Mumak: Efficient and Black-Box Bug Detection for Persistent Memory

João Gonçalves
Instituto Superior Técnico
(ULisboa) / INESC-ID

Miguel Matos
Instituto Superior Técnico
(ULisboa) / INESC-ID

Rodrigo Rodrigues
Instituto Superior Técnico
(ULisboa) / INESC-ID

Abstract

The advent of Persistent Memory (PM) opens the door to novel application designs that explore its performance and durability benefits. However, there is no free lunch, and to program PM applications, developers need to be aware of potential inconsistent application state upon machine or application crashes. To overcome this difficulty, several tools have been proposed to detect the presence of the so-called crash-consistency bugs. While these are effective in detecting a variety of bugs, they present several key limitations, namely relying on application-specific semantics, requiring the programmer to manually annotate the program or modify the PM library, and relying on techniques with poor scalability, making them impractical for production code.

In this paper, we introduce Mumak, a tool that detects bugs in PM applications in an efficient and black-box manner. Our key insight to reduce the search space is to use a two-pronged approach with a first pass that is highly efficient by focusing only on key, error-prone code points without exhaustively testing all possible persistence orderings, and a second pass based on heuristics that try to compensate the shortcomings of the initial approach. Furthermore, we avoid application-specific knowledge or annotations by relying on the application's own recovery procedure as an (imperfect) consistency oracle. Our experimental results, with different applications and libraries, show that Mumak has bug coverage on par with the other state-of-the-art tools, while being up to 25× faster. We also found four new crash-consistency bugs, two in PMDK and two in Montage, three of which have already been acknowledged and fixed by the developers.

CCS Concepts: • **Hardware → Emerging technologies**; • **Software and its engineering → Software testing and debugging**.

Keywords: Persistent Memory, Bug Detection, Crash Consistency, Testing, Scalability

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

EuroSys '23, May 8–12, 2023, Rome, Italy
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9487-1/23/05.
https://doi.org/10.1145/3552326.3587447

ACM Reference Format:
João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. 2023. Mumak: Efficient and Black-Box Bug Detection for Persistent Memory. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3587447

1 Introduction

Persistent memory (PM) is a recent technology that promises to deliver performance in-between HDD/SSDs and DRAM, combined with data persistence guarantees across program and machine restarts. By acting as a mid tier between DRAM and SSD/HDD, PM opens the opportunity for programmers to explore novel designs to improve both raw and per-dollar performance [14, 21, 23, 24, 26, 28–30, 36, 39, 41, 49, 58–60].

Despite these opportunities, there are also limitations that can hinder PM adoption. Notably, upon crashes the application state can become inconsistent due to PM's limited support for failure-atomic writes. This means that developers need to take into account two considerations. First, the application must rely on carefully placed hardware constructs such as memory fences and cache flushing instructions, to ensure a crash-consistent snapshot in PM. To make this process less error-prone, namely when performing large update operations (e.g., updating complex data structures that require multiple operations), developers may rely on auxiliary techniques, such as write-ahead logging, consolidated long ago in areas such as databases or file systems [4, 18, 38, 44, 52, 57]. Second, upon restarting the application must run a recovery procedure to attempt to fix potential state inconsistencies.

Despite the use of such techniques, developing PM programs and libraries that fully exploit the performance potential of PM, and remain correct under crashes, is very challenging. In fact, it has been shown that several recently proposed PM programs and libraries are plagued with crash-consistency bugs [19, 29, 34]. To counter this state of affairs, developers might fall back on an extremely conservative approach (e.g., flushing and fencing more often than necessary), which may result in safer programs but substantially degrades performance, resulting in performance bugs.

In order to mitigate this problem, a growing body of research has been dedicated to developing tools and techniques to find crash-consistency bugs in PM programs [13, 15, 19, 22, 27, 29, 34, 35, 43]. However, and as we detail in §3, the

available state-of-the-art tools suffer from some key limitations. Some tools [13, 34, 35] require the developer to manually annotate the code, which is not only a time-consuming and error prone task, but it also requires expert knowledge about the PM semantics and underlying hardware guarantees, which is arguably not within the reach of most developers. In particular, PMTest [35] is very efficient, but it requires the programmer to heavily annotate both the underlying PM library and the target application to identify stores to PM or detect flush and fence instructions, among others. Alternatives such as XFDetector [34] and PMDebugger [13] mostly annotate the library itself; however, they still require manual annotations to detect some bugs specific to the application semantics, namely enforcing ordering constraints. Yat [27] automatically records all PM operations at execution time and later replays them in all permissible orderings without requiring programmer input. However, due to the very large search space, and because it relies on virtualization for recording and replay, Yat is extremely slow. Other tools achieve better scalability by guiding the space exploration and relying on model checking [19], symbolic execution [43] or application-specific semantics [15]. However, and as we will show in the evaluation, these approaches still scale poorly to large codebases.

Given this state of affairs, we argue that there is a pressing need for tools that are agnostic to both application and libraries, do not require manual user input and can analyze large applications in useful time – namely fast enough so that they could be integrated into existing development workflow pipelines. To fill this gap, we introduce Mumak, an efficient and black-box bug detection system for PM applications. Designing Mumak in a way that is practical for large codebases requires searching a very large space of possible failure points and persistence orderings. We address this challenge through a two-pronged approach that reduces the required instrumentation to a minimum: first, we rely on fault injection in key points while also avoiding exploring multiple combinations for the order in which persistent writes are applied; then, we use a trace analysis based on simple rules that try to identify the bugs that may evade the first pass. In addition, one of the key challenges faced by this class of tools is understanding, given the extremely large set of all possible persistent states, which are consistent from the standpoint of the application and which ones correspond to bugs – and to do so without relying on annotations or application-specific semantics. To address this, we rely on the observation that PM applications already come with a mechanism for differentiating valid and invalid states – the recovery procedure. When the recovery procedure is unable to fix inconsistencies and bring the application to a valid state, we flag this as a potential bug, thus building a reasonably reliable bug oracle without knowing the application semantics or relying on additional manual input such as annotations.

Our experimental evaluation of Mumak shows that, despite reducing the search space and relying on an imperfect oracle, it can detect bugs in variety of large, real-world applications using different libraries. Using only the application binary as input, Mumak detects 90% of the bugs found by the most complete tool in the state of the art [15] – which is dependent on precise key-value store semantics – in less than one tenth of the time and in a fully automated way. Furthermore, we demonstrate the generality of Mumak by analyzing Montage [58], a general system for buffered persistent data structures, which does not depend on PMDK, and finding two new crash-consistency bugs which lead to loss of data [55, 56]. Both were confirmed and fixed by the authors. In addition to this, Mumak also detected two new crash-consistency bugs in the latest version of PMDK, one of which has been confirmed as high-priority and already fixed by the maintainers [46, 47]. Finally, Mumak also has ergonomic concerns, by providing programmers with succinct bug reports and minimizing the impact on the existing development workflow.

Contributions. In summary, this paper makes the following contributions:

- The design and implementation of Mumak[1], a system that detects performance and crash-consistency bugs in PM programs in an efficient and black-box manner, which is key to allowing Mumak to be included in development pipelines, such as continuous integration approaches, thus contributing decisively towards bug-free and performant PM applications;
- A bug taxonomy that captures Intel-x86 PM semantics and is used to compare Mumak with the other state-of-the-art tools. This taxonomy is used to identify common patterns of misuse, which are used by Mumak for bug detection;
- A detailed evaluation of Mumak comparing it with other state-of-the-art tools and across libraries and versions, showing an order of magnitude better performance, and finding new bugs in large production codebases.

The rest of paper is organized as follows. §2 provides background on the persistency guarantees of x86 and defines a general taxonomy that captures easily identifiable patterns of PM misuse. §3 discusses the related work and compares it in light of this taxonomy. §4 introduces the design of Mumak and how it addresses the limitations of the state-of-the-art, and §5 discusses the implementation details. §6 evaluates Mumak, comparing it to state-of-the-art tools in a wide range of scenarios. Finally, §7 concludes the paper.

[1]https://github.com/task3r/mumak

2 Persistent Memory Semantics

There are many different non-volatile memory technology specifications, as well as processor architectures. For the purposes of this paper, the implementation and technical discussion focuses on Intel Optane DC memory, running on an Intel-x86 architecture. Nevertheless, this work is applicable to any persistent storage technology that shares the basic semantics of Optane, namely: i) writes not being immediately persisted (because they are buffered in a cache), and ii) the only way to guarantee this persistence is by calling an instruction that forces the data to be propagated to the persistent storage device. This is the case, for instance, for ARM's persistent memory technology and we believe that, in the future, other persistent memory technologies that appear are likely to share these semantics, as caches will always be an important instrument to ensure good performance.

Focusing the discussion on Intel Optane, it is possible to configure the memory module in two modes (which may operate concurrently). *Memory mode* uses PM as a volatile extension to main memory and leaves data placement and management to the memory controller; whereas *App Direct* mode exposes the memory module as persistent and byte-addressable memory and allows applications to explicitly control allocations and placements. In this paper, we only consider the latter, since using PM as cheaper volatile Memory [31, 49, 51] forgoes the need for crash-consistency requirements since the data is deemed volatile.

Furthermore, PM can be accessed either through a block device interface, such as a file system, as a regular disk, or through direct memory access using loads and stores as if it were main memory (DAX). In this paper, we focus on the latter since PM-enabled file systems already address many of the issues associated with crash-consistency [23, 59]. Note that accessing PM with DAX bypasses the kernel, and hence the techniques used to detect crash-consistency in PM-enabled file systems cannot generally be applied.

Upon a machine crash, PM is only able to provide failure atomicity for groups of 8 bytes, i.e., either all the updates in the group persist after the crash or none does. Therefore, auxiliary techniques, such as write-ahead logging or checksums [2, 17, 32, 40], must be employed to ensure the crash-consistency of larger updates. To ensure crash-consistency, all these techniques require that stores to PM are persisted in a specific order. Regardless of the techniques used, PM applications require a recovery procedure to be executed after a crash, which attempts to bring the application to a consistent state, either fixing it or flagging it as unrecoverable.

Different CPU architectures follow different persistency models. These can be classified as either strict or relaxed, and either buffered or unbuffered [48]. The first classification relates to the order in which writes are persisted relative to the order in which they become visible to other threads. Under strict persistency these orders match, whereas relaxed persistency removes this constraint. By allowing these orders to differ, and thus allowing persistent write instructions to be reordered by the hardware, relaxed persistency achieves better performance. However, it is also makes it impossible to predict the order in which stores are persisted (without additional constraints, as we will detail next) and thus makes it more difficult to reason about the crash-consistency guarantees of the code running on those architectures. The second property relates to the moment when the persistent writes occur. Under unbuffered persistency, persistent writes occur synchronously, meaning execution is stalled until these complete, which hinders performance. In contrast, buffered persistency allows for asynchrony, by queuing writes and proceeding with the execution.

The x86 architecture follows a relaxed, buffered persistency model [48]. In practice, when a store is performed, it is first queued in a store buffer and later reaches the volatile CPU caches, where it can remain indefinitely. For that store to be persisted, it needs to reach the Write Pending Queue (part of the PM memory controller). This can happen non-deterministically, as the cache evicts lines to load new data depending on a specific policy. However, this non-determinism can result in inconsistent states in case of a crash. In order for programmers to control the order in which stores are persisted, they need to employ special persistency instructions that enforce additional ordering constraints: flush instructions, which asynchronously write cache lines to memory, and, since flushes can be buffered, there are also fence instructions, which ensure the execution of buffered flushes and impose ordering guarantees between them. In particular, the x86 architecture offers `clflush`, `clflushopt`, `clwb`, `mfence` and `sfence` instructions. `clflush` persists a single cache line and cannot be reordered in relation to other stores. `clflushopt` and `clwb` persist a cache line but can be reordered until a fence instruction is executed. Moreover, `clflushopt` invalidates the cache line it acts upon, while `clwb` does not, allowing for better performance in certain workloads. Next, `mfence` imposes an order over buffered loads, stores, and flushes, while `sfence` orders only stores and flushes. Atomic updates such as *compare-and-swap* or *fetch-and-add*, commonly referred to as *read-modify-write* (RMW) instructions, also act as memory fences by flushing the store buffer in order to ensure their atomicity. Finally, x86 offers non-temporal stores, which bypass the cache entirely. However, these are still buffered and could be reordered, unless properly fenced. Throughout the paper, we will use the terms flush and fence when the difference in semantics is not relevant and use the x86 instruction name otherwise.

The improper use of the instructions discussed thus far, or its absence, can lead to a variety of bugs. To better guide the design choices of Mumak, we next present a taxonomy of PM bugs, divided into two main categories.

Correctness Bugs. Correctness, or crash-consistency bugs, cover situations where a crash might leave the PM in an inconsistent state. These bugs are the result of a violation of the durability and/or the ordering guarantees required by the application. *Durability bugs* are the simplest form of a crash-consistency bug. These include missing flush or fence instructions for a particular store, or relying solely on the non-deterministic cache eviction policies. But this does not mean that each store needs to be explicitly flushed. As flush instructions act on an entire cache line, in practice, and depending on the memory arrangement of a given application, a single flush can act on multiple stores. Additionally, applications may also exhibit dirty overwrites. These consist of persisting a store that overwrites previous stores to the same address that have not been persisted. In other words, writing to an address multiple times without persisting it, is a strong indication that the associated variable or data structure should be in volatile memory rather than in PM. Similarly to other works, we consider this a bug [13].

Correctness bugs also encompass *ordering bugs*. These are more complex to reason about, as they depend on the application semantics and the patterns used to recover after a crash. To define it generally, the ordering imposed over the persisted writes should be such that the application can successfully recover after a crash. A subset of these bugs can be defined as *atomicity bugs*, meaning that a set of stores should be performed atomically (at least from a logical standpoint). The hardware itself does not support these atomic transactions, although there are several software solutions that address this issue [2, 12, 17, 32].

Performance Bugs. Performance bugs do not affect the program's correctness, even in the presence of crashes, but result in degraded performance due to the excessive use of persistency instructions. The simplest forms of performance bugs are *redundant flushes* and *redundant fences*. A flush is redundant if the content of that address was not overwritten since it was most recently flushed. Additionally, a flush can also be considered superfluous if it acts on a volatile memory address, if the store was non-temporal (bypassing the cache), or if the memory alignment is such that multiple stores fit in the same cache line, thus requiring a single flush. A fence can also be considered redundant if there were no flush or non-temporal stores performed since the last fence instruction.

Finally, another example of a performance bug is the *use of PM to store transient data*. This can be the intended behavior, as existing works explore the use of PM as a larger and more affordable volatile memory for large-scale computing [31, 49, 51]. However, outside this scenario, this represents a misuse of PM as those accesses could be replaced by volatile memory, resulting in a performance improvement.

eADR. Enhanced Asynchronous DRAM Refresh (eADR) [6] extends the persistent domain up to the CPU caches, thus removing the need to perform cache line flushes. Yet, this does not guarantee crash-consistency by itself, since stores are persisted once globally visible, and thus fence operations need to be used to maintain store order correctness [53]. Additionally, extending the persistence domain to the CPU caches requires batteries (or an external uninterruptible power supply), which not only represent significant cost but also introduce additional maintenance given their shorter lifespan when compared to the other server components. In sum, it is unlikely that eADR will be used across all machines running PM applications, which means that applications should still tolerate the classic ADR domain [16, 53].

3 Related Work

In this section, we discuss the state-of-the-art in PM bug detection. We divide existing proposals in two major categories: those that rely on developer annotations, and those that perform automatic space exploration.

Annotation-based Debugging. Some proposals leverage manual code annotations to guide the bug detection process. These annotations can, for example, give hints about the semantic of the application (for instance, commit variables) or impose persistent ordering assertions. pmemcheck [7], which is part of PMDK, checks if both the library and applications built on top of it satisfy the expected persistence and order requirements. The library itself is extensively annotated, but user applications still require manual effort to complement the annotations according to the application semantics. PMTest [35] provides assert-like instructions that allow for assessing the safety of generic PM software (i.e., not necessarily developed with PMDK). It uses a record and replay mechanism decoupled from program execution. XFDetector [34] employs shadow memory, which traces and intercepts all PM accesses during the pre-failure phase, and, during post-failure executions, checks the persistency status of the addresses read by the application. Due to the execution of instrumented code for every failure point, both pre- and post-failure, as well as the management of the shadow memory, XFDetector is very slow. Moreover, detecting certain ordering bugs requires manual annotations. PMDebugger [13] builds upon a study, which concluded that, for most stores, data durability is guaranteed by the nearest fence. This means that most of the bookkeeping data is short-lived. While other tools [34, 35] organize all the information in a tree-like structure, PMDebugger initially stores the information in an array for quicker insertion. Later, when it encounters a fence, it clears persisted addresses and moves the remaining addresses to an AVL tree for the long-term benefit of quicker search. Yet, its efficiency is directly correlated to its dependence on pmemcheck's annotations, thus requiring annotations for any application that does not use PMDK. Moreover, it also requires annotations to detect ordering-based bugs even for applications built with PMDK.

Table 1. Tool classification according to the taxonomy presented in §2. The ✓* symbol denotes the need for manual annotations and ✓† denotes tools that detect the use of PM as transient data but do not distinguish it from durability bugs.

Tool	Bug Taxonomy						Application Agnostic	Library Agnostic
	Durability	Atomicity	Ordering	Redundant Flush	Redundant Fence	Transient Data		
pmemcheck	✓*			✓		✓†		
PMTest	✓*	✓*	✓*	✓				
XFDetector	✓*	✓*	✓*	✓	✓			
PMDebugger	✓	✓*	✓*	✓		✓†		
Yat		✓	✓				✓	
Jaaru	✓	✓	✓				✓	
Agamotto	✓	PMDK TXs		✓	✓	✓†	✓	
Witcher	✓	✓	✓	✓	✓			
Mumak (this work)	✓	✓	✓	✓	✓	✓	✓	✓

In general, annotation-based debugging shifts the complexity from the tool to the programmer. This entails a manual task that is prone to errors, and requires a substantial effort which might discourage adoption, especially for large production codebases.

Automatic Space Exploration. This class of tools automatically explores the state space and identifies crash-consistency violations. This is achieved through the use of either external crash-consistency checkers, employing the application's recovery as an oracle, defining custom application oracles, or inferring consistency oracles based on previous knowledge of the application's semantics. Yat [27] records PM instructions while the program is running and, later, replays the recorded stores in all permissible orderings and relies on a file system checker to verify if correctness is preserved. Because it relies on virtualization for recording and replaying all permissible orderings, Yat can be very slow. In fact, it is expected to require several years to provide 100% coverage in a program with a few thousand operations [27]. Jaaru [19] relies on model checking techniques to verify the crash-consistency guarantees of PM programs. It instruments memory and cache operations, simulating those instructions with full support for persistency semantics. This allows Jaaru to impose constraints on the possible values that a persistent variable can have in a post-failure execution depending on the last time that a cache line was flushed relative to the failure. In contrast to Yat, which eagerly enumerates all possible post-failure memory states, Jaaru uses a lazy approach that considers only the stores that were read by actual loads in the post-failure scenario. This allows Jaaru to reduce the search space for certain persistency patterns (namely the *commit store* pattern); however, it still results in an exponential search for many others. Agamotto [43] leverages symbolic execution (SE) to detect correctness and performance bugs in PM applications. SE is used to track the PM state across all different execution paths and then the tool relies on bug oracles to detect common patterns that might lead to anomalous behavior. The set of oracles provided by the authors is limited and needs to be extended by developers in order to detect other classes of bugs. Once more, this is a complex and time-consuming task, prone to human errors. Moreover, due to the nature of SE, the number of paths to explore can grow exponentially and therefore lead to impractical running times when analyzing large programs. However, it is important to note that Agamotto is capable of detecting a significant proportion of bugs in a reasonable amount of time, due to the exploration policy it employ, which prioritizes the paths that lead to PM accesses. Witcher [15] instruments memory operations to collect a trace of PM accesses in key-value store applications. It infers likely invariants from the collected trace and also through an analysis of the source code of the target application. These are based on patterns that include possible ordering and atomicity violations. Using deterministic test cases, it generates PM crash images that violate those likely invariants and applies output equivalence checking to determine whether they constitute a crash-consistency bug. This process has the upside of not reporting false positives. Nevertheless, it is an order of magnitude slower than other systems, and it is unclear how well the generation of deterministic test cases (which is not automatic) or output equivalence checking generalizes to applications with semantics different from key-value stores.

Other Systems. We now discuss some recent proposals that target related but different problems. PMFuzz [33] is a fuzzer that generates test cases and PM state images, mutating seed inputs based on a PM path coverage metric. It prioritizes inputs that result in executions that explore new code paths containing PM accesses. This is orthogonal to the actual bug detection, and thus PMFuzz can be used in conjunction with other systems and provide better bug coverage. RECIPE [29] presents a method for testing the crash-consistency of PM indexes by simulating faults for each 8-byte atomic store and testing the consistency by performing operations on top of the crash state, keeping track of the expected values for each key and finally checking if each key contains the correct value. The approach is efficient but also tailored to specific semantics (indexes), making it not generalizable. DURINN [16] and PMRace [3] target a subclass of ordering bugs in which concurrent reads of unpersisted

writes might lead to durable/observable side effects. PM-Race employs fuzzing and timer-based race detection, while DURINN infers potential bugs through trace analysis and constructs adversarial states and thread schedules to expose them. Yashme [20] focuses on a subclass of atomicity bugs in which compilers can implement a non-atomic store with multiple store instructions — store tearing — or generate new store instructions to store temporary values — store inventing – resulting in well-timed crashes to cause non-atomic stores to be made partially persistent. It employs model-checking with sampling, for scalability, and a constraint refinement mechanism similar to Jaaru [19], in order to extend the window in which these bugs are observable. DeepMC [50] uses static and dynamic analysis techniques, together with a set of predetermined rules, to check whether a given program respects some PM memory model such as strict, epoch or strand [45]. DeepMC works at a high abstraction level, following one of the models above, and hence can be very efficient, while Mumak relies only on the underlying hardware model. Moreover, DeepMC does not support programs with mixed memory models while Mumak is agnostic to the concrete model. Vinter [25] is a system to find bugs in full-systems, i.e., it considers a system as a whole including both user and kernel-space components. It relies on virtualization and dynamic binary translation to capture accesses to PM. Vinter focuses on PM file systems and, like Mumak, is fully automated. It is however unclear how it could be applied to PM applications that do not rely on a PM file system. Moreover, the overhead imposed by virtualizing a full system can become problematic when analysing systems that require large workloads (see §6.1). SafePM [1] is a PM memory safety mechanism to detect safety violations such as dangling pointers or buffers overflows. It relies on shadow memory techniques and AddressSanitizer [54]. Finally, Hippocrates [42] uses the output of PM bug finding tools to create bug fixes, using heuristics that automatically compute the effective location for interprocedural fixes, and doing so in a safe manner that guarantees the bug fixes do not introduce new bugs. Overall, these proposals target orthogonal problems to those we focus on this paper and hence we consider them complementary to Mumak.

Summary. Table 1 summarizes the most closely related work (corresponding to the first two paragraphs of this section) according to the taxonomy introduced in §2. The columns on the left side capture the types of bugs that each tool is able to uncover, and the last two columns indicate the generality of the tools, namely, which ones are agnostic to application and library semantics. This table highlights that Mumak is significantly more general than the state of the art, both in terms of range of bug types and independence from the application and library semantics. Furthermore, as §6 will highlight, it scales to large production codebases.

4 Mumak

In this section we describe the design of Mumak. Recall that our goal is to design a tool that is efficient, scalable to large codebases, and that treats the target application as a black-box, i.e., it does not require manual annotations or knowledge about the application semantics. These goals result in some tension as, in principle, application-specific knowledge or annotations could allow us to perform optimizations to reduce the search space, which our black-box approach foregoes.

We address this tension with a two pronged approach that comprises a fault injection and a trace analysis phase. The key intuition behind the fault injection phase is to execute the program, crash it at judiciously selected points in the execution, and run the recovery procedure. When the recovery procedure identifies an invalid state or fails to bring the program to a valid state, we are in the presence of a correctness bug. By using the recovery procedure as a correctness oracle, we forego the need for annotations and to know the application semantics, as these are implicitly encoded in the recovery procedure logic. Fault injection can be done very efficiently (see §4.1) allowing Mumak to be fast and scale to large codebases. However, this approach is not a silver bullet, as it can miss certain bugs, namely some instances of durability bugs and all performance bugs. Therefore, we complement fault injection with a trace analysis phase that identifies patterns of PM misuse and reports them (see §4.2).

The general Mumak pipeline is illustrated in Figure 1. The user must provide the application binary and a workload to drive the application. Similarly to all other tools, with the exception of Agamotto [43], Mumak requires a workload to test the application and exercise the different code paths. Therefore, bug coverage is limited by the coverage of the workload itself. It is possible to increase coverage by relying on automatic workload generators (such as PMFuzz [33]) but we consider this orthogonal to our contributions.

Given the application binary ①, Mumak automatically instruments it to generate the output required by the fault injection and trace analysis phases ②, after which it uses the user provided workload ③ and runs the instrumented application ④. This step generates two by-products: a failure point tree ⑤ and a PM access trace ⑥. Using the tree, Mumak executes the provided workload until reaching an unvisited failure point, marks it as visited in tree, injects the fault ⑦, recovers ⑧, and, if the recovery status determines so, reports the bug ⑨. Steps 6–8 are repeated in this order until all leaves in the tree are marked as visited. In parallel, Mumak analyzes the trace ⑩ and identifies patterns of misuse, generating bug reports or warnings ⑪. When both phases complete, Mumak combines the reports and presented them to the user.

4.1 Fault Injection

The main goal of this phase is to expose atomicity and ordering bugs by generating crash states in an efficient, automatic,

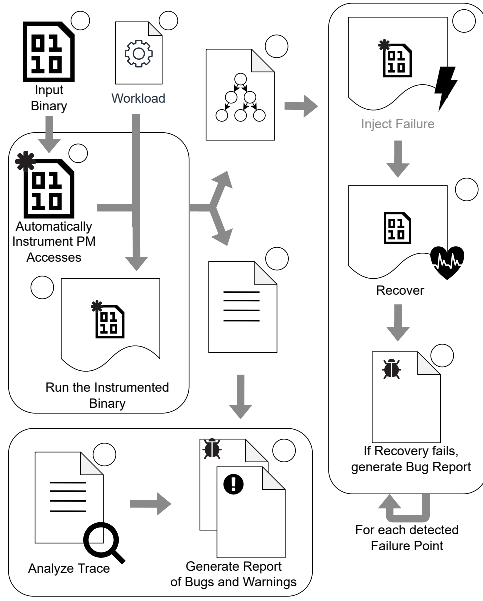


Figure 1. Mumak's analysis pipeline.

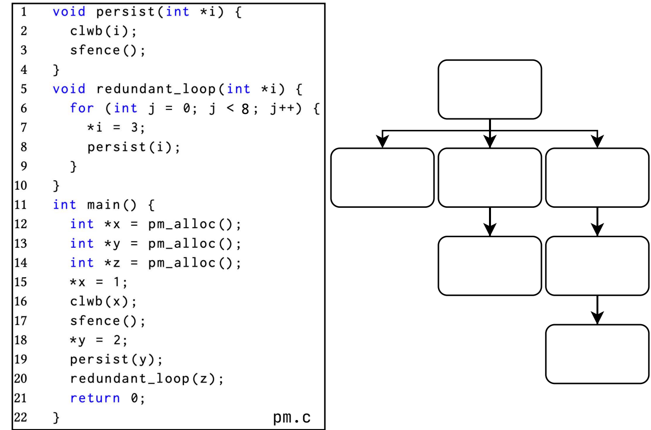


Figure 2. Sample program and corresponding failure point tree.

and reproducible manner. This raises several conceptual and practical challenges.

Conceptual Challenges. Designing Mumak's fault injection entails two main challenges: i) how to define the failure points, i.e., the points at which the application should crash, and ii) what should comprise the fault injection process itself.

We define a failure point as an instruction address in the original binary that we consider potentially prone to leaving the system in an inconsistent persistent state, if the system crashed at that point. This definition is intentionally generic and allows for further refinements, as we discuss later.

There are different policies to determine the set of failure points with different trade-offs. Approaches such as Yat and Jaaru [19, 27] opt to systematically check all possible post-failure states (taking into account the different valid orders for stores to be persisted) which do not scale, as shown in their respective evaluations. Consequently, we decided to only explore post failure states that respect program order for some prefix of the execution. This guarantees that our approach scales with the number of detected failure points, as each one will correspond to a single post-failure execution. In other words, a failure point materializes in a post-failure state from which the recovery procedure will attempt to recover. With that in mind, we can consider two different granularity levels for failure points: at the store level or at the persistency instruction level (flushes and fences). Considering each store as a failure point offers the best coverage of post-failure states, but can result in exploring many equivalent post-failure executions, depending on the recovery mechanisms employed. In contrast, only considering persistency instructions as failure points substantially reduces the number of faults injected. While this approach might seem too coarse, it is extremely efficient and covers all atomicity bugs and the vast majority of ordering bugs discovered by the state-of-the-art tools, as we show in §6.2. Additionally, we further reduce the number of failure points by only considering a persistency instruction if there was at least one store performed to PM since the last failure point, thus omitting equivalent post-failure states. In summary, the decision to only explore post-failure states that respect program order is key to the scalability of our approach, but also omits several possible combinations that do not respect this order, especially when developers impose few ordering constraints between stores. This is due to the fact that the persistence order does not necessarily respect program order, as detailed in §2. The trace analysis phase, which is further detailed in §4.2 addresses this limitation.

Regarding the fault injection process itself, the objective is to have a deterministic procedure to guarantee reproducibility. State-of-the-art solutions, such as XFDetector [34] or Jaaru [19], guarantee this by intercepting all PM accesses and controlling the values read by the application in the post-crash execution using some version of a shadow memory. However, this approach comes at a great instrumentation cost, which hinders scalability [34]. As such, we judiciously control the contents of the PM after each failure and execute *vanilla* recovery code (i.e., without any instrumentation). In detail, we crash the application gracefully, by killing the process after guaranteeing that pending stores are persisted before each failure point, as opposed to "pulling the power cord" and having to deal with non-deterministic post-failure states. This is also a key feature to ensure bug reproducibility, as we precisely control the contents of the PM after a crash.

Practical Challenges. Given the approach described above, a few challenges remain: i) how to automatically detect failure points in a black-box fashion, ii) how to guarantee that

faults are injected only at unique failure points, and iii) how to determine the consistency of a post-failure state.

We address the first challenge by treating the application binary as a black-box and considering only the instructions executed. As such, Mumak instruments the binary at the level of the assembly instructions by capturing their opcodes and respective arguments (when applicable). In detail, we capture stores, flushes and fences, as well as atomic instructions that have fence semantics. This allows Mumak to be agnostic of concrete application semantics and libraries.

For the second challenge, we want to guarantee that we inject faults at every failure point such that we explore all the code paths leading to that failure point. A naive approach that injects a fault only the first time the execution reaches a failure point (which corresponds to an assembly instruction) would miss many potential code paths that reach that failure point but have a different state. Conversely, an approach that injects a fault every time the execution reaches a failure point would explore many equivalent states and hence would not be efficient. Furthermore, the fact that developers often abstract the persistency calls (e.g., through the use of libraries) reinforces the point that both approaches have important shortcomings.

Our goal is therefore to find all unique code paths that reach a failure point. To reason about how this can be done efficiently, consider the code snippet of Figure 2, and assume we are injecting faults at the the persistency instruction granularity. This means that we must consider all unique code paths that reach line 2 and line 16, and ignore line 3 and line 17 since these generate equivalent post-failures states according to our granularity level. To achieve this, we start with a failure point tree, initially empty. During the execution of the application (step ④ of Figure 1), when the execution reaches a failure point, we collect the call stack and add it, address by address, to the tree, if not already there. Each node in the tree corresponds to an instruction address and each unique path from the root to a leaf corresponds to the call stack of a unique failure point. When the execution terminates, we have a tree composed of unique code paths (from the root to the leafs) leading to the failure points at the leafs. For the code snippet of Figure 2 the corresponding failure point tree is shown on the right side of the figure and it includes all the unique code paths leading to a persistency instruction that could originate different post-failure states.

Then, we execute the application (steps ⑦–⑨ of Figure 1) and when the execution reaches a failure point, we obtain the call stack and search it in the tree. If the corresponding leaf is unvisited, we mark it as visited and inject a fault, otherwise execution continues until all leaves are visited. Using a tree guarantees the uniqueness of each leaf and optimizes search and comparison. This is relevant since this approach performs many more comparisons than insertions. It is worth noting that this approach does not consider the parameters passed to each function, which in turn might skip potential failure points. This limitation stems from the need to achieve good scalability to large codebases.

Finally, to determine the consistency of the post-failure state, and as discussed above, we use the recovery procedure as an oracle. As such, we run the application in recovery mode without any instrumentation and, if it either fails or considers the state as unrecoverable, we report a bug and provide the path leading to that failure point. Additionally, if the recovery fails abruptly (e.g., with a segmentation fault), Mumak provides debug information for the recovery process including the recovery call trace that led to that failure. Note that this oracle is potentially imperfect. In particular, if the oracle fails to flag an inconsistent state then Mumak may incur in false negatives. However, it is in the best interest of the application developers for this procedure to be tailored to the application and its semantics, and for it to be as thorough as possible, as it increases the safety of the application in case of faults. The more complete the recovery is, the safer the application is and at the same time the fewer false negatives will occur in Mumak.

In summary, Mumak's fault injection phase has three main steps: i) automatically detect failure points and construct the failure point tree, ii) inject a fault, for each unique failure point and deterministically generate the corresponding post-failure state, and, iii) run the application in recovery mode and report bugs accordingly. Notably, each phase requires less instrumentation than the previous one, resulting in an efficient pipeline.

4.2 Trace Analysis

Despite reducing the search space for efficiency and scalability reasons, the fault injection approach described previously exposes the vast majority of the atomicity and ordering bugs that were found by previous approaches with much longer running times (see §6.2). The trade-off is that fault injection is unable to detect the remaining classes of bugs from our taxonomy (see §2). This stems from three main reasons. First, performance bugs do not originate an inconsistent state and hence cannot be detected by that approach. Second, some durability bugs might be masked by other correctly persisted operations. This is because durability bugs originate from the absence of proper persistency primitives, and since we only consider persistency primitives as failure points (in order to achieve better scalability), we cannot guarantee that the states that lack the proper durability are exposed. Finally, some ordering bugs are not observable if the post-failure state where they can be identified does not respect some prefix of program order.

We address these limitations of the fault injection phase with a trace analysis phase that efficiently detects the remaining classes of bugs. The approach is generally simple but differs from prior work [13, 34, 35] by being black-box and agnostic of specific libraries or application semantics. The trace analysis phase is composed of four stages, where

the first two are shared with the fault injection phase (Figure 1): i) automatic instrumentation of PM accesses (stores to PM, flushes, fences), ii) dynamic collection of a trace of PM accesses during the execution of the provided workload, iii) analysis of the trace by identifying possible patterns of PM misuse, and iv) report of those misuses accordingly.

Patterns. The trace analysis relies on a selection of well-defined patterns that can detect specific cases of PM misuse. For efficiency reasons, all patterns have been designed to require only a single pass through the trace. This fact, coupled with our black-box approach that foregoes the need for annotations or reliance on application semantics comes with the cost that, for some detected patterns, we cannot confidently state whether we are in the presence of a bug, as we detail below. We report those situations as warnings to guide the developer to reason about the intended semantics. Next, we present the set of patterns that are detected and how Mumak handles them.

► *Store instruction that is not explicitly persisted.* From our bug taxonomy, this might entail one of two bugs. Either this store should be explicitly persisted, or this is transient data that should be stored in volatile memory. Since it is impossible to provide a definitive answer without additional context information, we employ a simple rule: if the address in question is ever flushed during the execution, we report it as a durability bug, otherwise; we warn the developer for the potential use of PM to store transient data.

► *Flush instruction that acts on volatile address(es), or whose address(es) have not been written to the cache since the most recent flush.* In those cases, the flush instruction is not needed, and thus we report it as a bug.

► *Flush instruction that acts on more than one store.* If multiple stores fit in the same cache line, a single flush is enough, and therefore this is never a correctness bug. But in this case, subsequent flushes acting on the same stores, if any, are redundant and reported as such. However, this depends on the memory arrangement, which might change between platforms, compilers and compiler optimizations. As such, we warn the developer about a possible performance bug.

► *Fence instruction without pending flush or non-temporal stores.* In those instances the fence instruction is not needed, and thus we report it as a bug.

► *Fence instruction that acts on more than one* `clflushopt`*,* `clwb`*, or non-temporal store.* In these instances, the order in which those addresses are persisted is not deterministic. The fault injection phase already checks for possible inconsistencies when program order is respected, but that leaves out other possible combinations. These increase super linearly with the number of instructions on which the fence in question acts upon, and hence exploring all possible post-failure states is not scalable. Therefore, and with efficiency in mind, we warn the developer that, although we have not found a bug, we cannot guarantee that all possible orderings result in consistent post-failure states.

4.3 Discussion

Any tool to find bugs in PM applications needs to deal with a very large search space of failure points and persistence orderings. Existing approaches rely on manual annotations or application-specific semantics to reduce this space, but are prone to human errors or apply only to a subset of applications or libraries, respectively. Moreover, existing techniques scale poorly to large codebases, as they rely on heavy instrumentation and/or expensive program analysis.

We designed Mumak to be efficient and black-box. The key idea is to use a two-pronged approach that reduces the required instrumentation to a minimum. First, we rely on fault injection in key points in the program execution while intentionally avoiding the exploration of combinations of stores and persistency instructions that do not follow program order. Second, we rely on trace analysis to look for specific PM accesses that might result in bugs not identified in the fault injection phase.

Previous approaches [15, 16, 25] leverage their knowledge of the well-defined operations and semantics of their target applications to automatically find deviations from the expected post-failure state. However, by doing so, they limit their applicability to systems that follow those semantics. In contrast, Mumak's use of the application's own recovery procedure as a consistency oracle leads to a general approach that is independent of the underlying application semantics.

Regarding eADR systems [6], it is important to note that crash-consistency bugs can still occur, as the binary's instruction order might lead to inconsistent states and weakly-ordered non-temporal stores may still be reordered. For this reason, the atomicity and ordering bugs reported by Mumak's fault injection component would still be present in an eADR system. However, the trace analysis patterns would need to account for the different persistency semantics, otherwise they would wrongly report, for instance, durability bugs for stores that were not explicitly persisted.

Overall, and as we show in the evaluation, the design presented in this section results in a bug coverage that is on par with other state-of-the-art tools while being up to 25× faster. This is key to allowing Mumak to be included in development pipelines, such as continuous integration approaches, thus contributing decisively towards bug-free and performant PM applications.

5 Implementation

Mumak is implemented as a set of Intel Pin [37] tools, written in C++, and a Bash script that coordinates the analysis and acts as the frontend for the user. It is worth noting that, although Mumak's implementation uses Pin, our approach is

generic and can be applied to other instrumentation frameworks and methodologies. Pin uses dynamic compilation to instrument executables while they are running. This means that, in practice, steps ② and ④ from Figure 1 are performed simultaneously. However, they are conceptually independent and could execute as such using a different instrumentation framework. Finally, using Pin enables Mumak to analyze both the target applications and their dynamically linked dependencies. However, this implementation choice makes Mumak limited limited to user-space code.

When Pin instruments the application, it allocates memory in the address space of the application, which may cause application code, shared libraries, and dynamically allocated data to move. This raises a problem, if the memory allocations are not consistent between the phases of our pipeline, as we will detail next. Mumak creates a failure point tree using the call stack addresses for the failure points detected in the application. The tree is later serialized and stored in a file such that in a future fault injection execution it can be deserialized. For this approach to work, the addresses need to be the same during the tree construction and fault injection phases. However, because Pin dynamically allocates memory in the address space of the application, deserializing the tree causes shifts in the addresses of the application, which in turn would lead the same instruction to have a different address in each phase. To address this, we preemptively allocate memory for the tree before instrumenting, ensuring that the offset is the same in both phases and thus the same execution paths are assigned to the same addresses. This means that the pre-allocated memory (a configurable parameter) needs to be large enough to fit the entire tree.

Another important benefit of using Pin is that it offers an API to obtain a backtrace that filters out the calls made to instrumentation routines, thus showing only the relevant addresses that correspond to calls made by the application under analysis. This facilitates the debugging process when compared with tools that use LLVM, which produce very verbose outputs (see §6.5).

Additionally, to increase the determinism of applications and enhance the reproducibility of our fault injection process, we instrument non-deterministic calls (e.g., random number generators) and replace them with deterministic outputs.

In the early stages of the implementation, we observed that calls to `PIN_Backtrace` represented a significant portion of our execution time. As such, we optimized the tracing by only collecting the type of instruction, its argument(s) (in the case of flushes and stores), and an instruction counter (a monotonically increasing counter that uniquely identifies each traced instruction). This information is sufficient to identify bugs and their type during trace analysis. However, this approach requires an additional step to obtain the debug information. For that, we execute the target application once more using minimal instrumentation to collect the relevant backtraces according to the instruction counter. Unfortunately, due to the use of the instruction counter, this optimization depends on the determinism of the application. To analyze nondeterministic applications, we disable it.

6 Evaluation

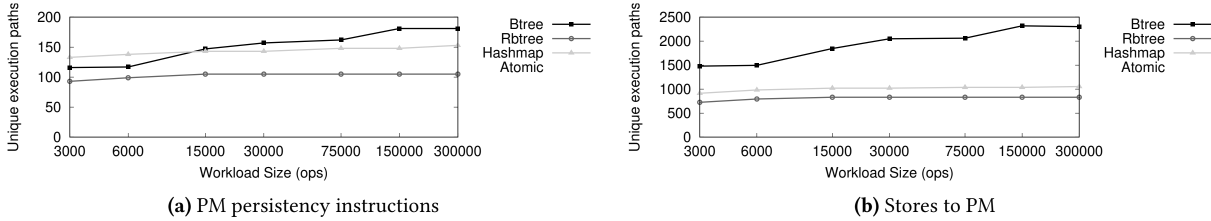
In this section we evaluate Mumak along several dimensions and compare it with other state-of-the-art tools.

Evaluation Settings. We evaluated Mumak on a 128 core Intel(R) Xeon(R) Gold 6338N CPU @ 2.20GHz, with 256 GB of RAM, and 1 TB Intel DCPMM in App Direct mode. As for software, we used Ubuntu 22.04, Linux kernel 5.15, Intel Pin 3.14, and Docker 20.10. The use of Docker allows us to have an unmodified host environment and keep tool-specific configurations inside the respective container images. To avoid the overhead of Docker's layered filesystem, we rely on volumes to store the outputs.

6.1 Performance Benchmarks

Baselines. We compare Mumak with four state-of-the-art solutions for PM bug detection. We selected Agamotto [43], XFDetector [34], PMDebugger [13], and Witcher [15] as these systems represent recent advances that use a variety of different approaches to tackle the problem at hand, namely symbolic execution, fault-injection and trace analysis.

Targets and Workloads. Like previous works, we used a set of PMDK's `libpmemobj` example applications that implement different data structures in PM, namely Btree, Rbtree and Hashmap Atomic. Since all prior approaches, with the exception of Agamotto, require a workload to drive the exploration, we conducted a preliminary study to determine the workload size required to provide sufficient coverage. To this end, we determine the number of unique execution paths that lead to persistency instructions (flushes and fences) and stores to PM. The results, for a workload with an equal number of puts, gets and deletes, are depicted in Figure 3a and Figure 3b for persistency instructions and stores, respectively. We observe that smaller workloads exercise few unique paths and therefore we need larger workloads to have good code coverage and, consequently, bug coverage. Based on these results, we evaluate the systems with workloads consisting of 150 000 operations, equally distributed among puts, gets and deletes. Another observation is that the number of unique code paths when considering stores to PM (Figure 3b) is roughly one order of magnitude larger than when considering persistency instructions (Figure 3a). This supports our decision of targeting the latter for fault injection, as otherwise the very large search space would prevent our approach from scaling. Still, as we show later, this is enough to detect most bugs identified by state-of-the-art tools. Finally, we note that the original applications perform all put operations inside a single PMDK transaction. Some systems [15, 34] changed this behavior by performing each



(a) PM persistency instructions

(b) Stores to PM

Figure 3. PMDK data store coverage based on workload size. Note that the x scale is not linear.

put inside a different transaction. This has an impact on the bug detection performance and, as such, we consider both alternatives, annotating the latter with "SPT", for "single put per transaction". In the case of Witcher and XFDetector, we only consider the latter alternative since their analysis depends on such behavior and/or annotations.

PMDK Versions. We perform the evaluation across two PMDK versions, namely 1.6 and 1.8. This stems from the fact that two of the systems, XFDetector and PMDebugger, require not only annotations to the application code but also to the underlying library. Therefore, evaluating all systems in the same PMDK version would not only require a substantial manual effort of porting the PMDK changes and application annotations to a specific PMDK version – one of the key limitations of annotation-based approaches – but could also lead to the inadvertent introduction of incorrect modifications to PMDK or the applications leading to the incorrect reporting of bugs not flagged in the original works. Hence, we compare Mumak with the other tools using the PMDK version used in their respective papers, namely: PMDK 1.6 for XFDetector and Agamotto, and PMDK 1.8 for PMDebugger and Witcher. In sum, this guarantees a fair comparison and highlights Mumak's agnostic design and implementation.

Metrics. We use the following metrics: analysis time (the total execution time of the tool), average CPU load, and peak memory overhead (volatile and persistent memory overhead relative to peak usage during vanilla execution).

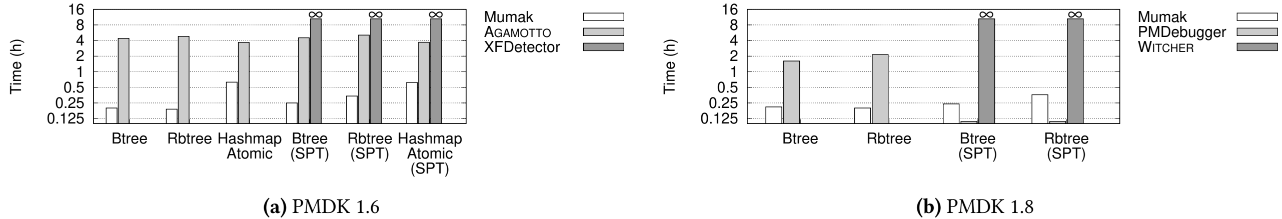
Results. Total analysis times are shown in Figure 4a and Figure 4b, for PMDK 1.6 and PMDK 1.8, respectively. Note that Hashmap Atomic does not work correctly with PMDK 1.8 and hence we exclude it from that version. We restrict the analysis time to 12 hours (represented as ∞ in the figures) since we believe this to be a reasonable maximum time for a tool to assess the application for the presence of bugs.

The main observation is that Mumak is substantially faster than all other approaches, in all but one case. In fact, Mumak took significantly less than ~1 hour to analyze each application. In more detail, XFDetector and Witcher did not complete the analysis within the 12 hours period for any of the applications. XFDetector takes a long time to analyze each operation (the original paper acknowledges it takes 40.6 seconds to analyze a single insert operation [34]) and hence, for large workloads such as the ones we used, it would take over 1000 hours to finish (on estimate). Witcher also failed to reach the scale of the workloads selected in our evaluation, despite aggressively parallelizing its analysis. In fact, this aggressive parallelization justifies some of the results, as we will discuss shortly. Agamotto required considerably more time than Mumak to finish its symbolic execution. However, we point out that, thanks to its search heuristic that prioritizes PM accesses, it detects a significant portion of the bugs within the first hour of the analysis. Finally, PMDebugger shows somewhat surprising results: it takes considerably longer than Mumak to analyze the original applications (close to 10×), but it takes only a couple of minutes to analyze the SPT variant. This is due to the annotations, which break the bookkeeping into segments corresponding to each transaction, hence, shorter transactions lead to less bookkeeping and faster analysis.

In terms of resource usage, Table 2 presents the peak CPU load and memory overheads for all experiments. Witcher is a clear outlier, as it tries to parallelize its analysis across all available cores. However, by not taking into account the memory required to do so, it exhausted the available memory (256GB). This helps justify why it was unable to analyze the target applications inside the predefined 12 hour timeout. It is worth noting that this behavior is not configurable and could only be altered by modifying the tool's code. Aside from Witcher, PMDebugger also consumes significantly more memory than the remaining tools. This is closely related to the trace analysis approach and the amount of information that it needs to bookkeep. Next up, Agamotto consumes 3.8 – 5.8× more memory than the target application requires, which is to be expected since symbolic execution is known to be resource intensive. XFDetector, which follows a fault injection-based approach, has the lowest volatile memory requirements of all tools. It is also the only tool that relies on PM to store analysis metadata. Although in these experiments this represents about 1 GB, if the trend is consistent across applications (close to 2× overhead), it could bottleneck the analysis of applications that use large amounts of PM. Overall, Mumak requires the least resources and performs the fastest analysis of the tools in this comparison.

Table 2. Average CPU load, peak RAM and PM overheads relative to vanilla executions. AGAMOTTO does not execute the applications and thus does not use PM. Hashmap Atomic does not operate correctly in PMDK 1.8.

PMDK	Tool	Hashmap Atomic			Btree			Rbtree			Hashmap Atomic (SPT)			Btree (SPT)			Rbtree (SPT)		
		CPU	RAM	PM	CPU	RAM	PM	CPU	RAM	PM	CPU	RAM	PM	CPU	RAM	PM	CPU	RAM	PM
1.6	**Mumak** (this work)	1.25	2.5×	1×	1.20	2.5×	1×	1.23	2.4×	1×	1.44	1.5×	1×	1.42	2.5×	1×	1.38	2.4×	1×
	XFDetector	—	—	—	—	—	—	—	—	—	1.03	1.5×	1.9×	1.03	1.6×	1.9×	1.03	1.6×	1.9×
	AGAMOTTO	1.56	3.8×	—	1.56	4.9×	—	1.56	4.8×	—	1.56	3.8×	—	1.56	5.8×	—	1.56	5.8×	—
1.8	**Mumak** (this work)	—	—	—	1.22	2.5×	1×	1.23	2.4×	1×	—	—	—	1.20	2.5×	1×	1.28	2.5×	1×
	PMDebugger	—	—	—	1.35	8.9×	1×	1.29	8.7×	1×	—	—	—	1.07	8.9×	1×	1.10	9.0×	1×
	WITCHER	—	—	—	—	—	—	—	—	—	—	—	—	138	232×	—	148	232×	—



(a) PMDK 1.6

(b) PMDK 1.8

Figure 4. Analysis time of `libpmemobj` benchmarks. ∞ denotes instances where the time needed to analyze the target application exceeded the defined 12 hour limit.

6.2 Coverage

In this section, we show how Mumak's design choices impact bug coverage. We used WITCHER's bug list as a baseline, since it is the most complete and most recently published, and it covers several applications, namely PMDK's data stores, RECIPE [29], Redis [10], WORT [28], Level Hashing [60], FAST&FAIR [21], and CCEH [41], comprising a total of 43 correctness bugs and 101 performance bugs. Overall, Mumak detects 90% of the bugs (and as shown earlier, does so in significantly less time than WITCHER) while being agnostic to library and application semantics, unlike WITCHER. The interpretation of these results should take the characteristics of our analysis into account. All correctness bugs detected by Mumak are found by the fault injection component. The trace analysis component detects performance bugs and potential correctness bugs, which we report as warnings and do not consider in our reported coverage. Finally, we find all the performance bugs reported by the state of the art, hence, the percentage of bugs we do not find are correctness bugs.

Moreover, Mumak achieves this coverage without reporting false positives. The reason is twofold. First, in the fault injection phase, Mumak generates crash states that the application can reach (i.e., possible states) but from which it cannot recover. Necessarily, this behavior represents a bug. Second, in the trace analysis phase, the heuristics employed define patterns of misuse that always correspond to real bugs. In other words, instances where a store is not flushed/fenced, or a flush/fence is redundant, cannot represent a false positive. The exceptions are the warnings, but those were not considered as positives when measuring coverage.

Interestingly, we observed that for Level Hashing Mumak failed to detect all but one bug out of the 17 reported by WITCHER. After further investigation, we pinpointed the cause to the lack of a recovery procedure in Level Hashing, which compromised our oracle. Therefore, we decided to implement a recovery procedure. This required less than 20 lines of simple C code to traverse the structure, count the reachable items and compare the result with the counters that were already maintained and persisted in the original code. This small change increased Mumak's bug coverage to 90% and highlights that our requirements regarding the recovery procedure can be easily met, but also serves as a motivation for developers to invest in the recovery code, not only for making the application robust in the presence of unexpected failiures, but also to aid testing tools that use the recovery procedure to help in finding bugs.

6.3 Scalability

We now study Mumak's scalability by analyzing larger codebases, with the goal of showing that Mumak's analysis time is not proportional to the size of the codebase under test. To that end, we selected two hashtable implementations from Montage [58] (`LfHashtable` and `Hashtable`), two persistent engines from pmemkv [9] (`cmap` and `stree`), and PM-aware implementations of Redis [10] and RocksDB [11]. As before, we performed a preliminary experiment to determine the size of the workload required to achieve sufficient coverage (omitted due to space constraints) and once more opted to perform 150 000 operations equally split among puts, gets and deletes. Figure 5 presents the analysis time in relation to the size of the codebase. We measured the size of the codebase as the number of lines ending in a semicolon for the target and their PM dependencies (for example, PMDK). We can conclude that Mumak's analysis has good scalability, as

Table 3. Qualitative analysis of the output and ease-of-use of state-of-the-art PM bug detection tools.

Tool	Presents complete bug path	Filters unique bugs	Runs any generic workload	Changes target code	Changes build process
XFDetector	No	No	**Yes**	Yes	Yes
PMDebugger	**Yes**	No	**Yes**	Yes	**No***
Agamotto	**Yes**	**Yes**	No (Symbolic Execution)	**No**	Yes
Witcher	No	No	No	Yes	Yes
Mumak (this work)	**Yes**	**Yes**	**Yes**	**No**	**No**

* PMDebugger uses pmemcheck's annotations, which are part of PMDK. This means that these annotations are available to developers that use the library. However, it also means that PMDebugger cannot be used by applications that are not built on top of PMDK.

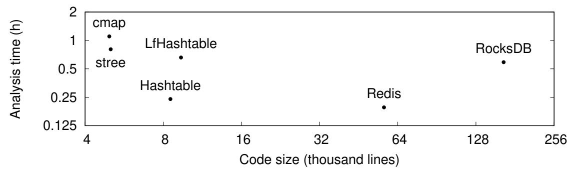


Figure 5. Mumak analysis time relative to code size.

the time required to analyze each target is not correlated with its code size. Mumak's fast analysis time allows it to be included in development pipelines, such as continuous integration approaches, hence contributing towards bug-free and performant PM applications.

6.4 New Bugs

Mumak discovered two crash-consistency bugs in Montage, which lead to data loss or corruption. The first one stemmed from the incorrect use of their persistent allocator, which broke the recoverability of the structures built on top of it. The second bug originated from a crash in a much narrower window during the destruction of the allocator object, once again potentially corrupting the structure's data. Both bugs were confirmed and fixed by the authors [55, 56].

We also analyzed the latest release of PMDK, at the time of writing, version 1.12.0 [8]. We found a bug by running the Btree data store workload described in §6.1. Interestingly, this is not a bug of the data store itself but of the actual library. In detail, this bug is triggered by a fault injected when committing a large transaction, which in turn leads to the possibility of a subsequent (post-failure) large transaction that needs to dynamically allocate extra undo log space to also crash the application. This bug was only exposed when performing a large number of operations, reinforcing the need for large workloads to provide sufficient coverage. This issue was classified as "high priority" and has since been resolved [47]. We also discovered a crash-consistency bug in the ART data structure. In particular, a fault injected during the commit of an insert operation leaves the tree in an inconsistent state, leading a post-crash insertion to fail an assertion as it tries to allocate too many children to the same node. This issue was confirmed but not yet resolved, at the time of writing [46].

In sum, Mumak is able to detect new bugs due to two main reasons. First, many previous works focus on PMDK and make assumptions based on the use of that library, limiting their applicability when the target applications do not use it. This contrasts with Mumak's black-box approach that enabled it to analyze Montage, which has its own PM allocator and does not leverage PMDK. Second, some bugs are only triggered when the target application is submitted to large workloads, and Mumak is much better equipped than the rest of the state-of-the-art to handle them within an acceptable timeframe (as shown in §6.1).

6.5 Ergonomics

This section provides a qualitative analysis of the output and ease-of-use of each tool. Table 3 summarizes the results, according to the following criteria that specify whether the system: i) provides a stack trace detailing the code path taken to reach the bug, ii) filters unique bugs, i.e., does not report multiple instances of the same bug, iii) can use a generic workload or if, instead, it requires a specially crafted workload, iv) requires changes to the application code, and v) requires changes to the build process.

XFDetector reports the bugs it detects without providing sufficient information to pinpoint their specific cause, and it does not filter duplicates. For example, if a bug is detected in a given annotation, it simply reports the line of that annotation. Besides requiring developer annotations, XFDetector requires the post-failure execution to terminate without errors, otherwise, it reaches an infinite loop. In addition to this, the publicly available artifact does not offer an intuitive interface to analyze generic applications, requiring users to script the analysis themselves.

PMDebugger reports enough information to pinpoint the root cause of bugs, but it also reports all occurrences of every bug, leading to a large amount of redundant information.

Agamotto provides complete paths and removes duplicate bugs from its output. However, it also introduces noise due to the KLEE's symbolic execution engine, along with references to LLVM's bitcode. Moreover, to use Agamotto, developers need to convert the target application to a single-file LLVM bitcode. Besides this, the oracles provided are generic yet limited (Table 1). The authors instruct developers to create application-specific oracles to achieve better

coverage. Once again, this is a complex and time-consuming task, prone to human errors.

Finally, according to the selected criteria, Witcher has the worst ergonomics from the tools used in this comparison. It generates large amounts of output (4 – 5GB in the experiments performed) without a clear summary of the bugs discovered nor instructions on how to analyze the results. In addition to this, Witcher requires developers to implement a driver (similarly to YCSB [5]) that interacts with their target application. This is a time-consuming task and restricts the tool to key-value store applications.

Mumak provides a stack trace of the code path leading to the bug, only reports unique bugs, supports any workload, does not requires changes to the application's code or build process, and, with the exception of warnings (that can be disabled), does not report false positives.

7 Conclusion

We presented Mumak, a tool for detecting correctness and performance bugs in PM applications. Our design detects 90% of the bugs reported by Witcher, the most recent and complete state-of-the art tool, while being up to 25× faster than Witcher, Agamotto, XFDetector and PMDebugger, in most scenarios. Unlike other tools, this is achieved without relying on manual annotations, specific libraries, or application semantics. Mumak also detected new bugs in complex software, namely the latest version of PMDK and Montage.

The efficient and black-box nature of Mumak, together with its ergonomic concerns, makes it amenable to be integrated in existing continuous integration pipelines, which, we believe, is a crucial step towards bug-free and performant PM applications.

Acknowledgments

We thank our shepherd, Baptiste Lepers, and the anonymous reviewers for their feedback. We also thank João Margaço and Shady Issa for the exploratory work, and the authors of Montage for their help confirming and fixing bugs. This work was supported by Fundação para a Ciência e a Tecnologia (FCT) under grants 2021.07401.BD, UIDB/50021/2020, PTDC/CCI-COM/4485/2021 (Ainur), PTDC/CCI-INF/6762/2020 (MS3).

References

[1] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. SafePM: A sanitizer for persistent memory. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys'22, page 506–524. ACM, 2022. doi:10.1145/3492321.3519574.

[2] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. In *IEEE International Parallel and Distributed Processing Symposium*, IPDPS'18, pages 368–377, 2018. doi:10.1109/IPDPS.2018.00046.

[3] Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. Efficiently detecting concurrency bugs in persistent memory programs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'22, page 873–887. ACM, 2022. doi:10.1145/3503222.3507755.

[4] 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. *SIGARCH Computure Architure News*, 39(1):105–118, 2011. doi:10.1145/1961295.1950380.

[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154, 2010.

[6] Intel Corporation. eADR: New opportunities for persistent memory applications. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[7] Intel Corporation. Enhanced valgrind for persistent memory. URL: https://github.com/pmem/valgrind.

[8] Intel Corporation. Pmdk version 1.12.0. URL: https://github.com/pmem/pmdk/releases/tag/1.12.0.

[9] Intel Corporation. pmem/pmemkv: Key/value datastore for persistent memory. URL: https://github.com/pmem/pmemkv.

[10] Intel Corporation. pmem/redis: Redis adapted to use persistent memory. URL: https://github.com/pmem/redis.

[11] Intel Corporation. pmem/rocksdb: A version of rocksdb that uses persistent memory. URL: https://github.com/pmem/redis.

[12] Intel Corporation. Persistent memory development kit, 2022. URL: https://pmem.io/pmdk/.

[13] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings ofthe 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'21, pages 503–516, 2021. doi:10.1145/3445814.3446744.

[14] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16. ACM, 2016. doi:10.1145/2901318.2901344.

[15] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP'21, page 100–115. ACM, 2021. doi:10.1145/3477132.3483556.

[16] Xinwei Fu, Dongyoon Lee, and Changwoo Min. DURINN: Adversarial memory and thread interleaving for detecting durable linearizability bugs. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'22, pages 195–211. USENIX Association, 2022. URL: https://www.usenix.org/conference/osdi22/presentation/fu.

[17] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'20, page 59–74. ACM, 2020. doi:10.1145/3385412.3385991.

[18] E. R. Giles, K. Doshi, and P. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *31st Symposium on Mass Storage Systems and Technologies*, MSST'15, pages 1–14, 2015. doi:10.1109/MSST.2015.7208276.

[19] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru : Efficiently Model Checking Persistent Memory Programs. In *Proceedings ofthe 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'21, pages 879–892. ACM, 2021. URL: https://doi.org/10.1145/3445814.3446735, doi:10.1145/3445814.

[20] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Yashme: Detecting persistency races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'22, page 830–845. ACM, 2022. doi:10.1145/3503222.3507766.

[21] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in Byte-Addressable persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 187–200. USENIX Association, 2018. URL: https://www.usenix.org/conference/fast18/presentation/hwang.

[22] Louis Jenkins and M. Scott. Persistent memory analysis tool (pmat). 2019.

[23] 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'19, page 494–508. ACM, 2019. doi:10.1145/3341301.3359631.

[24] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies*, FAST'19, pages 191–205. USENIX Association, 2019. URL: https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet.

[25] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference*, ATC'22, pages 933–950. USENIX Association, 2022. URL: https://www.usenix.org/conference/atc22/presentation/werling.

[26] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, page 460–477. ACM, 2017. doi:10.1145/3132747.3132770.

[27] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ATC'14, page 433–438. USENIX Association, 2014. URL: https://dl.acm.org/doi/10.5555/2643634.2643678.

[28] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 257–270. USENIX Association, 2017. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon.

[29] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting Concurrent DRAM Indexes to Persistent- Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, page 462–477. ACM, 2019. doi:10.1145/3341301.3359635.

[30] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *20th USENIX Conference on File and Storage Technologies*, FAST'22, pages 35–50. USENIX Association, 2022. URL: https://www.usenix.org/conference/fast22/presentation/li.

[31] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. Hierarchical hybrid memory management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2223–2236, 2019. doi:10.1109/TPDS.2019.2908175.

[32] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. *SIGPLAN Not.*, 52(4):329–343, 2017. doi:10.1145/3093336.3037714.

[33] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. PMFuzz: test case generation for persistent memory programs. In *Proceedings ofthe 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'21, pages 487–502. ACM, 2021. doi:10.1145/3445814.3446691.

[34] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings ofthe 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 1187–1202, 2020. doi:10.1145/3373376.3378452.

[35] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings ofthe 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, pages 411–425, 2019. doi:10.1145/3297858.3304015.

[36] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.*, 13(8):1147–1161, 2020. doi:10.14778/3389133.3389134.

[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005. doi:10.1145/1064978.1065034.

[38] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker . Rethinking main memory oltp recovery. In *IEEE 30th International Conference on Data Engineering*, pages 604–615, 2014. doi:10.1109/ICDE.2014.6816685.

[39] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to Byte-Addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage'17. USENIX Association, 2017. URL: https://www.usenix.org/conference/hotstorage17/program/presentation/marathe.

[40] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS'13. ACM, 2013. doi:10.1145/2524211.2524216.

[41] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies*, FAST'19, pages 31–44. USENIX Association, 2019. URL: https://www.usenix.org/conference/fast19/presentation/nam.

[42] 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'21, page 401–414. ACM, 2021. doi:10.1145/3445814.3446694.

[43] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 1047–1064, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/zhang-quanlu.

[44] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *10th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage'18. USENIX Association, 2018. URL: https://www.usenix.org/conference/hotstorage18/presentation/ni.

[45] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA'14, page 265–276. IEEE Press, 2014.

[46] pmem/pmdk. Crash-consistency bug within libart. URL: https://github.com/pmem/pmdk/issues/5512.

[47] pmem/pmdk. Crash-consistency bug within pmemobj_tx_commit. URL: https://github.com/pmem/pmdk/issues/5461.

[48] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the intel-x86 architecture. In *Proceedings of the ACM on Programming Languages*, volume 4 of *PACMPL'20*, 2020. doi:10.1145/3371079.

[49] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. *HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM*, page 392–407. SOSP'21. ACM, 2021. URL: https://doi.org/10.1145/3477132.3483550.

[50] Benjamin Reidys and Jian Huang. Understanding and detecting deep memory persistency bugs in nvm programs with deepmc. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'22, page 322–336. ACM, 2022. doi:10.1145/3503221.3508427.

[51] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. Optimizing large-scale plasma simulations on persistent memory-based heterogeneous memory with effective data placement across memory hierarchy. In *Proceedings of the ACM International Conference on Supercomputing*, ICS'21, page 203–214. ACM, 2021. doi:10.1145/3447818.3460356.

[52] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO'15, page 672–685. ACM, 2015. doi:10.1145/2830772.2830802.

[53] Steve Scargall. *Persistent Memory Architecture*, pages 11–30. Apress, 2020. doi:10.1007/978-1-4842-4932-1_2.

[54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ATC'12, page 28. USENIX Association, 2012.

[55] urcs sync/Montage. Fix allocator destruction. URL: https://github.com/urcs-sync/Montage/commit/3384e50105348fab6d80e897bfb4a0efdd8aa825.

[56] urcs sync/Montage. Fix for automatic recoverability. URL: https://github.com/urcs-sync/Montage/pull/36.

[57] 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'11, page 91–104. ACM, 2011. doi:10.1145/1950365.1950379.

[58] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. A fast, general system for buffered persistent data structures. In *50th International Conference on Parallel Processing*, ICPP'21. ACM, 2021. doi:10.1145/3472456.3472458.

[59] 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'16, pages 323–338. USENIX Association, 2016. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[60] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'18, pages 461–476. USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/zuo.

A Artifact Appendix

A.1 Abstract

This appendix contains instructions to obtain the source code, build, and evaluate Mumak, a tool that detects bugs in Persistent Memory applications in an efficient and black-box manner. The repository includes the source code and instructions on how to build and run the experiments.

A.2 Description & Requirements

A.2.1 How to access. The artifact is available at https://github.com/task3r/mumak , or as a persistent DOI at 10.5281/zenodo.7737117.

A.2.2 Hardware dependencies. The evaluation of this artifact depends on the use of a machine equipped with an Intel x86 processor with support for *clwb*, *clflushopt*, *clflush* and *sfence* instructions, and a physical persistent memory module (e.g., Intel Optane DCPMM) mounted using a DAX-enabled file system.

A.2.3 Software dependencies. The system requirements to evaluate this artifact are:

- Linux (tested with Ubuntu 22.04 LTS, kernel 5.15.0)
- Docker (tested with version 20.10)
- gnuplot (tested with version 5.4)

The repository contains Dockerfiles that install the dependencies for each system evaluated.

A.2.4 Benchmarks. All relevant benchmarks are included in the repository. Experimental results are reproduced with these benchmarks as described next.

A.3 Set-up

The evaluation of this artifact depends on the use of a machine equipped with an Intel x86 processor with support for `clwb`, `clflushopt`, `clflush` and `sfence` instructions, and a physical persistent memory module (e.g., Intel Optane DCPMM) mounted using a DAX-enabled file system. To format and mount the drive (assuming the device name is `/dev/pmem0`), follow the instructions below:

```
sudo mkdir /mnt/pmem0
sudo mkfs.ext4 /dev/pmem0
sudo mount -t ext4 -o dax /dev/pmem0 /mnt/pmem0
sudo chmod -R 777 /mnt/pmem0
```

Additionally, Mumak uses an auxiliary tmpfs mount to store temporary data. Create it as follows:

```
sudo mkdir /mnt/ramdisk
sudo mount -t tmpfs -o rw,size=50G tmpfs /mnt/ramdisk
sudo chmod -R 777 /mnt/ramdisk
```

To obtain Mumak's artifact, run:

```
git clone git@github.com:task3r/mumak.git
cd mumak && git submodule update --init
```

Finally, make sure to disable address space randomization, as the analysis depends on it:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Additionally, you should also configure the coredump naming convention to obtain better debugging information:

```
echo '/tmp/core-%e.%p.%h.%t' | \
    sudo tee /proc/sys/kernel/core_pattern
```

These last two steps can be automated by running:

```
./scripts/setup_host.sh
```

A.4 Evaluation workflow

A.4.1 Major Claims.

- *(C1) Workload coverage:* Experiment E1 shows that large workloads are required in order to increase coverage during bug detection, contrasting with the evaluation presented by previous works. The results should resemble Figure 3a and Figure 3b.
- *(C2) Performance:* Mumak out-performs the state-of-the-art in PM bug detection by up to 25×, as proven by experiment E2. The results should resemble Figure 4a, Figure 4b, and Table 2.
- *(C3) Scalability:* Mumak's analysis is scalable, in the sense that the analysis time is not proportional to the size of the system under analysis, as proven by experiment E3, making it fit for real-world applications. The results should resemble Figure 5.

A.4.2 Experiments. To reproduce the experiments, navigate to the `artifact-evalution` directory of repository and build the docker images required:

```
cd artifact-evaluation
./build_images.sh
```

The automated scripts assume that the set-up was performed and that `/mnt/pmem0/` is the directory where PM is mounted.

Experiment (E1)*: [Workload coverage] [5 human-minutes + 1 compute-hour]:*

This experiment measures the coverage based on the size of the workloads imposed. It uses PMDK's `btree`, `rbtree`, and `hashmap_atomic` as benchmarks. The results are used to produce Figure 3a and Figure 3b.
[Preparation] None.
[Execution] The automated script will analyze the PM coverage of different workload sizes. Run it by issuing:

```
./run_coverage.sh
```

[Results] To plot the results, run:

```
./plot_coverage.sh
```

The plots will be generated in the `plots` folder in `eps` format. The resulting graphs should resemble Figure 3a and Figure 3b.

Experiment (E2)*: [Performance] [5 human-minutes + 140 compute-hours]:*

This experiment compares the analysis performance of Mumak with the rest of the state-of-the-art. It uses PMDK's `btree`, `rbtree`, and `hashmap_atomic` as benchmarks for PMDK 1.6 and PMDK's `btree`, `rbtree` for PMDK 1.8. Each test will run 3 times, with the exception of XFDetector and Witcher, which are expected to reach the 12h threshold. The results are used to produce Figure 4a and Figure 4b.
[Preparation] None.
[Execution] The automated scripts will analyze each target using each PM bug detection tool. Run them by issuing:

```
./run_all_pmdk1dot6.sh
./run_all_pmdk1dot8.sh
```

Or alternatively, run each system separately:

```
./run_mumak_pmdk1dot6.sh
./run_agamotto.sh
./run_xfdetector.sh
./run_mumak_pmdk1dot8.sh
./run_pmdebugger.sh
./run_witcher.sh
```

[Results] To plot the results, run:

```
./plot_pmdk1dot6.sh
./plot_pmdk1dot8.sh
```

The plots will be generated in the `plots` folder in `eps` format. The resulting graphs should resemble Figure 4a and Figure 4b.

Experiment (E3)*: [Scalability] [5 human-minutes + 12 compute-hours]:*

This experiment analyzes larger code-bases, to show that Mumak 's analysis time is not proportional to the size of the code-base under test. As benchmarks, it uses `LfHashtable` and `Hashtable`, from Montage, `cmap` and `stree`, from pmemkv, and PM-aware implementations of Redis and RocksDB. Each test will run 3 times. The results are used to produce Figure 5.
[Preparation] None.
[Execution] The automated scripts will analyze each target using Mumak. Run them by issuing:

```
./run_all_scalability.sh
```

Or alternatively, run each system separately:

```
./run_pmemkv.sh
./run_montage.sh
./run_redis.sh
./run_rocksdb.sh
```

[Results] To plot the results, run:

```
./plot_scalability.sh
```

The plot will be generated in the `plots` folder in `eps` format. The resulting graph should resemble Figure 5.