Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs

Bang Di
Hunan University
Changsha, China
dibang@hnu.edu.cn

Jiawen Liu
University of California, Merced
Merced, USA
jliu265@ucmerced.edu

Hao Chen
Hunan University
Changsha, China
haochen@hnu.edu.cn

Dong Li
University of California, Merced
Merced, USA
dli35@ucmerced.edu

ABSTRACT

Debugging persistent memory (PM) programs faces a fundamental tradeoff between performance overhead and bug coverage (comprehensiveness). Large performance overhead or limited bug coverage makes debugging infeasible or ineffective for PM programs. We present PMDebugger, a debugger that detects crash consistency bugs in PM programs. Unlike prior work, PMDebugger is fast, flexible and comprehensive. The design of PMDebugger is driven by a characterization that shows how three fundamental operations in PM programs (store, cache writeback and fence) typically occur in PM programs. PMDebugger uses a hierarchical design composed of PM debugging-specific data structures, operations and bug-detection algorithms (rules). We generalize nine rules to detect crash-consistency bugs for various PM persistency models. Compared with a state-of-the-art detector (XFDetector) and an industry-quality detector (Pmemcheck), PMDebugger leads to 49.3x and 3.4x speedup on average. Compared with another state-of-the-art detector (PMTest) optimized for high performance, PMDebugger achieves comparable performance (within a factor of 2), without heavily relying on programmer annotations, and detects 38 more bugs on ten applications. PMDebugger also identifies more bugs than XFDetector and Pmemcheck. PMDebugger detects 19 new bugs in a real application (memcached) and two new bugs from Intel PMDK.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

Persistent Memory, Crash Consistency, Testing, Debugging

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446744

ACM Reference Format:
Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446744

1 INTRODUCTION

Persistent memory (PM) is high performance, byte-addressable, and persistent. PMs are often paired with DRAM on the memory bus and can be accessed via load/store instructions, enabling direct manipulation of persistent data in memory. PMs have been widely studied in many software systems that are expected to recover to a consistent state and be able to resume execution in the event of a failure (e.g., system crash or power failure). Those systems include databases [1, 10, 35, 50], file systems [17, 26, 30, 32, 55, 56], key-value stores [3, 27, 52, 54] and indexes [34, 40, 42, 58, 59]. The ability to restore to a consistent state is often called the *crash consistency guarantee* [8, 38, 48].

Developing a crash-consistent PM application is challenging, because it requires both *data durability* (ensuring that the data reaches the persistence domain) and *ordering constraints* (ensuring that updates become visible in the correct order). For example, in a key-value store, when a new key-value pair is inserted, the value must be created and persisted *before* the key. However, caching and reordering of writes in the memory hierarchy makes it difficult to establish data durability and ordering guarantees. To enforce data persistence, new instructions, such as `CLWB` [9], have been introduced to efficiently write back cache lines to memory; To provide ordering guarantees, new instructions, such as `SFENCE` [9], have been introduced to order memory stores. On top of these low level instructions, there are higher-level transactional abstractions provided by some user-space libraries [4, 15, 51] to simplify programming.

Crash-consistent programs can have PM-specific bugs not seen in the traditional programs. Those bugs happen under different persistency models that order persists [18, 45]. Some bugs are caused by missing data durability or violating ordering guarantee in the persistent model, making the program unrecoverable after a crash. Unnecessary cache writebacks hurt performance. Identifying those bugs is critical to the success of PM-aware applications.

However, PM debuggers are often slow, which makes bug detection infeasible. For example, Pmemcheck (an industry-quality bug detector) and XFDetector (a state-of-the-art bug detector) introduce 218x slowdown (two hours) and 1000x slowdown (nine hours) respectively to application execution, when debugging a PM-aware real workload, memcached [35] with 1M persist operations. Such large performance overhead comes from instrumentation of memory stores, cache writebacks, memory fence instructions, and reasoning about the durability and ordering of persist operations; as well as from bookkeeping which can often dominate total overhead (e.g., 82% on average in Pmemcheck using benchmarks listed in Table 4). In particular, whenever there is a store, debugger tools record the PM location that has been modified. Whenever there is a cache writeback or fence, the debugger searches these locations to update the persistency status. Given a program with a large number of stores, cache writebacks and fences, frequent bookkeeping and expensive searching is the main overhead for time-consuming PM debugging.

To reduce performance overhead, some existing debuggers, e.g., PMTest [39] heavily rely on the programmer to add assertion-like checkers to selectively test durability and ordering guarantees. Furthermore, to support debugging for a given persistency model, this method requires the programmer to introduce new checkers into the program and re-annotate it. Adding checkers requires the programmer to have deep understanding of application semantics, persistency model and hardware primitives employed in the model, which imposes a heavy burden on the programmer. As a result, this method has limited bug coverage, which means some bugs cannot be detected due to missing programmer-added checkers.

In conclusion, debugging PM programs faces a fundamental tradeoff between performance overhead and bug coverage. Large performance overhead or limited bug coverage makes debugging ineffective or even infeasible for PM programs. We present PMDebugger, a tool to detect crash consistency bugs. Unlike prior work, it is fast, flexible and finds more bugs in PM Programs.

PMDebugger is fast. It enables high-speed debugging by introducing a highly efficient bookkeeping and updating mechanism. This mechanism is driven by a characterization study on how three fundamental operations in PM programs (store, cache writeback and fence) are interleaved and distributed in typical PM programs. Based on the study, we introduce two optimization techniques: (1) collectively managing the status of memory locations, and (2) using a hybrid data structure for bookkeeping. Using (1), PMDebugger is able to greatly accelerate deletion of records when processing fence instructions, and updating records when processing cache writeback instructions. For (2), PMDebugger combines an AVL tree and an array. Leveraging the strength of each data structure, PMDebugger splits and distributes the records of memory locations, based on the record lifetime, frequency of operations, and overhead of data structure maintenance. Our design leverages a characterization of PM programs, which is largely ignored in existing PM debuggers [8, 11, 33, 38, 39]. With consideration of the program characterization, PMDebugger is able to greatly reduce the performance overhead of PM debugging without losing bug coverage, hence breaking the fundamental tradeoff between the two.

PMDebugger is flexible. It allows the user to introduce any rule for bug detection because PMDebugger can efficiently process the three fundamental operations (store, cache writeback and fence). In essence, PMDebugger uses a hierarchical design composed of PM debugging-specific data structures, operations and bug-detection algorithms (rules)

Given the flexibility provided by PMDebugger, we generalize nine rules to detect crash-consistency bugs for various persistency models. Among the nine, four of them are unique to the emerging relaxed persistency models [18, 45]).

PMDebugger finds more bugs, because we have reduced overhead sufficiently to allow for more in-depth program analysis and PMDebugger has capability to detect various bugs for various persistency models. Using PMDebugger, we are able to identify bugs not identifiable by the existing tools [8, 11, 33, 38, 39].

In conclusion, we make the following contributions.

- We characterize PM programs in terms of how store, cache writeback and fence typically happen, shedding lights on the efficient design of a PM debugger;
- We introduce PMDebugger, a fast and flexible PM debugger that finds more bugs than previous work. It is open-sourced [46]. We generalize nine detection rules for various persistency models.
- Compared with a state-of-the-art detector (XFDetector) and an industry-quality detector (Pmemcheck), PMDebugger leads to 49.3x and 3.4x speedup on average. Compared with another state-of-the-art detector (PMTest), which is optimized for high performance, PMDebugger achieves comparable performance (within a factor of 2), without heavily relying on programmer annotations yet detects 38 more bugs than PMTest on ten applications.
- PMDebugger identifies 78 synthetic or reproduced bugs (ten bug types), while XFDetector, Pmemcheck and PMTest identify 65 (six bug types), 55 (four bug types) and 61 bugs (five bug types) respectively. More importantly, PMDebugger detects 19 new bugs in a real application (memcached [35]) and two new bugs from the Intel PMDK (the two bugs are confirmed by Intel [12, 14]).

2 BACKGROUND

2.1 Programming with Persistent Memory

A crash-consistent PM program uses store, writeback and fence instructions. A store modifies a persistent *memory location* (or a data object). A writeback persists a dirty memory location by flushing it out of the cache hierarchy. We use writeback, cache line flushing (CLF), and persist operation interchangeably in this paper. We use the term *persistency status* to indicate whether a memory location has been persisted by writeback. A fence ensures the completion of writeback and enforces ordering constraints among writes.

2.2 Recent Development in Bug Detection for PM

There are several tools to detect crash consistency bugs. We compare them in Table 1 from multiple perspectives. In terms of target software, some tools focus on file systems (such as Yat [33] for PMFS [47]) or user-space PM libraries (e.g., Pmemcheck for PMDK [15]), while PMDebugger can work for any user-space application. In terms of programming efforts, the state-of-the-art tools such as PMTest involve significant effort to annotate PM programs, while PMDebugger minimizes such programmer effort. In terms

Table 1: Comparison between existing work and PMDebugger.

	Perf. overhead	Bug coverage	Target domain	Prog. efforts	Support of relaxed models?
PMTest	Small	Low	Any	High	N
Pmemcheck	High	Medium	PMDK	Low	N
Persist. Ins. [11]	high	Medium	PMDK	Low	N
Yat [33]	High	Medium	PMFS	Low	N
XFDetector	High	Medium	Any	Low	N
PMDebugger	**Small**	**High**	**Any**	**Low**	**Y**

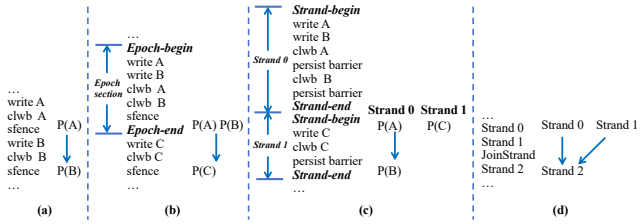

Figure 1: Examples of persistency models. `P(x)` means a write to `x` persists. The blue arrow specifies persist ordering: (a) strict persistency, (b) epoch persistency, (c) and (d) strand persistency.

of the support of various persistency models, most of the existing tools do not detect bugs specific to the relaxed persistency models, while PMDebugger does.

Most of existing tools have a similar debugging process: They instrument PM programs manually [39] or automatically [8, 11, 33, 38] to intercept stores, CLF and fences, and then keep track of information for memory locations updated and persisted in the program. Bookkeeping accounts for the major debugging overhead. It organizes the information for memory locations into a tree-like structure where each tree node records the persistency status of memory location(s). The structure is re-organized from time to time to accelerate tree operations (e.g., search and deletion). During re-organization, tree nodes are deleted or merged to record information for a larger memory location.

2.3 Persistency Model

A persistency model [18, 24, 45] defines the order in which memory stores persist to PM. There are three common persistency models: strict persistency, epoch persistency, and strand persistency (shown in Figure 1).

The strict persistency model (Figure 1a) unifies consistency and persistency models: Any two stores to PM are guaranteed to be in an order inferred by observing volatile memory order (i.e., the consistency order). The epoch and strand persistency models are relaxed persistency models: They relax persist ordering, allowing the order of persists to differ from the order of stores. As a result, they improve persist concurrency, which in turn improves performance.

The epoch model (Figure 1b) separates execution into persist epochs delineated by persist barriers (fences). The barrier enforces that no persist operation after it can happen before any persist operation before it. Persist operations within the epoch can be reordered and occur in parallel. Any two stores that conflict due to accesses to overlapped memory addresses assume the order from volatile memory order. Any two stores from the same thread and separated by a barrier are ordered. The epoch model has been frequently studied in existing work [15, 20] and employed in industry. For example, Intel PMDK bases their PM transaction mechanisms on the epoch model. A PM transaction uses TX_BEGIN and TX_END to mark the beginning and end of an epoch. Stores between TX_BEGIN and TX_END can be persisted without any order constraint, and the persist operations must be finished by TX_END.

The strand model (Figures 1c and 1d) minimizes the constraints on persist dependencies. A strand is an interval of memory execution. Memory accesses from different strands (no matter whether they come from the same thread or not) are concurrent unless explicitly ordered by the programmer. There is no implicit persist ordering constraint across strands.

To program with the relaxed models, the programmer often uses annotations, such as `TX_BEGIN` and `TX_END` in PMDK for the epoch model, and `NewStrand` and `JoinStrand` (setting up explicit persist ordering across strands) for the strand model.

3 CHARACTERIZATION OF PM PROGRAMS

Persistent memory programs, including those using low level primitives and user-space libraries, have at least three components: memory store instruction, cache writeback (i.e., cache line flushing or CLF) to enforce durability, and memory fences to provide ordering guarantees. We characterize how the three components are interleaved and distributed in typical PM programs, which motivates the design of PMDebugger. We refer to the interleaving and distribution of the three components in a PM program as the *PM program pattern*.

To identify the PM program patterns, we develop a `Valgrind` [49] tool to instrument the instructions of memory store, cache line flushing (`CLWB`, `CLFLUSH` and `CLFLUSHOPT`) and memory fence (`SFENCE`). We use PM programs from PMDK, WHISPER [41], and benchmark collections from PMTest and XFDetector. Those programs cover the most common usages of PM, including PM-aware data structures, databases, and memory caching (listed in Table 4). They use various persistency models. To cover various read/write patterns, we run YCSB (loads A-F) [7] against memcached.

We first count the distance between a store instruction and the corresponding fence instruction that guarantees the durability of that store. The fence instruction that guarantees the durability of a store is the first fence instruction following a CLF for that store. The *distance* is defined in terms of the number of fence instructions, and the fence that guarantees the durability of that store is also counted. The distance could be larger than 1, because the CLF to persist the store may be issued after the nearest fence (the first fence). As a result, the nearest fence cannot ensure the durability of the store. Figure 3 gives an example where the distance between a store to $B[1]$ and the fence instruction that ensures the durability of $B[1]$ is larger than 1. We focus on fence when calculating distance, because it is closely related to how information for memory locations should be maintained for faster debugging.

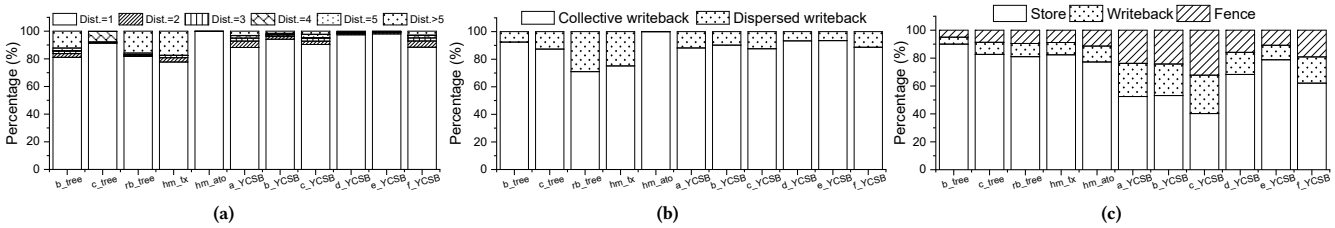

Figure 2: Characterization results. (a) Distribution of the distance between store and the corresponding fence; (b) Percentage of collective writeback and dispersed writeback in all CLF intervals; (c) Percentage of the three instructions in all of the three.

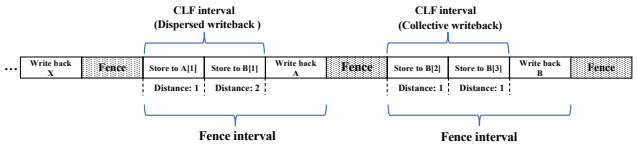

Figure 3: Examples of collective writeback, dispersed writeback, fence interval, and CLF interval. A and B represent two cache lines; A[i] or B[i] is an element in a cache line. A dispersed interval writes A[i] and B[i], while a collective writeback writes A[i] and A[j].

Figure 2a summarizes distance distribution: More than 77.7% of memory stores have distance of 1; less than 4.6% of memory stores have the distance of 2; less than 2.2% of memory stores have the distance of 3. In total, 84.5% of memory stores have distance of less than or equal to 3.

Pattern 1: For most stores, data durability is guaranteed by the nearest fence.

Inspiration from Pattern 1. Pattern 1 gives us critical information on how to store and organize information for memory locations. Traditional debuggers such as Pmemcheck, PMTest and XFDetector organize memory locations based on their addresses into a tree-like structure, for the convenience of searching (for handling CLF) and deleting (for handling fence). This method, however, comes with the overhead of tree reorganization (e.g., merging and balancing). This overhead must be outweighed by the performance benefit brought by tree reorganization. The performance benefit comes from faster search and deletion. However, Pattern 1 tells us that the bookkeeping mechanism such as tree re-organization cannot be amortized very well for many memory locations, because once the nearest fence happens, the information for the memory location is deleted, providing little opportunity to gain performance in the long term. On the other hand, we see some memory locations survive multiple fences, showing the potential of using the tree-like structure.

We partition the instruction stream collected from the `Valgrind` tool into intervals. Store instructions between two neighbouring CLF instructions form a CLF interval. Within a CLF interval, if memory locations updated in the interval are persisted together by a single cache writeback, then we say this CLF interval has *collective writeback*. If memory locations have to be persisted by multiple cache writebacks, then we say this CLF interval has *dispersed writeback*. Figure 3 gives examples of collective writeback and dispersed writeback. We count the number of CLF intervals that have collective writeback and dispersed writeback. Figure 2b shows that more than 71% of all CLF intervals have collective writeback.

Pattern 2: Memory locations updated in a CLF interval are likely to be persisted together by the same single CLF.

Inspiration from Pattern 2. Pattern 2 gives us critical information on whether it is promising to collectively maintain and update persistency status of memory locations. Collective processing enables fast query on status of memory locations, but can bring large performance benefit only when the persistency status of many memory locations *can be* collectively maintained. Pattern 2 shows us such potential.

We count the number of stores, CLFs and fences. Figure 2c shows the percentage of each instruction. In most of cases (except YCSB A, B, C and F), store accounts for 70% of the three instructions; in all cases, store accounts for at least 40.2%.

Pattern 3: Store happens more frequently than CLF and fence.

Inspiration from Pattern 3. Pattern 3 informs us that we must efficiently process memory store because of its frequent occurrences.

4 DESIGN

Overview. Figure 4 provides a high-level view of PMDebugger. As the traditional PM debugging tools [8, 33, 38], PMDebugger instruments memory store, CLF and fence instructions. PMDebugger splits the instruction trace into intervals delineated by fence instructions, called *fence interval*. Within a fence interval, there is at least one CLF interval.

PMDebugger uses a combination of an array (called *memory location array*) and an AVL tree for bookkeeping. We refer to the combination of the array and tree as the *bookkeeping space*. The array is used to store information for memory locations updated in the recent CLF intervals. Using the array structure, adding information for memory locations into the bookkeeping space is fast, incurring no overhead for re-organizing data. Such fast processing of stores is necessary due to pattern 3. This information about memory locations is deleted or re-distributed at fence instructions. In particular, the information for many memory locations in the array can be deleted at a fence, according to pattern 1, and such deletion is fast, because PMDebugger only invalidates the array metadata and does not delete the array. Also, the information for

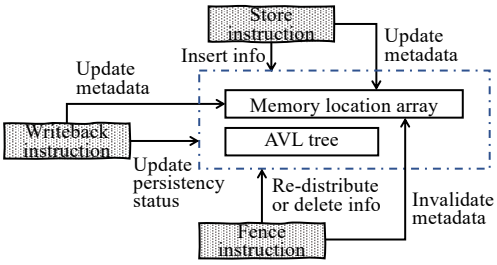

Figure 4: The overview of PMDebugger.

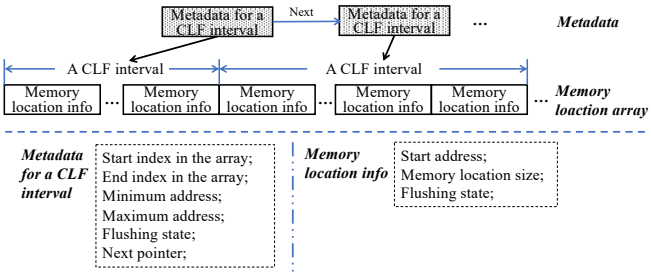

Figure 5: The array structure to record information for memory locations.

some memory locations is moved to the AVL tree, for the long-term benefit of quick searching. This data re-distribution is also driven by pattern 1.

PMDebugger introduces metadata associated with each CLF interval. The metadata records collectively if all memory locations updated in the CLF interval have been persisted or not. Using the metadata enables fast query: The persistency status of all the memory locations can be quickly examined to determine their re-distribution or deletion. Pattern 2 suggests the design of interval-based metadata.

In the following sections, we describe how PMDebugger uses the customized data structures (Section 4.1) to enable efficient processing of store (Section 4.2), CLF (Section 4.2) and fence instructions (Section 4.4). Then, building upon the data structures and algorithms to process instructions, we present rules for bug checking (Sections 4.5 and 5).

4.1 Bookkeeping of Memory Locations and Persistency

PMDebugger uses the memory location array to collect information from store instructions in a fence interval, depicted in Figure 5. Each element of the array has information collected from a store instruction, including the memory location address, location size, and flushing state (i.e., whether the memory location is persisted by a CLF instruction or not).

Based on the memory location array, PMDebugger tracks and maintains the information for CLF intervals in a fence interval. In particular, PMDebugger uses a linked list and each node of the list has metadata for a CLF interval. The metadata for a CLF interval has the following information: (1) The beginning and end of the interval, represented with the array indexes corresponding to the first and last store instructions in the CLF interval; (2) the address range of memory locations collected in the CLF interval, represented with the maximum and minimum addresses of the address range; and (3) cache flushing state of the interval.

The flushing state of a CLF interval can be either all flushed, partially flushed, or not flushed. "All flushed" means all memory locations updated in the CLF interval are flushed, while "partially flushed" and "not flushed" mean that not all memory locations or no memory locations updated in the CLF interval have been flushed, respectively. The above state is updated and maintained when processing each CLF instruction (see Section 4.4).

The memory location array is fix-sized and used to track memory locations updated by store instructions in a fence interval, where the number of store instructions is typically less than 100,000. In the rare case when the array is not big enough, the new memory locations are added into the AVL tree. At the end of a fence interval, the elements (and CLF intervals) whose flushing state is "flushed" are invalidated. The remaining elements of the array are moved to the AVL tree. In the next fence interval, the array is overwritten with information collected from store instructions (See Section 4.4).

The AVL tree is used to track those memory locations whose durability cannot be guaranteed in the short term. Hence, organizing them into the tree, even though the overhead of tree reorganization has to be paid, is beneficial in the long term, because repeated search/insertion are accelerated by the tree.

Using a small array to track store instructions for a CLF interval has multiple benefits. First, it is simple and has minimal maintenance overhead. Second, it provides better data locality for search. Third, adding information for a new memory location into the array is fast.

4.2 Processing Memory Store Instructions

Given a memory store instruction, PMDebugger performs two steps: (1) Append the store information right after the last valid element in the memory location array; and (2) update the metadata for the current CLF interval. In Step (2), the end index and address range of memory locations in the metadata are updated. Depending on the rules for bug checking, processing each memory store instruction may include extra work (e.g., detecting those repeated stores with the lack of CLF), discussed in Section 4.5.

4.3 Processing CLF Instructions

Given a CLF instruction, PMDebugger must update the flushing state of memory locations in the memory location array and AVL tree accordingly.

To update the states in the array, PMDebugger traverses the array at the granularity of CLF interval. If the persisted address range in the CLF instruction is a superset of the address range of any CLF interval, then the flushing state of that CLF interval is updated to *all flushed*. If the persisted address range partially overlaps with the address range of any CLF interval, then the state of that CLF interval is updated to *partially flushed*. The information collected from store instructions in that CLF interval must be examined *individually* to update the flushing state of individual memory locations.

When examining the information and finding that a memory location is partially overlapped with the persisted address range,

then the memory location is split into two sub-ranges: a fully overlapped and a non-overlapped. The fully overlapped sub-range stays in the memory location array, while the non-overlapped sub-range is inserted into the AVL tree. The non-overlapped sub-range is not appended to the array, because that creates difficulty of tracking the start/end indexes of the CLF interval in the array. The non-overlapped sub-range does not happen often.

After updating the flushing states in the array, PMDebugger traverses the AVL tree to update the flushing states. Since the AVL tree does not have CLF intervals to collectively annotate the flushing state of memory locations, traversing the tree can be slow. However, using the memory location array significantly avoids repetitive traversal of the tree. After updating all flushing states, PMDebugger starts a new CLF interval by creating a metadata node in the metadata linked list.

4.4 Processing Fence Instructions

Given a fence instruction, PMDebugger removes those persisted nodes from the AVL tree or memory location array, because their durability is guaranteed at the fence.

To remove nodes from the AVL tree, PMDebugger traverses the tree to look for persisted nodes. To remove elements from the memory location array, PMDebugger leverages metadata for CLF intervals to collectively and efficiently search and remove elements that have been flushed, similar to the way PMDebugger updates the flushing states when processing a CLF instruction. After the elements' removal, if there are elements remaining but not flushed, they are inserted into the AVL tree. These elements have not been persisted recently and may need to be tracked for long time. Hence putting them into the AVL tree may be beneficial for performance. After processing the memory location array, PMDebugger removes all metadata for CLF intervals. The array is ready for the next fence interval.

In the above process of removing persisted nodes/elements, PMDebugger works on the AVL tree first and then on the array. Working in this order is useful to improve performance, because when working on the array, the AVL tree is reduced and smaller, which is useful to accelerate node insertion.

When inserting elements of the memory location array into the AVL tree, we grow the tree size, which increases the overhead of node search and insertion in the future. To reduce the overhead, we could merge tree nodes as is done in traditional implementations. However, the merging operation is expensive, because it often comes with re-structuring the tree. To avoid the above tradeoff between simplifying tree and the overhead of tree reconstruction, PMDebugger performs a merging operation, only when the number of tree nodes is larger than a threshold (500).

4.5 Rules for Bug Detection

Building upon the data structures (Section 4.1) and algorithms to process instructions (Sections 4.2, 4.3 and 4.4), PMDebugger has the flexibility of implementing or extending any rule for bug checking. PMDebugger supports the detection of ten types of bugs in the three persistency models. We define the rules to detect five of them as follows. They are common to all persistency models. We define the rules to detect bugs specific to the relaxed persistency models in Section 5. We consider both correctness bugs and performance bugs, following the convention of existing bug detection tools (Pmemcheck, PMTest, XFDetector, and Agamotto [43]). For example, flushing a cache line more than once does not affect program correctness, but does incur unnecessary overhead; we consider this a bug.

- **No durability guarantee:** A persistent memory location is not persisted after the last write to it. This bug can happen when the programmer misses CLF or fence instruction. To detect this bug, after the PM program finishes, PMDebugger checks if there are any remaining memory locations in the bookkeeping space: If the flushing state of a memory location is *flushed*, then the PM program is missing a fence. If the flushing state of a memory location is *not flushed*, then the PM program is missing a CLF.
- **Multiple overwrites:** The program writes to the same persistent memory location multiple times, before the durability of the memory location is guaranteed. To detect this bug, when processing a store, PMDebugger examines if the memory location to which the store instruction writes already exists in the array or tree. If so, PMDebugger reports a multi-write bug. PMDebugger does not detect this bug for relaxed persistency models, because multiple overwrites is not a bug in those models.
- **No order guarantee:** The program cannot guarantee the order in which writes become persistent. To detect this bug, PMDebugger asks the programmer to specify in a debugger configuration file which variable X must be persisted before Y and at which application function. During debugging, PMDebugger maps the variables to their addresses based on symbol tables or by intercepting dynamic memory allocations, instruments the application function, and registers a callback function. When the application function is called and PMDebugger processes a fence instruction, PMDebugger checks if X is persisted and the durability of X is guaranteed by the fence before Y.
- **Redundant flushes:** A store to a memory location is flushed multiple times before the nearest fence. This bug causes performance loss. To detect this bug, PMDebugger checks if the flushing state of the memory location is *flushed* when processing a CLF instruction that persists the memory location. If yes, then a bug is detected.
- **Flush nothing:** A CLF instruction does not persist any prior store. When processing a CLF instruction, if PMDebugger finds that the memory location persisted by the instruction does not exist in the bookkeep space, then PMDebugger reports a bug.

5 DETECTING BUGS SPECIFIC TO RELAXED PERSISTENCY MODELS

The relaxed persistency models can introduce some special bugs unseen in the strict persistency model.

5.1 PMDebugger Extension

To detect bugs for the epoch persistency model, PMDebugger extends the information for memory locations in the bookkeeping space by adding a flag to indicate if the store that writes to a memory location comes from an epoch section, shown in Figure 6. Processing CLF and fence for the epoch persistency model remains the same.

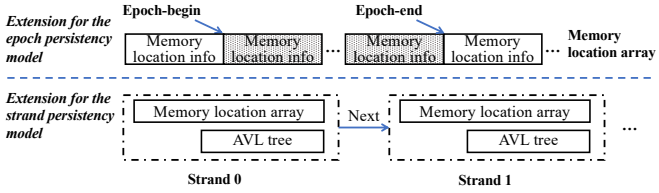

Figure 6: Extension to support a relaxed persistency model.

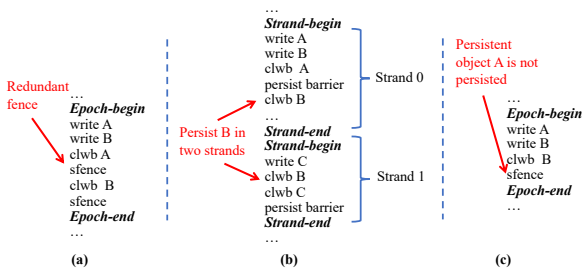

Figure 7: Three new bugs in relaxed persistency models.

To detect bugs for the strand persistency model, PMDebugger sets up a memory location array and an AVL tree for each strand section. Providing an independent bookkeeping space to each section is necessary, because strand sections can happen in parallel, which can introduce conflicting updates to the array and tree. When processing store, flush and fence instructions, PMDebugger must be able to identify the strand sections from which the instructions come, such that PMDebugger can update the corresponding bookkeeping spaces. Identifying the strand sections for the instructions can come from hardware support [18] or software instrumentation.

It is possible that multiple strand sections update the same memory location, but persisting the updates from the strand sections must follow a specific order specified by the programmer. To support the detection of mis-ordering, PMDebugger introduces a small array, and each element of the array is used to track the flushing state of a memory location shared and updated by the multiple sections. The array is shared between the sections and used to check persistency order.

5.2 Rules for Bug Detection

Building upon the above extension, PMDebugger introduces rules to detect four bugs specific to the relaxed persistency model. The first three bugs have never been studied before.

- **Redundant fences in an epoch section (referred to as redundant epoch fence):** More than one fence can exist in an epoch section as shown in Figure 7a. Unnecessary fences degrade performance. To detect this bug, PMDebugger records the number of fence instructions in an epoch section. At the end of the epoch section, if the number of fence instructions is greater than one, PMDebugger reports a bug.
- **Violating order guarantee because of multi-strands (referred to as lack ordering in strands):** Persisting memory locations across strands can violate the order guarantee. We use Figure 7b as an example to illustrate this problem. There are two strands in Figure 7b. Assume that there is a requirement that persisting *A* must happen before persisting *B* in strand 0. However, in strand 1, persisting *B* can happen earlier and break the order. To detect this bug, when processing a CLF instruction, PMDebugger checks if there is any ordering requirement related to the memory location persisted by the CLF in another running strand. If yes, PMDebugger checks if the CLF violates the ordering. If it does, PMDebugger reports a bug.
- **Lack durability guarantee in an epoch section (referred to as lack durability in epoch):** At the end of an epoch, the durability of *all* memory locations updated by store instructions in the epoch cannot be guaranteed. Figure 7c gives an example where the durability of a memory location *A* cannot be guaranteed at the end of an epoch due to a missing writeback. To detect this bug, at the end of an epoch section, PMDebugger processes the barrier (i.e., fence) as depicted in Section 4.5 and then checks if there is any memory location in the bookkeeping space that belongs to the epoch. If yes, PMDebugger reports a bug.
- **Redundant logging:** In a logging-based transaction in PMDK, a data object is updated once but logged multiple times. Redundant logging leads to performance loss. To detect this bug, PMDebugger treats writing the log as a store instruction. The address of the data object in the log is treated as the address to be stored into in the store instruction. To detect redundant logging, PMDebugger uses the existing rule of detecting multiple overwrites. When "overwrite" happens, PMDebugger reports a bug.

Table 2: Software interfaces for PMDebugger.

Register_pmem (addr_base, size, offset)	Register a persistent memory location for debugging
Epoch_begin	Mark an epoch region for PMDebugger detection
Epoch_end	
Strand_begin	Mark an strand region for PMDebugger detection
Strand_end	

6 IMPLEMENTATION

PMDebugger uses `Valgrind` [49] to instrument store, writeback and fence instructions and registers three callback functions (each instruction has a corresponding callback function) to be called each time the instructions are intercepted. PMDebugger implements bug detection rules as individual functions, giving the callback functions great flexibility to use them for bug detection. The memory location array is essential for PMDebugger.

PMDebugger has a set of interfaces summarized in Table 2. `Register_pmem` in Table 2 is used to register a memory region for debugging. This API can be embedded into a low-level system API (such as `mmap`) to enable automatic annotation. The other APIs are used to mark code regions for bug detection, not for selecting variables to test durability and ordering guarantee as in existing work [39]. Using those interfaces is simple and does not require deep understanding of applications.

To handle nested transactions, PMDebugger uses the design from Pmemcheck. Its effectiveness is evaluated in Section 7. In particular, PMDebugger treats the nested transactions as a single transaction, delineated by the outermost `epoch_begin` and `epoch_end`. This method is based on the fact that nested transactions do not guarantee data persistence until the outermost `epoch_end`.

Table 3: System configurations.

CPU	Intel Xeon Gold 5218 CPU @ 2.30GHz, 32 cores
PM	6x128GB Intel DCPMM, App Direct
DRAM	156G DDR4, 2666MT/s
OS	Ubuntu 18.04, Linux kernel 5.0
Tools	gcc/g++-9.2, Valgrind-3.15

Table 4: PM programs for evaluation.

	Name	Model	LOC	Configurations
Micro-bench	synth_strand	strand	1.7k	default
	b_tree	epoch	981	
	c_tree		698	
	r_tree		756	
	rb_tree		855	
	hashmap_tx		741	
	hashmap_atomic		837	
Real workloads	memcached	strict	23k	Memslap (5% set)
	redis	epoch	66k	redis-cli (LRU test)

The current implementation of PMDebugger is designed to debug programs in user space. But PMDebugger can be extended to debug programs in kernel space. To debug a program in kernel space, the programmer would have to register the memory space used by the kernel program using `Register_pmem`. Then, leveraging the instrumentation infrastructure of Valgrind, PMDebugger instruments instructions and debugs kernel program, in a way similar to how PMDebugger debugs the user-space program.

7 EVALUATION

7.1 Methodology

Machine. We evaluate PMDebugger on a machine with Intel Optane DC Persistent Memory Module (DCPMM) [13] using the App Direct mode of DCPMM. Table 3 shows details.

Benchmarks. To evaluate PMDebugger, we choose a set of benchmarks including seven micro-benchmarks (one synthetic benchmark plus six benchmarks from PMDK [15]) and two real world workloads redis [10] and memcached [35]. Table 4 summarizes all benchmarks, including their persistency models, LOC (lines of code), and execution parameters. We use these benchmarks, because most of them are used in previous studies [15, 38, 39]. In addition, because there is no hardware and applications supporting strand persistency [18], we develop a synthetic benchmark (`synth_strand` in Table 4) based on two programs (`b_tree` and `c_tree`). In `synth_strand`, `b_tree` and `c_tree` are placed into two independent strands. All results reported in this section are the average of ten runs.

7.2 Performance

Figure 8 compares the performance of PMDebugger and Pmemcheck. We choose Pmemcheck for comparison, because it is an industry-quality detector based on `Valgrind` [49] as is PMDebugger. To precisely quantify performance overhead without including instrumentation overhead, we present application execution time under `Nulgrind`, which is a `Valgrind` tool using the same instrumentation as PMDebugger without any bookkeeping.

Table 5: The performance improvement of PMDebugger over Pmemcheck. The second column shows the overall speedup (including instrumentation time), and the third column shows speedup without instrumentation time.

Benchmarks	With Instru.	W/O Instru.
b_tree	2.67x	4.04x
c_tree	1.85x	2.74x
r_tree	1.81x	1.86x
rb_tree	2.24x	3.42x
hashmap_tx	1.34x	1.39x
hashmap_atomic	3.32x	6.83x
synth_strand	2.2x	3.34x
memcached	4.67x	7.89x
redis	2.1x	2.74x

Microbenchmarks. We use seven PMDK-based microbenchmarks. We test each program with a different number of insertions (1K, 10K and 100K). Table 5 summarizes the results shown in Figure 8. Compared with Pmemcheck, PMDebugger achieves 2.2x performance improvement on average (up to 3.6x in `hashmap_atomic` with 100K insertion). After removing the overhead of instrumentation, PMDebugger has 3.5x improvement on average (up to 7.5x in the case of `hashmap_atomic` with 100K insertion). The program `hashmap_atomic` has the largest performance improvement, because the ratio of collective writeback to the total number of CLF intervals in this benchmark is the highest among all benchmarks (see Figure 2b), which provides opportunities to put most of the memory location information in the memory location array. `hashmap_tx` exhibits relatively low performance improvement, compared with other benchmarks, because more memory location information stays in the AVL tree, which degrades performance (detailed in Section 7.5). For most of transactional programs (`b_tree`, `c_tree`, `r_tree`, and `rb_tree`), PMDebugger leads to large improvements (3.02x on average).

Real workloads. Table 5 summarizes the results for `memcached` and `redis` shown in Figure 8. For `memcached`, we observe 4.67x improvement on average (7.89x when excluding the instrumentation overhead) over Pmemcheck. For `redis`, PMDebugger outperforms Pmemcheck by 2.1x (2.74x when excluding the instrumentation overhead). PMDebugger performs better on `memcached`, because `memcached` generates many more store instructions, which provides more opportunities to reduce debugging time.

Comparison with other state-of-the-arts. We compare with XFDetector and PMTest (two state-of-the-art debuggers for PM programs). To enable fair comparison, we do not include their instrumentation time, because they use PIN-based binary instrumentation and annotation-based manual instrumentation respectively, which is different from the instrumentation mechanism in PMDebugger. We use all benchmarks listed in Table 4 except the benchmark `r_tree`. We do not use `r_tree`, because PMTest and XFDetector do not evaluate the performance of `r_tree`.

Table 6: Comparison of bug detection capability among Pmemcheck, PMTest, XFDetector and PMDebugger.

	No durability guarantee	Multiple overwrites	No order guarantee	Redundant flushes	Flush nothing	Redundant logging	Lack durability in epoch	Redundant epoch fence	Lack order in strands	Cross-failure semantic
Bug cases	44	2	4	6	3	5	4	4	2	4
Pmemcheck	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗
PMTest	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
XFDetector	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓
PMDebugger	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

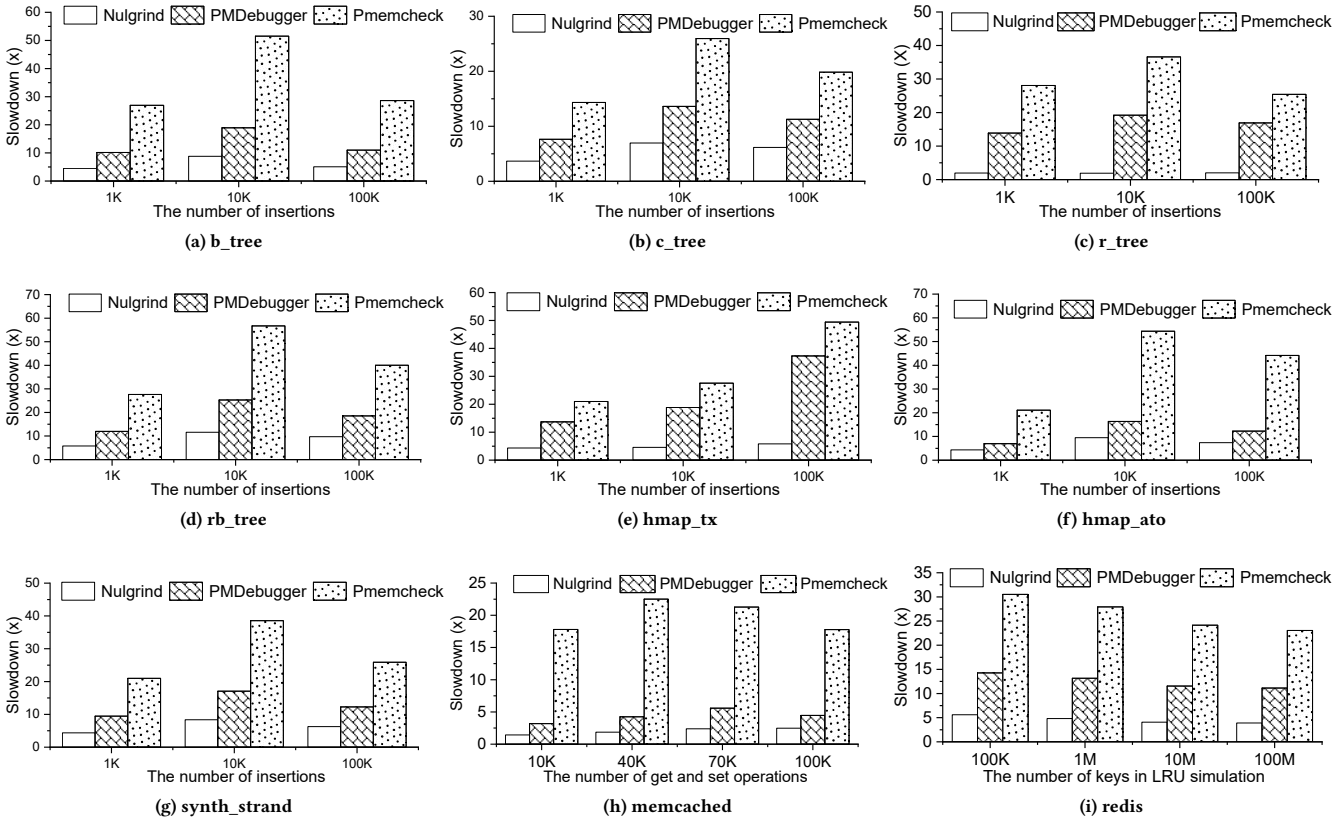

Figure 8: Performance comparison. Results are normalized by the execution time of original programs with detectors disabled.

XFDetector causes 370x slowdown (on average) over the original program [38], which is much worse than PMDebugger (7.5x slowdown on average). PMTest causes 3.8x slowdown in our evaluation, which is better than PMDebugger (but the performance difference is less than 100%), because it selectively tests durability and ordering guarantee. However, PMTest trades performance for low bug coverage. PMDebugger detects 38 more bugs than dose PMTest (detailed in Sections 7.3-7.4).

7.3 Bug Detection Capability

We use a dataset containing 78 bugs. 68 of those bugs come from bug evaluation suites [8, 38, 39], which include synthetic bugs and reproduced from the commit history of PMDK. Because the bug evaluation suites do not include those bugs specific to the relaxed persistency models, we add ten extra synthetic bugs. Table 6 shows the evaluation results.

PMDebugger detects ten types of bugs (78 bugs in total). A cross-failure semantic bug means the program reads semantically inconsistent data during the post-failure execution. Note that to detect cross-failure semantic bugs [38], the debugger must have the capability of pausing and resuming threads at runtime to recover a program at the failure point; Valgrind used in PMDebugger cannot do that. To solve this problem, we manually call the recovery program in benchmarks to detect the cross-failure bugs.

Comparison with state-of-the-art. We evaluate Pmemcheck, PMTest and XFDetector. PMTest can detect only five types of bugs (61 bugs out of the 78 bugs) . Its bug detection capability is limited, because it relies on the programmer to annotate the program and these annotations also include calls to explicit debugging routines (the annotation in the benchmarks are added by the PMTest developers). Pmemcheck and XFDetector detect only four and six types of bug respectively (55 and 65 bugs respectively), because

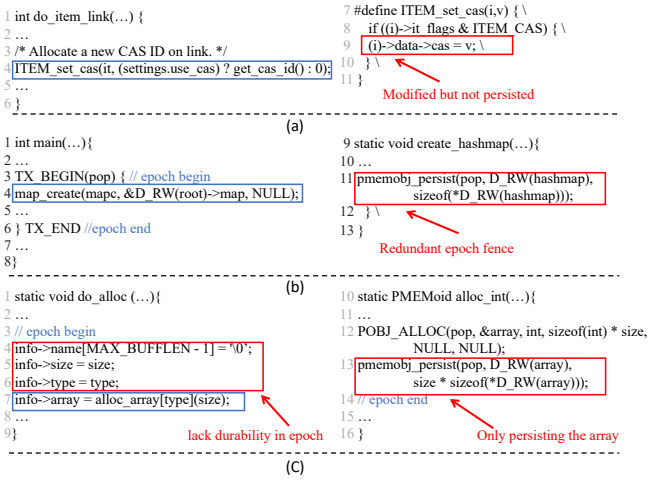

Figure 9: New bugs detected by PMDebugger.

they cannot thoroughly examine instructions and have rich rules as PMDebugger have due to their extremely time-consuming analysis.

False positives and false negatives. We report the rates of false positive (i.e., treating a correct code as a bug) and false negative (i.e., missing bugs). The false negative rate of Pmemcheck, PMTest, and XFDetector is 29.5%, 21.8%, and 16.7% respectively, while PMDebugger does not have any false negative case. All tools do not have any false positive case.

7.4 New Bugs Found By PMDebugger

PMDebugger finds 19 new bugs in memcached [35] and two new bugs ("redundant epoch fence" and "lack durability in epoch") in PMDK. These bugs were not reported before. Due to space constraints, we present only one of the 19 new bugs and the two new bugs in PMDK (see Figure 9). The remaining 18 new bugs are reported in our tech report [46].

Bug 1: No durability guarantee. Figure 9a shows a code snippet from `Items.c` in memcached. The program defines a macro function (Line 7 in Figure 9a) to allocate a new CAS ID. However, the CAS ID is not persisted (Line 4 in Figure 9a).

Bug 2: Redundant epoch fence. We find this bug in the PMDK example `hashmap_atomic` (`data_store.c` and `hashmap_atomic.c`, and shown in Figure 9b). The program declares an epoch section (surrounded by `TX_BEGIN`, and `TX_END` from Line 3 to Line 6). `TX_END` inserts a fence instruction at the end of the epoch section. However, a function call `map_create` (Line 4) is redirected to another call (`create_hashmap`) that inserts a redundant fence in `pmemobj_persist` (Line 11) in the epoch section. This 'bug' is confirmed by Intel [14].

Bug 3: Lack durability in epoch. We find this bug in the PMDK example `array` (in `array.c`, and shown in Figure 9c). This example has an epoch section where the program modifies PM from Line 4 to Line 6. Then `alloc_int` (Line 10) is called at Line 6 to allocate and initialize an array structure. However, the program only persists the modified array structure at the end of the epoch section (Line 14) and does not persist the data (Line 4 to Line 6). This bug is confirmed by Intel [12].

The above bugs were not detected by other detectors (particularly, XFDetector and PMTest). For instance, XFDetector uses `memcached`, but could not identify the above bugs [38], because it has to restrict the number of instrumented failure points to reduce its overhead, resulting in lower bug coverage. PMTest misses the above bugs [39], because it heavily relies on the programmer, which imposes big constraints on its detection capability for complicated applications.

7.5 Other Analysis

Scalability. We use memcached for evaluation, because it is PM-operation intensive and hence can be used effectively to evaluate the scalability. We change the number of memcached threads for evaluation (larger number of threads means higher PM-operation intensity.). Figure 10 presents the results for Pmemcheck and PMDebugger. The slowdown of Pmemcheck increases almost linearly as the number of threads increases, while the slowdown of PMDebugger increases much more slowly due to the effectiveness of bookkeeping. This shows that PMDebugger can effectively reduce debugging time even in multi-threading environment.

Quantification of AVL tree size. PMDebugger uses the memory location array to store information for memory locations and re-distributes it between the array and tree when processing fence instructions (detailed in Section 4.4). However, a large AVL tree can degrade performance for search and node insertion. We measure the average number of tree nodes in all fence intervals to quantify tree size.

Figure 11 shows the results. We compare Pmemcheck and PMDebugger, because Pmemcheck also uses an AVL tree. In most cases, the tree size of PMDebugger is less than 25 tree nodes, which shows the effectiveness of using the memory location array to reduce the tree size. In `hashmap_tx`, the tree size is relatively large (528), because many stores are persisted very late after stores, and data durability is not guaranteed by the nearest fence. Hence, the information for many stores is saved in the tree. But still, PMDebugger has 1.4x performance improvement over Pmemcheck, which does not use the hybrid data structure. PMDebugger reduces the tree size for all benchmarks: For all benchmarks except `hashmap_tx` (`b_tree`, `c_tree`, `r_tree`, `rb_tree`, `hashmap_atomic`, `memcached`, and `redis`), the tree size is reduced from 17.6 to 8.9 on average. For the benchmark `hashmap_tx`, the tree size is reduced from 619 to 528.

Key insight of why PMDebugger works better. In general, PMDebugger finds bugs that other state-of-the-art solutions (Pmemcheck, XFDetector and PMTest) cannot for the following reasons: (1) Pmemcheck and XFDetector come with large performance overhead, which makes PM-debugging too time-consuming to comprehensively detect bugs (especially for those complicated applications). Such a large overhead is unacceptable in real workloads. Such a large overhead comes from the fact that those debugging tools do not consider how PM applications are typically programmed. As a result, their tree-like data structures cannot efficiently be used for bookkeeping. For example, to debug `hashmap_atomic`, Pmemcheck performs 35,9209 expensive tree reorganizations, while PMDebugger performs only 788. (2) PMTest relies on manual annotations to

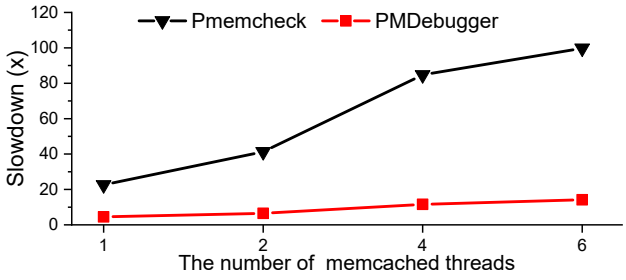

Figure 10: Execution time with various number of threads.

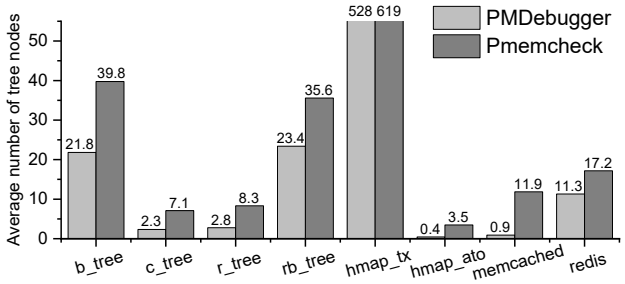

Figure 11: Average number of tree nodes in all fence intervals.

reduce debugging overhead, but this results in missed bugs. For example, PMTest cannot detect the bug shown in Figure 9a, if PMTest missed annotating a store (Line 9 in Figure 9a).

8 DISCUSSIONS

Programmer-supplied persistency order. To detect the bug of "no order guarantee", the programmer must provide order information (see Section 4.5). This is inevitable when the order information cannot be automatically and correctly derived from programs due to implicit program semantics. All of the existing tools (e.g., PMTest, Pmemcheck and XFDetector) ask the programmer to provide order information. In terms of usability, PMDebugger is no worse than those tools. In fact, using PMDebugger is easier, because the information only needs to be specified once in a configuration file (without using any API or UI) and is then repeatedly used during the debugging process. Other tools (such as PMTest) ask the programmer to manually specify order information everywhere in the program, which is laborious.

Annotation difference between PMDebugger and existing works. Both XFDetector and PMDebugger need only a few annotations to debug. However, XFDetector causes much longer debugging time than PMDebugger (7.5x longer on average). PMDebugger requires much less annotation than PMTest. For example, to debug `memcached`, PMTest requires 410 annotations, while PMDebugger and XFDetector require only 25 and 23 respectively. Adding annotations to use PMDebugger is less laborious for three reasons: (1) Annotations in Table 2 can be inserted into traditional high-level PM-programming primitives (e.g., TX_BEGIN and TX_END), such that adding annotations happens automatically when the programming primitives are used by the programmer; (2) PMDebugger uses a configuration file, which allows the users to generally specify the persistence order of certain variables, while PMTest asks the user to find exactly where the variables are referenced in the program and annotate them; (3) PMTest introduces annotations customized to bug types, while PMDebugger does not need those annotations due to its rule-based detection mechanism.

9 RELATED WORK

Crash consistency debugging. Yat [33] is designed for PMFS [47] using an exhaustive testing method. Yat incurs extremely large overhead, and its approach cannot be applied to generic programs due to its dependency on file system check (fsck) to detect inconsistencies. Pmemcheck and XFDetector work for applications at the user space. They do not leverage PM program patterns to improve debugging performance. PMTest relies on the programmer to annotate applications to assist bug detection. It sacrifices debugging accuracy for high performance.

Static analysis methods (such as NVM-C [16]) identify bugs before program execution and provide quicker turnaround to fix bugs. However, they cannot detect bugs related to dynamic memory allocation. Thus, they can incur false positives and negatives. PMDebugger does not use static analysis.

Enabling crash consistency on PM. Existing studies propose a variety of crash consistency mechanisms in software [2, 5, 19, 21, 22, 31, 37, 53, 57] and hardware [23, 25, 28, 36, 44]. NV-Heaps [4], Mnemosyne [51], and PMDK [15] are libraries to lessen the burden of programmers to build crash-consistent PM applications. PMFS [47], BPFS [6] and NOVA [55] are PM-aware file systems that support persistent data storage on PM. DPO [29] and HOPS [41] introduce efficient persistency models to allow programmers to build crash consistent applications with various performance and crash consistency. PMDebugger could be applied to debug these solutions, since the mechanisms proposed in this work are general and not restricted by hardware or programming environments.

10 CONCLUSIONS

The existing methods to detect crash-consistency bugs in PM programs are either very time-consuming or suffer from limited bug coverage. We use a new approach to examine bug detection for PM programs. By characterizing PM programs, we find that the existing methods have a mismatch between the design of data structures and algorithms and PM program patterns, which leads to inefficient debugging mechanisms. We introduce PMDebugger. By considering PM program characterization, PMDebugger enables high-performance debugging operations without losing bug coverage. Based upon its program pattern-aware data structures and algorithms, PMDebugger enables efficient bug detection for various persistency models.

ACKNOWLEDGMENTS

We thank anonymous reviewers and our shepherd, Margo Seltzer, for their valuable feedback. This work is partially supported by the National Science Foundation of China under grants 61772183 and 61972137 and the China Scholarship Council (CSC).

A ARTIFACT APPENDIX

A.1 Abstract

This artifact provides the source code of PMDebugger, a debugger to detect crash consistency bugs for persistent memory (PM) programs. This artifact also provides a set of example workloads and necessary dependencies. The design of PMDebugger is driven by the characterization of how three fundamental operations in PM programs (i.e., store, cache writeback and fence) typically happen. PMDebugger leverages a hierarchical design composed of PM debugging-specific data structures, operations and bug-detection algorithms (rules) based on the instrumentation of memory store, CLF and fence instructions. Because PMDebugger is used to debug PM systems, this artifact requires a real or emulated PM system and a compatible Linux distribution.

A.2 Artifact Check-List (Meta-information)

- **Program:** The debugging tool, PMdebugger.
- **Data set:** Open-source workloads from Intel and Lenovo.
- **Hardware:** A system with a real or an emulated PM.
- **Output:** Bug reports for test programs.
- **Experiments:** Bug detection and execution time.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** GNU GPL V2.0.
- **Archived DOI:** https://doi.org/10.1145/3410281.

A.3 Description

A.3.1 How to Access. We archive the source code and workloads at ACM DL: https://doi.org/10.1145/3410281. For the latest version, you can access our GitHub page: https://github.com/PASAUCMerced/PMDebugger.

A.3.2 Hardware Dependencies. This artifact expects to use a system equipped with persistent memory hardware (e.g., Intel Optane DC persistent memory module). The user can also use an emulated PM hardware based on DRAM (see https://pmem.io/2016/02/22/pm-emulation.html as an example). Note that the real or emulated PM device must be mounted as a DAX file system.

A.3.3 Software Dependencies. The following is a list of software dependencies for PMDebugger and workloads (the listed versions have been tested, and other versions might work but not be guaranteed):

- OS: Ubuntu 18.04, Linux kernel 5.0.
- Compiler: g++/gcc-9.2.
- Tool: Valgrind-3.15.
- Dependent libraries: libevent, libseccomp, autoconf, pkg-config, libndctl-devel (v63 or later), libdaxctl-devel (v63 or later).

A.3.4 Data Sets. The evaluation workloads are as follows:

- Five workloads (b_tree, c_tree, rb_tree, hashmap_tx and hashmap_atomic) from PMDK [15] and a synthetic benchmark (synth_strand).
- PM-aware Redis [10] implementation from Intel.
- PM-aware Memcached [35] implementation from Lenovo.

A.4 Installation

The code repository is organized as follows:

- `valgrind-pmdebugger/`: The source code of PMDebugger (v1.0).
- `pmdk/`: Intel's PMDK (v1.8) library, including its example PM programs.
- `memcached-pmem-master/`: An Memcached (v1.5.4) implementation.
- `memslap/`: The tool (v1.0) to run Memcached.

To build PMDebugger and the test workloads, the user can run the following scripts:

```
$ cd <PMDebugger Root>
$ ./build_pmdebugger.sh
$ sudo -E ./build_pmdk.sh
$ ./build_redis.sh
$ ./build_memcached.sh
```

In addition, our tool and test programs (workloads) include detailed instructions to build PMDebugger and workloads separately.

A.5 Experiment Workflow

PMDebugger is based on Valgrind. To debug a PM program, the user directly runs the program with PMDebugger. To detect bugs of "no order guarantee", the user is required to specify persistence orders in a configuration file. During the program execution, Valgrind automatically intercepts memory store, CLF and fence instructions, and send them to PMDebugger. Once a bug is detected, PMDebugger immediately reports it and then continues the debugging process. After the program is finished, PMdebugger outputs a detailed bug summary.

A.6 Evaluation and Expected Result

The evaluation includes both bug detection and performance measurement (execution time or system throughput).

A.6.1 Performance Evaluation. We provide scripts to run programs. The detailed steps to run them are as follows. For comparison purposes, the user can run Pmemcheck as well. Pmemcheck is an industry-quality bug detector from PMDK. Both Pmemcheck and PMDebugger are based on Valgrind.

PMDK examples. We use a script `pmdk/run.sh` to run all programs (workloads). The usage of the script is as follows.

```
$ ./run.sh <CHECKER> <INPUTSIZE> <WORKLOAD>
```

- CHECKER: Debugger tool name (pmdebugger or pmemcheck).
- INPUTSIZE: The number of data insertions.
- WORKLOAD: The workload to be tested.

For example, to insert 1024 elements into b_tree to evaluate the performance of PMDebugger, the user can run the following command:

```
$ ./run.sh pmdebugger 1024 btree
```

Redis. We use a script `redis/run.sh` to run Redis.

```
$ ./run.sh <CHECKER> <INPUTSIZE>
```

– CHECKER: Debugger tool name (pmdebugger or pmemcheck).
– INPUTSIZE: The number of LRU tests.

Memcached. We use a script `memslap/run.sh` to run memcached.

```
$ ./run.sh <CHECKER> <INPUTSIZE>
```

– CHECKER: Debugger tool name (pmdebugger or pmemcheck).
– INPUTSIZE: The number of operations to execute.

Output. PMDebugger reports all detected bugs (if any) after a test is complete. The evaluation of PMDK and Memcached programs includes execution time, and the evaluation of Redis includes the throughput (i.e., the number of "get" operations per second).

A.6.2 Bug Detection Capability. We implement the nine bug detection rules in PMDebugger (see the paper) for various persistency models. To verify PMDebugger's capability of bug detection, we add bug cases. They can be found in `valgrind-pmdebugger/pmdebugger/tests` and classified by their bug types.

The bug cases are organized in the following way:

- `address_specific/` and `logging_related/`: Function tests.
- `no_durability_guarantee/`: A persistent memory location, since the last write to it, is not persisted.
- `multiple_overwrite/`: The program writes to the same persistent memory location multiple time, before the durability of the memory location is guaranteed.
- `no_order_guarantee/`: The program cannot guarantee the order in which writes become persistent.
- `redundant_flush/`: A store to a memory location is flushed multiple times before the nearest fence.
- `flush_nothing/`: A CLF instruction does not persist any prior store.
- `epoch_redundant_fence/`: More than one fence exist in an epoch section.
- `lack_ordering_in_strand/`: Persisting memory locations across strands violates the order guarantee.
- `epoch_durability`: At the end of an epoch, the durability of all memory locations updated by store instructions in the epoch cannot be guaranteed.
- `redundant_logging/`: In a logging-based transaction, a data object is updated once but logged multiple times.

The user can run the following command to verify the bug detection capability:

```
$ cd <PMDebugger Root>/valgrind-pmdebugger
$ perl tests/vg_regtest pmdebugger
```

Output. The number of bugs is reported. If no bug is detected, PMDebugger outputs a file to report the reason.

A.6.3 New Bugs Found by PMDebugger. PMDebugger finds 19 new bugs in Memcached and two new bugs in PMDK (confirmed by Intel [12, 14]. Those bugs were not reported before. We provide scripts to reproduce the detection of those bugs.

- No durability guarantee in Memcached: Use the script `memslap/new_bug1.sh` to detect the bugs.
- Redundant epoch fence in PMDK: Use script `pmdk/new_bug2.sh` to detect the bug.
- Lack durability in epoch in PMDK: Use script `pmdk/new_bug3.sh` to detect the bug.

Output. PMdebugger reports the detected bugs.

A.7 Experiment Customization

For programs based on PMDK, the user can directly use PMDebugger without adding any annotation in the programs. For example:

```
$ valgrind --tool=pmdebugger ./WORKLOAD
```

For more details, please run `valgrind --tool=pmdebugger -h`.

For other programs, the user is required to insert limited annotations into the programs, such as epoch_begin (VALGRIND_PMC_EPOCH_BEGIN) and epoch_end (VALGRIND_PMC_EPOCH_END). Then, after recompiling the programs, the user can debug them in the above way. Note that those annotations can be inserted into PMDK APIs, and hence added automatically and transparently. For example, the user can insert epoch_begin and epoch_end into PMDK's TX_BEGIN and TX_END to achieve automatic annotation.

REFERENCES

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[2] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*.

[3] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[4] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[5] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*.

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

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*.

[8] Intel Corporation. 2015. An introduction to pmemcheck. https://pmem.io/2015/07/17/pmemcheck-basic.html.

[9] Intel Corporation. 2018. Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[10] Intel Corporation. 2018. Redis. https://github.com/pmem/redis/tree/3.2-nvml.

[11] Intel Corporation. 2020. Detect Persistent Memory Programming Errors Using Persistence Inspector. https://software.intel.com/content/www/us/en/develop/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.html.

[12] Intel Corporation. 2020. Inconsistency bugs in array example for libpmemobj. https://github.com/pmem/pmdk/issues/4927.

[13] Intel Corporation. 2020. Intel Optane DC persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[14] Intel Corporation. 2020. Obj: fix data_store example transaction logic to remove redudant fences. https://github.com/pmem/pmdk/pull/4939/commits/e394307ef2baea1de31fa054a1e2c3dff3581a59.

[15] Intel Corporation. 2020. Persistent memory programming. https://pmem.io/.

[16] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2016. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[17] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[18] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *ACM/IEEE 47st International Symposium on Computer Architecture (ISCA)*.

[19] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*.

[20] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[21] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-Threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*.

[22] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-Aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment* (2014).

[23] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[24] Jungi Jeong and Changhee Jung. 2021. PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency). In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[25] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[26] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[27] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*.

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

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

[30] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. 2017. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *15th USENIX Conference on File and Storage Technologies, (FAST)*.

[31] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.

[32] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas E. Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.

[33] Philip Lantz, Dulloor Subramanya Rao, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *Proceedings of the 2014 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*.

[34] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[35] Lenovo. 2018. Memcached-pmem. https://github.com/lenovo/memcached-pmem.

[36] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[37] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[38] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas F. Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[39] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Manabi Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[40] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment* (2020).

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

[42] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST)*.

[43] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[44] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[45] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*.

[46] PMDebugger. 2020. https://github.com/PASAUCMerced/PMDebugger.

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

[48] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[49] Valgrind. 2020. An instrumentation framework for building dynamic analysis tools. https://valgrind.org.

[50] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[51] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[52] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. 2016. NVMcached: An NVM-based Key-Value Cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*.

[53] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[54] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*.

[55] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST)*.

[56] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.

[57] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[58] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[59] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. (2019).