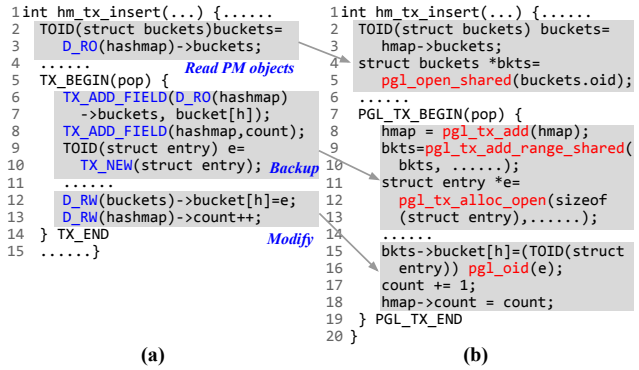

Figure 5: Comparison of hashmap insert function between (a) the original version, and (b) Pangolin [78].

is not. Therefore, the checksum is stale and inconsistent with data upon recovery. While ensuring crash consistency is necessary, a naive update scheme can cause major performance degradation.

Figure 4a shows a code snippet updating a hashmap entry using a transaction, where the old version of the data is backed up in an undo log using `TX_ADD`, where each `TX_ADD` contains `persist_barrier`'s to persist the logs. Figure 4b shows a timeline indicating the value of the data and checksum/parity after each operation, where each circled letter represents an update. Naively performing all the checksum/parity updates in-place before all the persist barriers will bring a significant performance overhead. At the same time, all data and the corresponding checksum updates need to be atomic to maintain crash consistency, introducing extra programming difficulties. Therefore, the *first challenge* is how can one devise a checksum update scheme delivering high performance as well as providing the fault tolerance and crash consistency guarantees.

Pangolin [78] is a prior work that attempts to overcome this challenge. It is a transactional library that provides fault tolerance to PM programs. Pangolin's solution to the crash consistency and performance challenge is to perform all data and checksum updates atomically at the end of a transaction via redo-logging. This will reduce the number of `persist_barriers` when updating checksums, and make the atomic update more efficient. However, Pangolin has two major limitations. First, Pangolin's fault tolerance protection is not uniform—some data may be more vulnerable than others (see Section 6.2.2 for details). Second, it is a tailor-made library that requires the program to use its programming model and interface. This can cause significant inconvenience when the user tries to adapt existing workloads to Pangolin.

Challenge 2: Automation of fault-tolerance support. The checksum and parity updates require the knowledge of which data are updated at what times. However, DAX-mode file systems allow programs to directly access the memory without the file system. Therefore, the *second challenge* is how can one *automatically* identify and track dirty data in the absence of file system support.

Pangolin's solution to tracking is to let the application explicitly convey the required information (PM objects' address and size) to the fault-tolerance mechanism via Pangolin's API. However, this solution requires a lot of manual effort and code changes. Thus, it cannot be easily applied to a wide variety of real-world applications, potentially limiting its practicality and impact. Figure 5 compares the original version of a hashmap insert function (`hm_tx_insert()`) with the manually-modified version that is based on Pangolin. The function first reads the PM objects and then starts a transaction that backs up these PM objects to an undo-log, and finally updates them in-place. When adapting to Pangolin, the code is *heavily modified* (as highlighted in Figure 5), as the interface that opens/reads/modifies objects is different. It is worth pointing out that the example in Figure 5 is transaction-based, which makes it easier to adapt to Pangolin. Other workloads can require a large portion of the code to be rewritten. For example, Ketaja et al. [35] changed more than 70.04% of the PM-relevant, effective lines of code (eLOC) in Redis to adapt to Pangolin. If the workload directly manages PM with low-level primitives (e.g., `CLWB` and `SFENCE` [21]), all the code regions related to PM need to be re-implemented. We conclude that adapting a workload to a fault-tolerance library requires a thorough understanding of the program and a careful re-implementation.

Vilamb [35] does not require significant code change by utilizing page-based checksums. However, in order to reduce overhead, Vilamb only updates checksums periodically, leaving data unprotected between updates. In summary, existing works cannot provide a solution that is directly applicable to existing PM applications with minimal manual effort, without compromising the level of fault tolerance. Therefore, it is desirable to have a *highly automated* framework that provides *strong fault tolerance* to the user's program.

4 PAVISE: KEY IDEAS

To address the aforementioned challenges, we design and implement Pavise, a framework that provides fault tolerance to PM applications. Pavise requires minimal manual programming effort while still providing strong fault-tolerance protection and maintaining crash consistency.

4.1 Automatic Tracking of Modified PM Data

As discussed in Section 3.2, a coarse-grained tracking mechanism can result in high performance overheads but fine-grained tracking introduces extra manual effort. Ideally, we would like to have the benefits from both worlds—low performance overhead but high automation. We observe that PM operations can be narrowed down to a few primitive forms, most of which can be extracted by the compiler. PM operations in PM applications (at least those that are publicly available [26, 40, 49, 60]) are in the following forms:

- Regular memory instructions, e.g., store, clwb/clflush, and sfence.
- Call instructions to memory-movement functions, such as `memcpy` and `memset` that modifies a chunk of memory.
- Atomic instructions that access synchronization primitives, such as compare-and-swap (CAS), or synchronization functions, such as mutex for PMDK's pool [26].

Given the categories of methods (instructions and functions) that access PM, a compiler pass can identify these PM-modifying methods and automatically insert code to track their modifications. However, there is a new challenge—the compiler pass cannot determine whether an instruction operates on PM or DRAM at compile time.

One way is to conservatively instrument all instructions and check whether they access PM or DRAM at runtime by injecting

if-statements on the addresses. However, the drawback is the performance overhead. To mitigate the overhead, we perform two optimizations, applied to different parts of the program. As discussed in Section 2, PM accesses within most applications are offloaded to a PM library. The first optimization we perform is to precisely instrument PM instructions within the application code, according to the use of PM library methods (e.g., direct read/writes to PM objects via `D_RW()` in `libpmemobj` [25]) and primitive functions (e.g., memory copy via `pmem_memcpy()` in `libpmem` [24]). The second optimization is applied to the sophisticated PM library code, where there can exist a mix-use of methods that access volatile and persistent memory. We first provide a conservative method that tracks all memory instructions. On top of that, we allow the programmer to supply an ignore-list of instructions that would never access PM. Those instructions within the list will not be instrumented. The ignore-list can be obtained from the library documentations and library functions profiling. In summary, Pavise tracks all store instructions in the PM library unless it is on the ignore-list and tracks all PM-accessing functions in the application code, to guarantee correctness. Pavise computes and verifies checksums based on the PM data it tracks, confirming the correctness of instruction tracking at runtime.

With the mechanisms above, the entire process of tracking PM modifications can be done automatically (unless an ignore-list is needed) and efficiently with our compiler pass. Therefore, the engineering effort required to port an existing PM application to use Pavise is minimal, especially compared to prior work. The user only needs to add commit points, modify their build configuration to link their program with the Pavise library, and run Pavise's compiler pass. Methods of adaptation from previous work such as Pangolin [78] would require extensive change to allocation and access functions, as shown in Figure 5.

Note that Pavise's protection scope is the DAX-mmapped PM region, which is directly managed by the PM program. Other data such as metadata belonging to the file system is instead protected and managed by the file system, whose protection complements that of Pavise. Furthermore, Pavise can detect and correct corruptions or undesired memory modifications originating outside of the PM program, but issues from the PM program itself, such as a bug, are not within the scope of Pavise. These programming issues can be detected by prior testing works for PM software [16, 42, 43, 52].

4.2 Co-design of Fault Tolerance and Crash Consistency

We observe that the validity of data is determined by the crash consistency semantics of the application. The example of Figure 4a performs a failure-recoverable insertion operation to a hash table (`hashmap`), where updates at lines 4 and 5 are *not* considered *valid* until the procedure (e.g., a transaction) completes. Therefore, an ideal solution is to perform all updates atomically at the end of the transaction, as the operations marked in *blue* demonstrate. More generally, the fault tolerance guarantee only needs to be enforced for the *consistent and recoverable version of the persistent* data. If an instruction does not affect the consistency of the persistent state as it completes, there is no need to immediately update its redundancy. Rather, it can be delayed until its modification becomes a consistent version in PM (i.e., committed to PM). We observe that this characteristic naturally aligns with the working principles of redo-logging, which directs PM updates to the logs and commits them together at the end of a transaction (or a failure-recoverable PM update procedure).

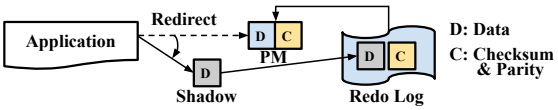

Figure 6: Co-designing fault tolerance and crash consistency.

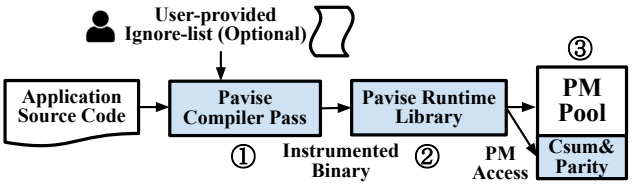

Figure 7: Overview of the Pavise framework. Pavise components are marked in blue.

For workloads that are already transaction-based, the original transaction boundaries can be kept and no changes are needed. For low-level workloads that do not use transactions, we allow the user to place commit functions where they want (e.g., at the end of a function that completes an insertion operation) and conveniently "transactionalize" the workload. Based on this observation, we co-design fault tolerance and the crash consistency semantics of the program in the following steps: (i) redirect all PM operations to a *volatile* shadow copy first (while keeping track of them), and (ii) commit the shadow copy to the real copy periodically via redo-logging, postponing checksum computation and verification until the commit point. Figure 6 illustrates this procedure. The data being written to PM by the application will be redirected to a shadow copy (details of the redirection mechanism are in Section 5.3). During commit, the shadow data will be copied to a persistent redo-log and committed along with its data redundancy. The commit point can be specified by the programmer inside the application. For transaction-based programs, this can simply be the end of a transaction (See Figure 4a). For programs using low-level primitives (e.g., hashmap-atomic [26] and memcached [40, 49] in Section 6.1), a commit can happen when a functional piece of code ends (e.g., node insert). In both cases, the programmer is only required to add a few function calls in their program where they would like to commit. In general, the commit point is inserted to mark out the end of a procedure that needs to be failure-atomic, and allow recovery to the stage either before or after the procedure happens if a failure occurs in the middle. The commit operation stores data and checksum atomically with redo-logging. Any changes made before the commit will remain in the shadow copy. Therefore, the persistent state will always be consistent.

5 PAVISE: DESIGN AND IMPLEMENTATION

This section discusses the design and implementation of Pavise. We first give an overview of the design and then describe each component of Pavise and our design choices.

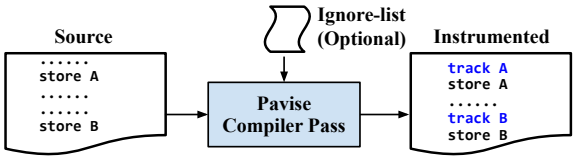

Figure 8: Inserting tracking functions via compiler pass.

Algorithm 1: Pavise compiler logic.

```
1  begin CompilerPass(funcList, ignoreList)
2      PMInstList.appendGetAppPMLibInst
3      PMInstList.appendGetPMLibMemInst
4      PMInstList.removeignoreList
5      foreach inst ∈ PMInstList do
6          if isStorePMinst.type then
7              inst.insertBeforePaviseTrackStore
8          else if isLoadPMinst.type then
9              inst.insertBeforePaviseTrackLoad
10         else if isCallinst.type then
11             callee = inst.getCalleeName
               /* Expandable list of library functions */
12             if callee ∈ funcList then
13                 inst.insertBeforePaviseTrackCall
14     end
15 end
```

5.1 Overview

Pavise consists of two components: (i) a compiler pass and (ii) a runtime library. Figure 7 shows an overview of the workflow. ① The *Pavise compiler pass* compiles the source code. During compilation, it inserts trackers before memory access instructions/functions. For application code, only instructions from the use of PM library functions are instrumented. For PM library code, all store instructions are instrumented conservatively. If an ignore-list of volatile-access-only functions within the PM library is provided, instructions in those functions will not be instrumented (Section 5.2). ② The *instrumented binary* is executed and trackers are called during runtime. These trackers invoke the *Pavise runtime library*, which keeps track of PM accesses, i.e., address and size. ③ The *Pavise runtime library* commits the data and redundancy to the actual persistent memory and performs verification (Section 5.3–5.5).

5.2 Compiler Instrumentation

The Pavise compiler pass is based on LLVM [39]. It inserts a tracker function before the instruction (as shown in Figure 8). The compiler pass handles application code and library code separately. For application code, the compiler precisely instruments PM-access instructions according to the application's use of PM library functions. For example, the use of `libpmemobj` primitive method for direct PM read/write, `D_RW(obj)`, reveals the access to PM; the use of `libpmem` function for writeback. For library code, the compiler performs a conservative instrumentation by default, unless an *optional* ignore-list for non-PM-accessing functions is provided. In our experiments, we obtain an ignore-list by checking the library documentation [26] to first filter out non-PM functions, and then perform a thorough

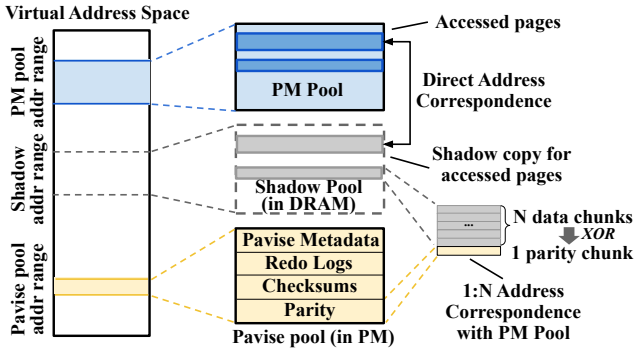

Figure 9: Memory layout of the program.

profiling to identify other non-PM functions. Algorithm 1 describes the compiler procedure in detail. Note that the existing transactional programs (PMDK-based applications in our evaluation) use undo-logging to backup data before modification, which is unnecessary under Pavise. Therefore, Pavise automatically disables undo logging interface functions (e.g., `TX_ADD`) during compilation. After compilation, the program will call the tracking function before each instrumented instruction, and pass the modified memory address and its size to the Pavise runtime library.

5.3 Pavise Runtime Library

The Pavise runtime library serves three purposes: (i) redirects and tracks PM accesses, (ii) manages the tracked PM accesses, and (iii) updates the redirected modifications and their new redundancies, while maintaining crash consistency.

Write redirection. In order to postpone all persistent operations to a commit point (marked by `PaviseCommit()`), Pavise redirects all PM updates (including functions that access PM) to a shadow copy. This is done by intercepting the calls to the `mmap` system call in the program. Instead of having `mmap` return the regular PM mapping address to the program, the Pavise runtime performs another anonymous mapping (by setting the `MAP_ANONYMOUS` flag in `mmap`) which represents the shadow copy and returns the base address of the shadow. This way, the actual PM mapping will only be accessible by the Pavise runtime, and the PM accesses in the original program will be redirected to the shadow.

Figure 9 shows the memory layout of the application's virtual address space during execution. The shadow pool and the PM pool have a one-to-one direct correspondence for each address. Every byte in the shadow is a fixed offset from the corresponding PM byte. Even though the shadow region occupies the same size of address range as the PM pool, the actual amount of DRAM allocated during runtime will only be the current working set of the program. In case there are multiple PM pools, the runtime library also allocates the same number of shadows. In addition, Pavise reserves its own PM pool (later referred to as Pavise pool) to store its metadata, the redo logs, as well as the checksums and parity. Within the Pavise pool, the addresses in the parity block also have a direct one-to-many correspondence with the PM pool (see Section 5.5 for details). The entire shadow region is initially set to be read- and write-protected. Any read or write from the program will trigger

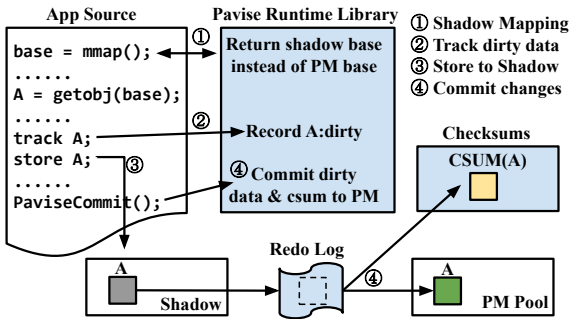

Figure 10: Update process for data and checksum.

segmentation faults, which are handled by Pavise's custom signal handler (`pavise_signal_handler` in the Pavise library). Once the handler is triggered, Pavise will copy the currently-accessed page from the PM mapping (which is backed by the actual PM pool) to the shadow, and mark the page as read/write accessible. In essence, this is an on-demand copy-on-read/write scheme. Subsequent reads will read from the latest data in the shadow, while writes will stay in the shadow without affecting the PM pool until the commit point.

Handler for tracker functions. During runtime, every time the tracking function is called, the handler will check the address to confirm it is a PM address and append the address and size to a list. This way, Pavise runtime can calculate their redundancies and apply them back to PM during commit.

Walk-through of Pavise runtime. Figure 10 shows the process in which data is tracked and updated. ① When the application performs `mmap`, Pavise maps the shadow and returns it to the application. ② Before any PM operations, the target address is tracked and recorded by the Pavise runtime library. ③ PM-accessing instructions in the application will now instead be performed on the shadow. ④ During commit, Pavise will compute checksums, and commit the data and checksums to the actual PM via redo-logging.

Crash consistent updates. Once the program hits a specified commit point, where `PaviseCommit()` is called, Pavise will begin committing all the writes recorded by the tracking functions in a crash-consistent manner. Algorithm 2 shows the pseudocode of Pavise's commit routine. Verification of PM data (see Section 5.4 for details) is first performed before the commit. Commit is done via redo-logging similar to libpmemobj [26]. The checksums are computed and added to the redo-log, along with the data so the updates are atomic. After all log entries are written, the log is persisted. The log entries will then be applied to their destinations in the PM pool followed by a persist. If a crash happens before the redo log is completed, all entries will be discarded; if a crash happens after the redo log is completed but before all entries are applied, all entries will be reapplied until completion. Therefore, the updates are failure-atomic and the recovered state will always be *crash consistent*.

5.4 Checksum Computation and Verification

Checksum computation. Checksums are computed at the granularity of fixed-size chunks. The size is configurable and we empirically determine that 512 Byte chunk size performs the best. The checksum is recomputed for the entire region during commit if any data within the chunk region is recorded by the tracking function to be dirty.

Algorithm 2: Pavise commit routine.

```
1  begin PaviseCommit(dirtyAddresses, redoLog)
2      verifyChecksums
3      chunks = getDirtyChunksdirtyAddresses
       /* Chunk-grained checksum update with logging */
4      foreach chunk ∈ chunks do
5          csum = adler32chunk
6          redoLog.append&csum
7      end
       /* Fine-grained data update with logging */
8      foreach addr ∈ dirtyAddresses do
9          redoLog.appendaddr
10         updateParityaddr
11     end
12     persistredoLog
13     foreach entry ∈ redoLog do
14         entry.apply /* Copy to PM */
15         persistentry.dest
16     end
17     redoLog.discard
18 end
```

The checksum algorithm is Adler-32 [44] and we utilize Intel's high-performance ISA-L library [22]. The checksum is stored in a hash table (for faster access) located in the Pavise pool. During commit, checksums are first logged in the Pavise pool and then copied to the final hash table location to guarantee crash consistency.

Checksum verification. The checksum verification mechanism detects data corruption of each chunk, by recomputing the checksum of the chunk and comparing it with the existing checksum. A checksum mismatch indicates the data chunk was corrupted and needs to be corrected. Before each commit point, Pavise's runtime library will verify and correct all the tracked data (Section 5.5).

Verification schemes. The overhead of verification depends on the verification scheme that determines which data will be verified and when verification happens. Pavise provides two checksum verification schemes: verify *all* stores (*AllStores*) and verify *all* loads (*AllLoads*). In the *AllStores* scheme, we perform checksum verification on the data address of all PM stores collected by the tracking functions. In the *AllLoads* scheme, we perform checksum verification on all PM loads that access PM. Thus, to support load tracking, the *AllLoads* scheme extends the Pavise compiler's scope to include loads that access PM. And, those loads will be collected by the tracker during runtime. In comparison, *AllLoads* is a stronger scheme as data to be loaded are always guaranteed to be intact. However, as loads are usually more frequent than stores, the performance overhead of *AllLoads* is higher. The user can choose the scheme according to their needs. We evaluate the performance overhead of both schemes in Section 6.2.4.

5.5 Parity Computation and Correction

Pavise adopts RAID4-style parity, similar to prior fault-tolerance works for PM (Pangolin [78] and Tvarak [34]). Although RAID was designed for traditional disk storage, its principle can be applied to other types of data storage as well. Pavise borrows the RAID4

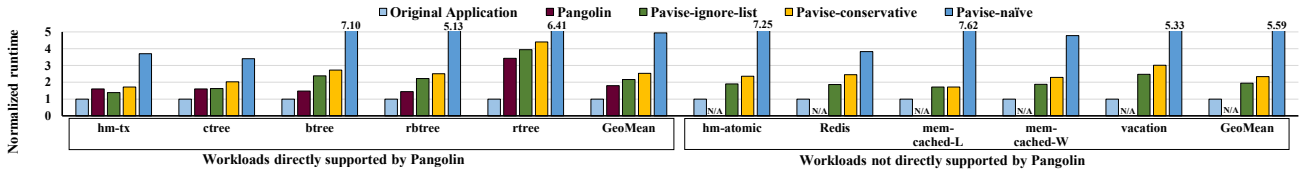

Figure 11: Run time of Pavise, Pangolin, and the original applications.

principle of partitioning the data into multiple blocks and a parity block to protect against data corruption. As shown in Figure 9, the parity block has a 1-to-N correspondence with the data in the PM pool. Thus, each column can correctly restore if 1 out of N bits is corrupted using parity. The entire PM pool is divided into N rows, with a parity row stored in the Pavise pool which is the XOR of all N rows. Consequently, each byte-width data column consists of N bytes of data plus one byte of parity (N is a user-configurable parameter). A larger N will result in more bits in the original data associated with each parity bit and reduce the storage overhead. However, there is a higher chance that more than one corrupted bit are associated with the same parity bit (which will render the data uncorrectable), thereby lowering the strength of protection. Parity is computed using Intel's ISA-L library [22] during the commit phase. Similar to prior work, Pangolin [78], parity updates are not logged to reduce the performance overhead. As parity will only be used when data corruption is detected and needs to be corrected, data will only be truly lost if corruption and a crash happened at the exact same time on the same set of data-parity blocks—an extremely rare event. Upon detecting a checksum mismatch, the chunk of data is restored using its parity (if parity is valid) along with other data in the same data-parity column. If there is no more than one corruption per column, data can be recovered.

5.6 Scope of Protection and Fault Model

Pavise maintains checksums and parity uniformly across the DAX pool managed by the application, as indicated by the PM pool region in Figure 9. If the application uses a PM library (e.g., PMDK [26]) to access PM, the library partially or fully manages the PM pool on behalf of the application, and maintains its own metadata stored within the PM pool. All data within the PM pool are treated uniformly by Pavise with checksums and parity computed, including PM library metadata if it exists. Therefore, all data used by the application and PM library are under the scope of Pavise's protection.

Pavise stores all data required to provide fault tolerance within the Pavise pool, which is a separate memory region from the application's PM pool as shown in Figure 9. These data include Pavise metadata and fault-tolerance data (checksums and parity) of the application data. The former is needed for crash consistency and is also protected by checksums. The latter allows Pavise to detect and recover from errors within the application PM pool.

Pavise assumes errors in the PM pool manifest themselves as bit-flips of arbitrary patterns. While the potential sources of the errors are described in Section 3.1, for simplicity we treat the generation of errors as a black box and only simulate bit-flips within the PM pool as end results in our evaluations, as described in Section 6.2.2. During runtime, the checksum validates the data, and if an error is detected through a checksum mismatch, the parity will repair the corrupted data. Corruption can also occur within the Pavise pool. If only one of the three pieces of information: application data, checksum, and parity, contains errors, Pavise can still recover to a correct state based on the other two. If two or more of them are corrupted, then Pavise cannot recover. However, the latter case has a much lower probability of happening. We only observe uncorrectable errors at high error rates in our experiments (see Section 6.2.2).

CPU	Intel Cascade Lake, 8 cores, 16 threads, 3.0 GHz
Caches	64kB L1, 1MB L2, 10MB L3
DRAM	64GB DDR4 2666 MT/s
PM	512GB Intel Optane DC Persistent Memory
OS	Ubuntu 18.04.3, Linux kernel v5.4
Env.	LLVM-11, gcc-7.5, Intel Pin-3.10, PMDK-1.10

Table 1: System configuration.

Application	Configuration
hashmap-tx (hm-tx), ctree, btree, rbtree, rtree, hashmap-atomic (hm-atomic)	1M insert requests, 256B value size
Redis, Memcached-Lenovo, Memcached-WHISPER	100k set requests, 256B value size
Vacation	100k relations, 200k tasks

Table 2: Applications and their configurations.

6 EVALUATION

In this section we first describe our methodology and then present the results of our evaluations.

6.1 Methodology

System Configuration. Table 1 shows the system setup for our evaluation. The Optane DC Persistent Memory Modules are set to the *AppDirect* mode, and exposed as a DAX device through the ext4-DAX file system [45] The application access PM through the `mmap` system call.

Applications. We perform experiments using several benchmarks and real-life PM applications, as listed in Table 2. Our workloads include PMDK benchmarks [26], Redis [60], memcached [13], and vacation [47]. PMDK benchmarks are simple key-value store structures; Redis is an in-memory database and we use a version adapted for PM [28]; vacation is a travel reservation system from the STAMP

suite [47]; memcached includes two versions, one is developed by Lenovo [40] and the other is from the WHISPER benchmark suite [49]. Note, for PMDK transactional applications, Pangolin's versions are heavily modified and optimized for their interface. In comparison, Pavise is directly applied to the original code.

Baseline and Pavise configurations. We take Pangolin, a state-of-the-art fault-tolerance software library for PM workloads, as the baseline. As Pangolin is not compatible with generic PM applications, we only evaluate the compatible ones based on `libpmemobj` [25]. For Pavise we evaluate three configurations:

- **Pavise-naive:** Performs fault tolerance operations after *every store instruction*, as illustrated in Figure 4b.
- **Pavise-conservative:** Performs fault tolerance operations *only at commit points* (co-design of fault tolerance and crash consistency). Instrument all memory instructions within PM libraries.
- **Pavise-ignore-list:** Performs fault tolerance operations *only at commit points*. Instrument a subset of PM-related memory functions within the PM library (detail in Section 5.2).

Metrics and Parameters. We collect the execution time for all workloads and configurations and compare Pavise against the baselines. We evaluate each setting 20 times and calculate the relative variance σ^2/μ and relative standard error. We evaluate Pangolin with its default mode where checksums and parity features are enabled. For Pavise, we set the checksum chunk sizes to 512 bytes and the data-parity ratio (N) to 100 (the same parity strength as Pangolin [78]). We also evaluate a configuration with N=20 for higher fault-tolerance strength in Section 6.2.2.

6.2 Evaluation Results

6.2.1 Performance. Figure 11 shows the normalized execution time of the original applications, Pangolin, and Pavise across all workloads (lower is better). The verification scheme of Pavise is to verify upon stores (*AllStores*)—the same scheme that Pangolin uses. On average, Pavise brings fault tolerance to the PMDK benchmarks at an overhead of having 2.16× run time with our ignore-list and 2.54× with the conservative tracking (the maximum relative standard error is 0.89%). In addition, the naive approach of committing after every store instruction results in a prohibitively high overhead of 4.94× run time when compared against the original PMDK application. On the other hand, Pavise (with an ignore-list) is on average 2.28× faster, as it delays fault tolerance operations until PM commit points. We conclude that our proposed co-design of fault tolerance and crash consistency significantly reduces the fault tolerance overhead.

While Pangolin on average performs at 1.80 × run time across the workloads it supports, the process of applying Pavise to the workloads is much easier and near-automatic. In comparison, adapting a workload to run on Pangolin requires heavy modifications. We also notice that Pavise performs worse on btree, rbtree, and especially rtree. This is likely attributed to the size and space pattern of the data structures leading to inefficient checksum updates. The problem is less severe in other real-life workloads as they are less PM-data intensive than the PMDK benchmarks. In summary, although Pavise is automatically applied to the original PMDK applications, it can achieve 83.2% performance with ignore-list and 70.9% with conservative tracking compared to the heavily optimized Pangolin version.

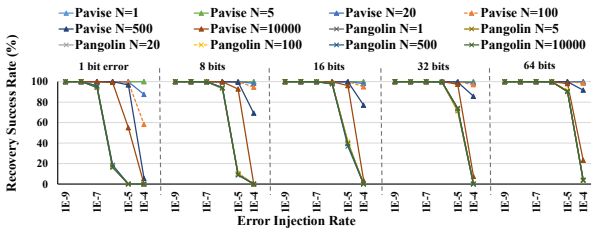

Figure 12: Recovery success rate of Pavise and Pangolin across PMDK benchmarks with different error injection rates, data-parity ratio (N), and error sizes.

6.2.2 Fault tolerance test. To mimic error patterns discussed in Section 3.1, we inject errors at various sizes, ranging from 1 bit to 64 bits. To simulate PM storage media errors, we randomly flip single bits within the PM pool. To simulate processor hardware errors, we inject errors at various word sizes, including 8 bits, 16 bits, 32 bits, and 64 bits. We inject errors to the PM pool to evaluate the error-correction capability of Pavise and our baseline, Pangolin [78]. We take error rates, varying from 10^{-9} to 10^{-4} [48, 59, 75]. Such uniform-probability error models are suggested by prior studies on errors in persistent devices [3, 59, 62, 73, 80]. While our default data-parity ratio is 100:1, we evaluate Pavise and Pangolin at extreme data-parity ratios ranging from 1:1 to 10000:1. More ratios are also evaluated and discussed in section 6.2.3. For each error injection rate in each workload, we perform 100 trials and report the number of trials in which error correction and recovery are successful, in order to evaluate the fault-tolerance capability of both Pavise and Pangolin. Figure 12 shows the recovery success rate at different error rates, data-parity ratios (N), and error sizes. At every setting, Pavise has an equal or a higher chance of successful recovery than Pangolin. At lower error rates (10^{-9} and 10^{-8}) both Pavise and Pangolin can successfully recover in all trials. At higher error rates (10^{-7} to 10^{-4}), Pavise performs better than Pangolin and has a higher chance of successful recovery in all cases.

After analyzing the implementation of Pangolin, we found that Pangolin does not have the same protection strength for different PM data. PM objects, e.g., a tree node structure, are protected strictly with parity and checksums, whereas Pangolin metadata and logs are only replicated. As a result, if errors occur simultaneously in the main copy and the replica, respectively, both versions end up being corrupted but there is no way to locate the error. In contrast, Pavise provides uniform protection across all PM data in the granularity of fixed-sized chunks. Therefore, Pavise will only fail to recover if two errors fall in the same parity column, which is less likely than two errors falling into the same main-replica pair in Pangolin given the large size (3MB) of Pangolin's logs.

6.2.3 Performance vs. recoverability. To study the tradeoff between Pavise's performance and recoverability, we evaluate the two metrics across different data-parity ratios (N). The lower the ratio, the higher the recovery success rate and vice versa. This is due to having more parity bits per data bit, thereby having higher data redundancy. Since having 100% recoverability is unattainable due to the worst case that all data and redundancy become corrupted, we used the lowest N possible ($N = 1$) as the upper bound of recoverability for this

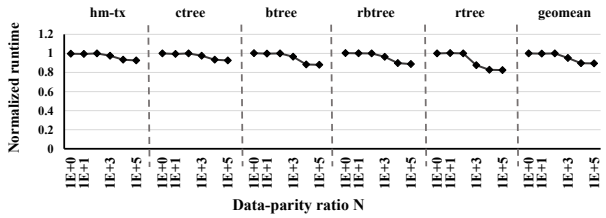

Figure 13: Run time of Pavise-ignore-list under various data-parity ratios (N). The run times are normalized against N=100, which is the default setting of Pavise.

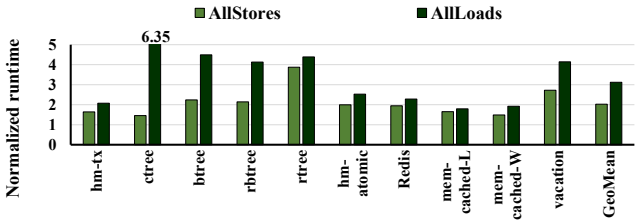

Figure 14: Execution time of different verification schemes of Pavise on PMDK benchmarks.

study. Similarly, 0% recoverability is simply the baseline setting without having Pavise. We use a large value of N ($N = 10000$) to approximate the lower bound of recoverability. Figure 13 shows the normalized run times of Pavise under various data-parity ratios (the maximum relative standard error is 1.07%). At a low data-parity ratio (e.g., 1:1), Pavise has higher overhead due to more time spent on computing and storing parity.

We also observe that the run times remain largely constant for data-parity ratios less than 100. While setting a lower ratio will provide better recoverability without compromising performance, the storage overhead to store the parity bits become proportionally larger (i.e., $N = 1$ uses 100× more space than $N = 100$). Since the recoverability under $N = 100$ is not much lower than that under $N = 1$, we choose $N = 100$ as our default ratio for a good tradeoff between performance, recoverability, and storage overhead.

6.2.4 Verification schemes. Pavise has two verification schemes: verify all stores (*AllStores*) and verify all loads (*AllLoads*). The default scheme is *AllStores*, which verifies all data that that were modified before committing the changes. This is the same default scheme utilized by Pangolin. Figure 14 shows the performance (with ignore-list enabled) of both verification schemes (the maximum relative standard error is 0.49%). The execution times are normalized to the no-fault-tolerance baseline. The overheads in different verification schemes vary. A stricter verification scheme such as *AllLoads* (as there are typically more loads than stores) will have a higher overhead, but at the same time will provide a higher level of protection because more data is verified. *AllLoads* will also have a higher overhead from tracking functions that handle both loads and stores.

6.2.5 Overhead breakdown. Figure 15 shows the overhead breakdown from different operations within Pavise. The configuration is Pavise-ignore-list and the verification scheme used is *AllStores*. For *AllLoads*, the breakdown will be the same except for a higher overhead from verification. For verification, the overhead includes iterating through a list of data chunks, computing the newest checksums,

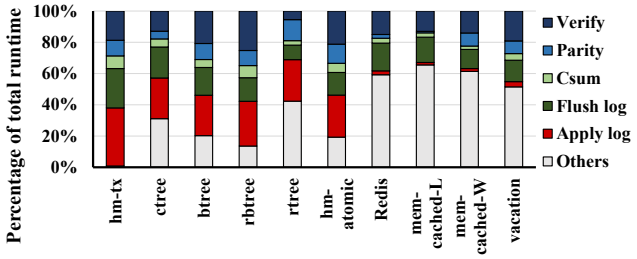

Figure 15: Overhead breakdown of Pavise (*AllStores*) across PMDK benchmarks.

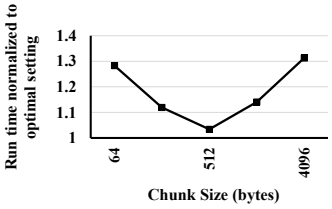

Figure 16: Performance sensitivity to chunk sizes.

and comparing against old checksums. For parity and checksum, the overhead mainly comes from Intel's ISA-L library computes the checksum/parity of the data. The overhead from logging includes creating and flushing the logs and applying the log to the final PM locations during commit. The remaining run time may include non-PM operations from the original workload, tracking operations, and Pavise internal bookkeeping operations.

The breakdown shows that logging is the main source of overhead. Log-flush is the extra overhead Pavise introduces by writing the data twice, once from the shadow pool to the redo log and once from the redo log to the PM pool. While the computations of checksum and parity have less overhead than logging, they still take a considerable fraction of the total run time. The majority of overhead from verification is also from checksum computations. For real-world workloads such as Redis and Memcached, a large portion of the run time is attributed to other non-PM operations (e.g., network stack).

6.2.6 Sensitivity study of checksum chunk size. We perform a sensitivity study on how the performance of Pavise varies when the checksum is computed based on variable-sized chunks of data on all workloads. When the chunk size setting is large, a small-sized update will trigger a checksum computation over the entire large chunk. This computation may be excessive and lead to higher overhead. On the other hand, a small chunk size will result in multiple checksums being updated if the update size is large and covers multiple chunks. Having a smaller chunk size also means the total number of checksums will be higher and will result in more space overhead to store the checksums. We empirically determine the optimal chunk size setting by testing out the performance of the workloads using different chunk size settings. Figure 16 shows how the average performance varies with different chunk sizes. The execution times are normalized to the ones when best-performing chunk size. Based on the results, in both cases where the chunk size is too small or too large, the performance becomes worse as predicted. On average, a chunk size of 512 bytes provides the best overall performance.

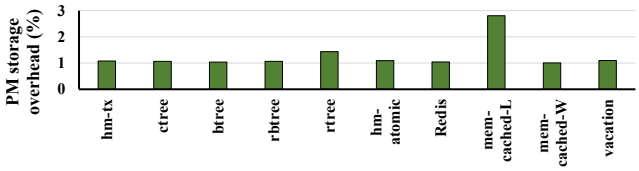

Figure 17: PM space overhead (% of total PM pool size) used by Pavise in all workloads.

Therefore, we have used that chunk size setting in all of our other evaluations.

6.2.7 PM Storage Overhead. Figure 17 shows the PM storage overhead of Pavise across all applications. The size of the metadata used by Pavise is negligible compared to the program's PM pool size and is omitted in our calculations. The main storage overhead comes from checksums and parities, where the parity storage is always inverse-proportional to the number of data-parity rows (N). We calculate the checksum space overhead by counting the number of checksums computed runtime and multiplying it with the checksum size. The space overhead of parity is always inversely proportional to the number of data-parity rows (N). As we set N to be 100 in our implementation, the parity space overhead is close to 1% of the PM pool size—consistent with our empirical study. Note that memcached-L has higher space overhead due to its smaller pool size, leading to a larger proportion of checksum storage.

7 RELATED WORK

Crash consistency in PM programs. Software PM libraries provide convenient programming interfaces for users to manage PM under DAX mode. The majority of these libraries utilize either undo logging or redo logging techniques to provide crash consistency. For example, Mnemosyne [69], NV-Heaps [7], and PMDK [26] use variations or combinations of redo and undo logging to provide crash-consistent transactions. MOD [17] uses functional shadowing to provide crash-consistent data structures. Atlas [4], NVThreads [19], and PMThreads [70] provide crash consistency by inferring failure-atomic regions from the semantics of multithreaded programs. While these libraries provide convenient interfaces for users to program a PM application and provide crash consistency, they do not provide fault tolerance, and thus are not resilient to data corruptions. Pavise can be directly applied on top of applications that use these libraries to provide fault tolerance, with minimal manual effort required.

Memory fault tolerance. Memory errors may occur for different reasons, such as defective cells and electromagnetic interference. There have been decades of studies on mitigating memory errors in both DRAM and PM systems, using techniques such as ECC [48, 76], Chipkill [9, 77], and wear-leveling [56]. These techniques can effectively mitigate errors stemming from the memory itself but do not protect errors in the higher-level system stack, such as CPU defects, software bugs, and firmware bugs. Therefore, a fault-tolerant software system like Pavise is necessary to protect PM programs against different sources of errors in the system stack.

Conventional fault-tolerant storage systems. Fault tolerance for storage systems is a well-studied area [14, 31, 46, 50, 63]. Data stored on traditional disk or flash devices (including PM, when it is accessed as a block device) can enjoy fault tolerance from the system [53, 57, 72, 79]. File systems specifically optimized for PM such as NOVA-Fortis [72] can provide fault tolerance to files backed by PM but cannot deal with accesses in DAX mode because accesses in DAX mode bypass these protections. Therefore, there is the need for a framework like Pavise that can react to DAX mode updates. System-level protections and Pavise are complementary.

PM fault tolerance. Prior work that aims to provide fault tolerance to DAX-mode PM applications considered both software [26, 35, 78] and hardware [34] approaches. While hardware solutions, e.g., Tvarak [34], provide attractive performance benefits, they require hardware modifications—not directly applicable to existing systems. Therefore, we keep such approaches out of the scope of this work. All that said, the key idea of co-designing crash-consistency and fault tolerance mechanisms from our work are also applicable to hardware solutions. Software solutions, such as Pangolin [78] and Vilamb [35], provide fault tolerance to PM applications by storing checksums and parity of data in PM. However, they either require significant manual efforts or do not provide strict protections and consistency guarantees. Hardware solutions such as Tvarak [34] also provide limited protection, as it only prevents errors visible at the hardware level (e.g., last-level cache). As discussed in Section 3.1, errors may also come from the software layer and CPU, which cannot be identified by the hardware at the cache, requiring higher-level software solutions to protect against. Pavise addresses the issues above by automating tracking with a compiler pass and coupling fault tolerance with crash consistency. Pavise can transparently provide fault tolerance to existing systems in a directly applicable manner.

8 CONCLUSION

In this work, we present Pavise, a framework that transparently provides fault tolerance for PM applications. Pavise does not impose restrictions on programming models and can be readily applied to existing workloads in real systems. Our evaluations show that Pavise can provide strong fault tolerance at a reasonable cost with minimal manual effort. Our evaluation shows that Pavise achieves 83.2% (with ignore-list) and 70.9% (with conservative tracking) performance of the state-of-the-art fault-tolerance software system, Pangolin. Because of a better coverage of persistent data, Pavise can sustain a higher error rate of 10^{-5} over Pangolin's 10^{-7}.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback and Lu Zhang for sharing Pangolin's artifact with us. We thank Zane Hu for his continuous support and feedback throughout the project. We acknowledge members of the EcoSystem research group for the help and feedback they provide. This work is supported in part by the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award, Facebook Faculty Research Award, Ontario Graduate Scholarship, and Vector Scholarship in AI.

REFERENCES

[1] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Trans. Storage*, 2008.

[2] Mary Baker, Mehul Shah, David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, TJ Giuli, and Prashanth Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of the 1st ACM SIGOPS/EuroSys*

European Conference on Computer Systems (EuroSys). Association for Computing Machinery, 2006.

[3] Y. Cai, Saugata Ghose, E. Haratsch, Y. Luo, and O. Mutlu. Errors in flash-memory-based solid-state drives: Analysis, mitigation, and recovery. *ArXiv*, 2017.

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

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

[6] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[9] Timothy J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory, 1997.

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

[11] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. arXiv, 2021.

[12] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014.

[13] Brad Fitzpatrick. Distributed caching with Memcached. *Linux J.*, 2004.

[14] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[15] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (ASPLOS)*, 2018.

[16] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[17] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[18] Peter H. Hochschild, Paul Jack Turner, Jeffrey C. Mogul, Rama Krishna Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. Cores that don't count. In *Proc. 18th Workshop on Hot Topics in Operating Systems (HotOS)*, 2021.

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

[20] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2012.

[21] Intel. Intel 64 and IA-32 architectures software developer's manual. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[22] Intel. Intel intelligent storage acceleration library. https://software.intel.com/content/www/us/en/develop/tools/isa-l.html.

[23] Intel. Intel optane dc persistent memory. https://www.intel.ca/content/www/ca/en/architecture-and-technology/optane-dc-persistent-memory.html.

[24] Intel. The libpmem library. https://pmem.io/pmdk/libpmem/.

[25] Intel. The libpmemobj library. https://pmem.io/pmdk/libpmemobj/.

[26] Intel. Pmdk. https://pmem.io/pmdk/.

[27] Intel. ipmctl. https://github.com/intel/ipmctl/issues/106, 2019.

[28] Intel. Redis, enhanced to use persistent memory. https://github.com/pmem/redis, 2019.

[29] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[30] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane dc persistent memory module. arXiv, 2019.

[31] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In *USENIX Annual Technical Conference (ATC)*, 2019.

[32] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure characteristics. In *6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[33] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[34] R. Kateja, N. Beckmann, and G. R. Ganger. TVARAK: Software-managed hardware offload for redundancy in direct-access NVM storage. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[35] Rajat Kateja, Andy Pavlo, and Gregory R. Ganger. Vilamb: Low overhead asynchronous redundancy for direct access NVM. arXiv, 2020.

[36] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[37] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[38] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[39] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO)*. IEEE Computer Society, 2004.

[40] Lenovo. memcached-pmem. https://github.com/lenovo/memcached-pmem, 2018.

[41] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. PMFuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[42] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[43] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[44] Jean loup Gailly and Mark Adler. zlib. https://zlib.net/.

[45] LWN. Add support for NV-DIMMs to ext4. https://lwn.net/Articles/613384/.

[46] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2015.

[47] Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[48] Prashant J. Nair, Chiachen Chou, Bipin Rajendran, and Moinuddin 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 (HPCA)*, 2015.

[49] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[50] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR)*, 2016.

[51] Ian Neal, Andrew Quinn, and Baris Kasikci. Hippocrates: Healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021.
[52] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.
[53] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1988.
[54] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.
[55] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*, 2005.
[56] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.
[57] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 2013.
[58] Andy Rudoff. Persistent memory on CXL. https://www.snia.org/educational-library/persistent-memory-cxl-2021, 2021.
[59] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.
[60] Salvatore Sanfilippo. Redis. https://redis.io/.
[61] Steve Scargall. Persistent memory architecture. *Programming Persistent Memory*, 2020.
[62] Stuart Schechter, Gabriel H. Loh, Karin Strauss, and Doug Burger. Use ecp, not ecc, for hard failures in resistive memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010.
[63] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
[64] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.
[65] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability (StorageSS)*. Association for Computing Machinery, 2005.
[66] SNIA. NVM programming model. https://www.snia.org/tech_activities/standards/curr_standards/npm.
[67] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.
[68] H. H. K. Tang, C. E. Murray, G. Fiorenza, K. P. Rodbell, M. S. Gordon, and D. F. Heidel. New simulation methodology for effects of radiation in semiconductor chip structures. *IBM Journal of Research and Development*, 2008.
[69] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
[70] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. PMThreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.
[71] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
[72] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.
[73] X. Xu and H. H. Huang. Exploring data-level error tolerance in high-performance solid-state drives. *IEEE Transactions on Reliability*, 2015.
[74] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. arXiv, 2019.
[75] Doe Hyun Yoon, Jichuan Chang, Robert S. Schreiber, and Norman P. Jouppi. Practical nonvolatile multilevel-cell phase change memory. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.
[76] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P. Jouppi, and Mattan Erez. FREE-p: Protecting non-volatile memory against both hard and soft errors. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.
[77] Da Zhang, Vilas Sridharan, and Xun Jian. Exploring and optimizing chipkill-correct for persistent memory based on high-density NVRAMs. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.
[78] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *USENIX Annual Technical Conference (ATC)*, 2019.
[79] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A ZFS case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.
[80] L. Zhu, J. Zhan, S. Chen, Y. Zhang, J. Yang, W. Jiang, and L. Li. Fault tolerant algorithm for NVM to reuse the error blocks. In *13th International Conference on Embedded Software and Systems (ICESS)*, 2016.

A ARTIFACT APPENDIX

A.1 Abstract

Pavies is implemented using LLVM 11.0.0, PMDK 1.10, and Intel ISA v2.29. The artifact includes the Pavise library, PMDK modified to work with Pavise, and various Pavise-enabled real-life workloads. The artifact also includes the scripts to reproduce the major runtime results in Figure 11.

A.2 Artifact check-list (meta-information)

- **Program:** Pavise
- **Compilation:** clang/clang++
- **Metrics:** Normalized runtimes using Pavise ignorelist, Pavise conservative tracking, and without Pavise.
- **Output:** CSV file containing the above runtime results.
- **Experiment:** Average runtimes normalized to original PMDK (Original Application), as shown in Figure 11.
- **Disk space required:** 75GiB
- **Recommended system memory:** 32 GiB+
- **Recommended CPU count:** 8+
- **Time needed to prepare workflow:** 0.5 hours
- **Time needed to complete experiments:** 1 hour
- **Publicly available:** Yes

A.3 Description

A.3.1 How to access. We maintain a GitHub repository for the artifact at https://github.com/hjjq/pavise-pact22-artifact.

Software dependencies. This artifact depends on the following environment.

- Ubuntu 18.04
- Linux kernel 5.4.0
- LLVM 11.0.0
- gcc-7.5
- Intel Pin-3.10
- PMDK 1.10
- Intel ISA v2.29

Data sets. We evaluated the following workloads.

- PMDK examples: hashmap-tx, ctree, btree, rbtree, rtree, and hashmap-atomic
- Real-life workloads: memcached-L, memcached-W, redis, vacation

A.4 Installation

This artifact has the following structure:

- `apps-no_pavise/`: Real-life application source files without Pavise.

- apps/: Real-life application source files with Pavise.
- build/: Generated Pavise object files and libraries.
- include/: Pavise header files.
- isa-l/: Intel ISA source files.
- llvm/: Contains ignore lists used by Pavise.
- pmdk-1.10-no_pavise/: Original PMDK with no Pavise.
- pmdk-1.10/: Pavise-enabled PMDK.
- results/: Generated Figure 11 results.
- src/: Pavise source files.
- repro.sh: Reproduces major Figure 11 results.
- setup.sh: Environment setups.

A.5 Experiment Workflow

Pavise artifact comes with a single script, repro.sh, to reproduce the major results. Note that repro.sh will automatically execute setup.sh to initialize the workflow environment. Since reproducing the results may take a considerable amount of time, it is recommended to execute repro.sh in a terminal multiplexer such as screen or tmux.

```
source repro.sh
```

A.5.1 Viewing Results. After repro.sh completes, the generated results are placed in results/. results/summary.csv summarizes the results of all experiments. Note that in Figure 11, all bars are plotted as relative runtime to the Original Appplication. results/summary.csv records raw runtime/throughput instead. For redis, memcached-L, and memcached-W), which measure throughputs, the relative runtime is computed using the inverse of the measured throughput.
E.g. relative runtime of Pavise-ignore-list redis = (Original Appplication throughput) / (Pavise-ignore-list throughput).

To view the results of each individual experiment, please see the rest of the files in results/:

- microbench_ignorelist.csv:
  Results corresponding to Pavise-ignore-list bars for hm-tx, ctree, btree, rbtree, rtree, and hm-atomic in Figure 11.
- microbench_conservative.csv:
  Results corresponding to Pavise-conservative bars for hm-tx, ctree, btree, rbtree, rtree, and hm-atomic.
- microbench_no_pavise.csv:
  Original Application bars for hm-tx, ctree, btree, rbtree, rtree, and hm-atomic.
- {memcached-L|memcached-W|redis|vacation}_ignorelist:
  Pavise-ignore-list bars for {memcached-L | memcached-W | redis | vacation}.
- {memcached-L|memcached-W|redis|vacation}_conservative:
  Pavise-conservative bars for {memcached-L | memcached-W | redis | vacation}
- {memcached-L|memcached-W|redis|vacation}_no_pavise:
  Original Application bars for {memcached-L | memcached-W | redis | vacation}