Checking Robustness to Weak Persistency Models

Hamed Gorjiara
Weiyu Luo
Alex Lee
University of California, Irvine
U.S.A.
{hgorjiar,weiyul7,leea19}@uci.edu

Guoqing Harry Xu
University of California, Los Angeles
U.S.A.
harryxu@cs.ucla.edu

Brian Demsky
University of California, Irvine
U.S.A.
bdemsky@uci.edu

Abstract

Persistent memory (PM) technologies offer performance close to DRAM with persistence. Persistent memory enables programs to directly modify persistent data through normal load and store instructions bypassing heavyweight OS system calls for persistency. However, these stores are not immediately made persistent, developers must manually flush the corresponding cache lines to force the data to be written to persistent memory. While state-of-the-art testing tools can help developers find and fix persistency bugs, a prior study has shown that fixing persistency bugs on average takes a couple of weeks for PM developers. Developers have to manually inspect the execution to identify the root cause of the problem. In addition, most of the existing state-of-the-art testing tools require heavy user annotations to detect bugs without visible symptoms such as segmentation faults.

In this paper, we present robustness as a sufficient correctness condition to ensure that program executions are free from bugs resulting from missing flushes. We develop an algorithm for checking robustness and have implemented this algorithm in the PSan tool. PSan can help developers both identify silent data corruption bugs and localize bugs in large traces to the problematic memory operations that are missing flush operations. We have evaluated PSan on a set of concurrent indexes, persistent memory libraries, and two popular real-world applications. We found **48 bugs** in these benchmarks that **17** of them were not reported before.

CCS Concepts: • **Hardware Memory and dense storage**; • **Software and its engineering → Software verification and validation**;

Keywords: Persistent Memory, Robustness, Software Verification, Debugging, Testing

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523723

ACM Reference Format:
Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. 2022. Checking Robustness to Weak Persistency Models. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523723

1 Introduction

Persistent memory (PM) revolutionizes the storage-memory hierarchy [46, 78, 86]. This technology became commercially available with Intel's release of Optane DC Persistent Memory [45]. Persistent memory interfaces with the processor via the memory bus similar to DRAM, providing byte-addressable storage access to programs via processor load and store instructions. This enables PM to provide programs with a new level of performance by enabling them to manipulate data directly without needing heavyweight OS system calls. The low latency and durability of PM have spurred the development and redesign of file systems [17, 23, 53, 58, 59, 73, 93, 96, 98, 99], databases [3, 20, 60, 74, 81], log-based systems [16, 29, 42, 52, 66, 67], key-value stores [15, 39, 54, 95, 97, 102, 104], and concurrent DRAM indexes [12, 14, 63, 83, 92, 101, 105] for persistent memory.

Designing crash-consistent PM programs is especially challenging because the cache system is volatile and its contents vanish upon a failure, *e.g.*, a system crash or a power failure. Processor manufacturers have introduced new instructions such as `CLWB` and `SFENCE` on x86 [44], and `DC CVAP` on ARM [2], to force cache lines to be written back to persistent memory. Developers of PM programs need to carefully use these instructions since a missing flush instruction can make a program vulnerable to crash consistency bugs.

Researchers have taken two primary approaches to improve PM reliability. First, there is a body of work on developing high-level abstractions such as transactional libraries [8, 11, 18, 30–32, 47, 57, 68, 75, 89, 94, 100, 103], locks [5, 13, 41, 49, 69], or synchronization-free regions [35] to hide the complexity of using such instructions, but these abstractions come at a performance cost and their implementations are still susceptible to crash consistency bugs. Second, researchers have developed testing/checking frameworks [22, 28, 37, 38, 40, 48, 55, 62, 70–72, 79, 80] to find and fix performance problems (*e.g.*, redundant flushes and

fences) and crash consistency bugs (*e.g.*, missing fences and flushes).

Testing tools suffer from two major drawbacks. First, to detect persistency bugs, they require test cases that can expose an execution error such as a segmentation fault or an assertion failure. The issue is that not all bugs cause such visible symptoms. Some of these tools require user annotations to catch bugs that do not lead to a program failure. Writing annotations not only incurs a burden on users but also is error-prune itself. Consequently, such tools can report false positives that originate from users' mistakes in using annotations. Second, in most cases, when a bug causes an execution to crash, it can be difficult to locate what part of the execution contains the bug. In fact, a recent study [79] on 26 bugs reported by Intel's *pmemcheck* tool shows that these bugs took on average 23 days and a maximum of 66 days to fix. These results highlight that diagnosing persistency bugs demands arduous human efforts.

1.1 Correctness Criteria for Flush Operations

Bugs in the uses of flush and drain operations can be trivially eliminated by making stores become persistent in the same order that they become visible to other threads. Strict persistency [84] is such a persistency model that ensures that the "persistency memory order is identical to volatile memory order". Most hardware persistent memory specifications do not provide strict persistency. However, Intel has developed an optional new feature called enhanced *Asynchronous DRAM Refresh* (eADR) that relies on stored power to flush the contents of the cache to persistent memory during a power failure. Consequently, eADR-enabled persistent memory provides strict persistency. However, eADR functionality cannot be relied upon because it requires the system vendor to provide additional stored energy hardware such as a battery. Due to these specialized requirements on system vendors, it is expected that many Intel PM systems will not provide strict persistency for the foreseeable future according to our email discussions with Intel engineers.

As a result, PM developers must explicitly use *flush instructions* (or similar mechanisms) to ensure that program executions under weak persistency semantics are correct. *Our key observation is that the typical correct usage of flush instructions in PM programs ensures that program executions under weak persistency semantics are equivalent to those under strict persistency semantics.* Building on this observation, we define a new notion of correctness, *robustness*, for programs under weak persistency in terms of their equivalence to post-crash executions under strict persistency. A program is robust to a weak persistency model if, for any crash events, each post-crash execution of the program under that weak persistency model is equivalent to some post-crash execution after some crash event under the strict persistency model. Robustness is a *sufficient criterion* to assure correct usage of flush and drain operations—adding more flush and drain operations to a robust program will not alter the set of possible post-crash executions. Robustness is *not* a necessary condition because programs may (1) be tolerant of reading stale values, *e.g.*, counters that only need to be approximately correct, or (2) use other mechanisms like checksums to detect and discard inconsistent data after reading it.

In general, robustness does *not* require a developer to insert flush operations immediately after every store. For example, consider a PM program in which a new node is added to a persistent singly-linked list. Stores to the new node are not visible to post-crash executions unless a *commit store* to the `next` field of some existing node in the linked list adds the new node to the list before the crash. The program is robust as long as these stores are flushed before the commit store is performed. *This pattern of using a commit store is typically how developers write PM programs, and robustness precisely captures the pattern.*

```
1   void addChild(node *ptr, char * data) {
2     childNode * tmp = alloc_child();
3     tmp->data = data;
4     clflush(tmp, sizeof(childNode));
5     ptr->child = tmp;
6     clflush(&ptr->child, sizeof(childNode *));
7   }
8
9   char * readChild(node *ptr) {
10    if (ptr->child != NULL) {
11      return ptr->child->data;
12    }
13    return NULL;
14  }
```

Figure 1. An example of execution being robust to the x86 persistency model, where the pre-crash execution crashes before line 6, and the post-crash execution executes the `readChild` method on the same node.

Example. To illustrate, consider the example from Figure 1 on the x86 persistency model. Suppose that execution of the `addChild` method crashes immediately before line 6 and that after the crash the program executes the `readChild` method on the same node. There are two possible post-crash executions: (1) the post-crash execution that results from the pre-crash execution where the store of the reference to the `child` field was flushed, and (2) the post-crash execution that results from the pre-crash execution where the store of the reference was *not* flushed. The first post-crash execution is equivalent to the post-crash execution under strict persistency where the pre-crash execution crashes after the store in line 5. The second post-crash execution is equivalent to the post-crash execution under strict persistency where the pre-crash execution crashes before the store in line 5. Since all post-crash executions of this program under the weak persistency model are equivalent to some post-crash execution under strict persistency, this program is robust.

Relation to Prior Work. Robustness to weak persistency models builds on a rich literature of defining the correctness

of concurrent code by relating concurrent executions to other executions. In the context of weak memory models, a program is robust [7, 61, 76, 82] against a weak memory model if all of the program's executions under the weak memory model are permitted under the sequential consistency model.

Table 1. Comparison with other tools; robustness subsumes ordering heuristics/conditions used in existing tools.

Tool	Persistent Order
PSan	Robustness
Witcher [28]	Dependence heuristic
PMDebugger [22]	User annotations
PMTest [72]	User annotations
XFDetector [71]	Commit store annotations
Jaaru [37]	Crash/assertion failure
Yat [62]	Crash/assertion failure
Agamotto [80]	Does not check order
Pmemcheck [55]	Does not check order
PMFuzz [70]	Just fuzzes input, uses Pmemcheck or XFDetector for checking
Hippocrates [79]	Does not repair ordering bugs

Robustness provides a rigorous foundation that subsumes prior work that relied on heuristics or annotations to check whether stores are persisted in the correct order. Note that in the comparison with prior work and throughout this paper, we only focus on **ordering bugs that are result of missing/misplaced flush and fence instructions**. Prior tools are able to identify other types of bugs such as performance bugs and the bugs resulting from stores being issued in an improper order. PSan does not attempt to find those types of bugs, and our comparison does not focus on them. Table 1 summarizes the approaches other tools take to checking the order of PM stores. All these conditions are essentially instances of robustness violations. Witcher [28] relies on heuristic inference rules that use control and data dependencies to detect stores that are not made persistent in the correct order due to missing flushes and fences. PMTest [72] and PMDebugger [22] rely on programmers to explicitly annotate ordering constraints, e.g., that store `x=1` is persisted before store `y=1`. PMDebugger also has some built-in oracles that can find some bugs without heavy annotations. XFDetector [71] requires that ordering constraints are specified implicitly by annotating a set of commit variables, otherwise it can report false positive. Jaaru [37] and Yat [62] only detect ordering bugs when the program crashes or asserts, and developers must manually localize the bug. Pmemcheck [55] and Agamotto [80] only check that stores are flushed and do not check the order they are flushed in. Our comparison with prior work is based on set of benchmarks that overlap between their evaluation and PSan's evaluation.

In addition to these general-purpose debugging tools, there is a rich literature on systematic transforms for lock-free data structures to use persistent memory [5, 19, 21, 27, 50, 92]. Most of these constructive approaches leverage different techniques to deduce flush and fence instructions for lock-free programs. More recently, Mirror [27] keeps two copies of the data in both DRAM and persistent memory. Load operations in Mirror only access DRAM, but store operations update both DRAM and persistent memory. While this design enables Mirror to not require persistency barriers after load operations, it incurs substantial memory overhead. Israelavitz et al. [50] introduce the notion of *durable linearazibility* to data-race-free programs to become crash consistent. *Durable linearazibility* is implemented as a set of transformation rules, which preserve the original happens-before ordering for persistent memory. While these constructive approaches suffice to ensure robustness, they may inject unnecessary fence and flush instructions. PSan can determine that weaker transformations also preserve robustness in several cases: (1) To avoid reasoning about the challenging corners of weak memory models, many implementations might follow the standard advice to use only release/acquire or SC atomics even when weaker atomics would suffice. But with stronger atomics, prior work [19, 50] would generate unnecessary fence instructions. (2) These transformations do not consider that consecutive writes to the same cache line eliminate the need for flush/fence operations between the writes. (3) If there are multiple relaxed stores to the same cache line, some of these techniques will cause a flush to be generated after each store. (4) If some persistent memory locations are used as temporary storage and are never read from after a crash, prior approaches would force stores to those locations to have flushes. (5) An existing RMW operation may suffice to serve as the needed fence instruction. In Section 3, we describe the notion of robustness as a sufficient requirement to guarantee crash consistency in lock-free frameworks.

PSan can save significant manual effort compared to repairing bugs without a tool. While Hippocrates [79] automatically implements bug fixes, PSan can suggest bug fixes to developers, for example, where there needs to be a flush inserted for a specific store and it must be done before another specific store is executed. However, Hippocrates only detects and corrects bugs where a flush is missing and cannot fix bugs in which stores may be persisted in an incorrect order. Such bugs commonly happen when developers delay flushes until the end of an update, overlooking the possibility that the stores could persist in the wrong order. PSan's bug fixes ensure that stores are persisted and that they are persisted in the same order as the happens-before relation. There are also inter-thread persistency bugs in which a thread performs a store and stops before its flush instruction, but another thread reads from that store, performs another store based on the read value and then persists the later store (*e.g.*, the execution in Figure 7). PSan is the only tool to our knowledge that will suggest the correct fix of fixing this bug in the second thread. PSan largely complements the work on Hippocrates of implementing interprocedural fixes. PSan requires no ordering annotations, reducing developer burden and eliminating the potential for missed bugs or false

alarms due to incorrect annotations. Moreover, robustness is sufficient to guarantee the absence of missing flush or drain operations. As shown in our evaluation, PSan found *17 new bugs that were previously unknown.*

1.2 PSan: Checking Robustness with Constraints

We develop PSan, a tool that dynamically checks robustness for programs under the x86 persistency model and reports violations in a fully automated fashion. For a given execution, PSan can detect *all persistency bugs due to ordering issues* in that execution. Our definition of an ordering bug is a bug that result from stores being persisted in an order that is different from their happens-before order. These bugs can be corrected by the addition of flush and/or fence operations. Finding other types of bugs is not the focus of the paper. In this work, we focus on the x86 persistency model, while our ideas are generally applicable to other weak persistency models as well. Given a crash event and a post-crash execution, PSan computes a set of strictly persistent executions whose pre-crash executions are consistent with the post-crash execution. If this set becomes empty, *i.e.*, such a strictly persistent execution does not exist, PSan finds a robustness violation.

Our key insight is that we can efficiently compute this set of consistent pre-crash executions under strict persistency by *reasoning about the interval in which an equivalent strictly persistent pre-crash execution must have crashed using constraints.* In particular, each load in the post-crash execution that reads from a store *s* in the pre-crash execution under the x86 persistency model constrains where an equivalent strictly persistent execution may crash—the crash point must be somewhere between the store *s* and the next store to the same memory location. If this set of constraints is unsatisfiable, there is no equivalent strictly persistent execution.

```
1   x = 1;
2   y = 1;
3   x = 2;
4   y = 2;
```

(a) Pre-crash execution.

```
1   r1 = x;
2   r2 = y;
```

(b) Post-crash execution.

Figure 2. A weakly-persistent execution that reads `r1 = 1` and `r2 = 2` is not robust.

To illustrate, consider the executions in Figure 2, which shows a single-threaded program executed under a weak persistency model. If `r1 = 1`, we know that an equivalent strictly persistent execution must have crashed after the assignment `x = 1` but before the assignment `x = 2`. If `r2 = 2`, then we know that an equivalent strictly persistent execution must have crashed after the assignment `y = 2`. These two constraints are not simultaneously satisfiable, and therefore this execution is *not* robust.

Next, we extend this approach to support multi-threaded programs. The key idea is that PSan determines whether there is an equivalent trace that can be produced by selecting different (but compatible) crash points for different threads. Our idea for implementing this is to have the robustness analysis compute per-thread crash intervals and ensure that these intervals describe a prefix of the pre-crash execution that is closed under the happens-before relation.

Robustness enables PSan to infer the exact program line with a missing flush or drain operation. Each robustness violation involves an earlier store that was not made persistent and a later store that was made persistent—the earlier store is missing a flush operation. For instance, for the execution in Figure 2, PSan determines a flush instruction must be inserted after `x = 2` to fix the robustness violation.

This paper makes the following contributions:

1. **Robustness:** It defines robustness, a sufficient correctness condition for the placement of flush and drain operations in persistent memory programs.
2. **Detecting Robustness Violations:** It presents an approach that uses robustness to identify persistency bugs that may not have visible symptoms.
3. **Bug Localization:** It presents an algorithm that localizes bugs in PM programs to the specific stores where flush and drain operations should be inserted.
4. **Bug Fixes:** It presents an algorithm for translating robustness violations into bug fixes. PSan's bug fixes ensure that stores are persisted in the correct order.
5. **Implementation and Evaluation:** We implemented PSan with a full simulation of $Px86_{sim}$ semantics with different modes and strategies to support complex, real-world programs. We evaluated PSan on CCEH, FAST_FAIR, the RECIPE persistent memory indexes, the PMDK library, as well as two popular industrial applications Redis and memcached. PSan found 48 persistency bugs that 17 of them have never been reported before; so far 7 bugs have been confirmed.

2 Background on x86 Persistency Model

This section briefly overviews the Intel-x86 persistency semantics following the $Px86_{sim}$ model in Raad *et al.* [88]. For our purposes, the differences between Raad *et al.* [88] and Khyzha [56] are minor and do not affect our work. The $Px86_{sim}$ semantics capture the behavior Intel implemented and intended for the architecture. They differ slightly from the semantics in Intel's manual due to mistakes in precisely specifying the intended behavior in the documentation. Since the $Px86_{sim}$ semantics do not formalize non-temporal store semantics, we do not support them.

The x86 architecture provides instructions to force the cache to write data back to persistent storage. The three such instructions are: (1) the flush cache line instruction `clflush` that flushes a cache line, (2) the optimized flush cache line instruction `clflushopt`, and (3) the cache line write back instruction `clwb`. Each of these instructions takes as input the address of the cache line and flushes that line.

A key difference between these instructions is how they can be reordered across other instructions. The `clflush`

instruction is inserted into the store buffer just like store instructions, and when it exits the store buffer it causes the cache line to be flushed to persistent memory. The `clflushopt` instruction is also inserted into the store buffer, but it can be reordered across store instructions to other cache lines, `clflush` instructions to other cache lines, and other `clflushopt` instructions. The `clflushopt` instruction cannot be reordered across `mfenceit` or locked `RMW` instructions. The store fence instruction `sfence` also orders `clflushopt` instructions relative to `clflush`, `clflushopt`, `clwb`, and store instructions. We refer to `mfence`, `sfence`, and locked `RMW` instructions collectively as *drain operations*. The `clwb` instruction writes back the contents of the cache line and may not evict it from the cache and thus potentially has better performance. However, from a semantics perspective, the `clflushopt` instruction is identical to the `clwb` instruction [88], and thus we treat them identically.

3 Preliminaries

Recovery mechanisms often rely on specific persistency orderings in the program's execution. Failure to enforce such orderings can lead to data corruption and loss after a system crash. There are different memory persistency models that allow different persistency orderings to be observed by recovery procedures [24, 26, 36, 51, 84, 85]. Among them, *strict persistency* is the most conservative and intuitive model which integrates memory persistency into memory consistency [84]. Under strict persistency, the recovery procedure observes the memory in an equivalent state as a separate processor would under the memory consistency model.

This section formalizes the strict persistency model and the robustness condition. We will introduce some notations first. Given a PM program P, an *execution* of the program is the complete trace of memory operations, fences, cache flush operations, and crash events in executing the program P. We denote an execution as *Exec*. A crash is denoted by C, and we use C_i to denote the i-th crash event in an execution *Exec*. The crash events partition an *Exec* as:

$$Exec = e_1 C_1 e_2 C_2 ... e_n C_n e_{n+1},$$

where n is the number of crash events in *Exec*. We say that each e_i is a *sub-execution* of *Exec* and write $e_i \subset Exec$ to denote this relation. This terminology describes the scenario where a process crashes and then recovers multiple times.

Memory operations include load and store operations: a load is denoted as $ld\langle x, \tau\rangle$, and a store is denoted as $st\langle x, \tau\rangle$, where x is the memory location and τ is the thread executing the operation. Since we only care about which store a load reads from, the actual values that a store writes and a load reads from are not important in our context, and we omit them in the notation. When the memory location or the thread that performs that operation is irrelevant in the context, we will omit them in the notation and write $ld\langle x\rangle$ or $st\langle x\rangle$.

3.1 Strict Persistency

We formalize strict persistency in terms of the total store order (TSO) memory model. If in an execution, a store $st\langle x\rangle$ is ordered before another store $st\langle y\rangle$ in the x86-TSO memory consistency order, *i.e.*, $st\langle x\rangle$ takes effect in the cache before $st\langle y\rangle$, we write $st\langle x\rangle \xrightarrow{tso} st\langle y\rangle$ to represent *TSO-ordered-before* relationship between these two stores. Under strict persistency, the volatile memory order and persistent memory order are identical. That means that for two stores $st\langle x\rangle$ and $st\langle y\rangle$, if $st\langle x\rangle \xrightarrow{tso} st\langle y\rangle$, then $st\langle x\rangle$ is persisted before $st\langle y\rangle$.

3.2 Robustness Condition

One can naïvely implement strict persistency by inserting flush operations after every memory access. Developers typically do not do this because this strategy can incur unacceptable overheads. However, the strict persistency model can be utilized as a correctness condition in using a weaker persistency model, *e.g.*, *relaxed persistency*. Recall from Section 1, a program under weak persistency models can behave the same as the program under strict persistency without requiring flush operations after every load and store. Building on this idea, we next define *robustness* for a single execution in terms of *multi-threaded prefixes*.

Definition 1. *Let e be a sub-execution of some execution $Exec = e_1 C_1 ... C_n e_{n+1}$. We define a multi-threaded prefix of e as a subset G of operations in e such that G is closed under the happens-before relation over stores and the sequenced-before relation, and that stores in G maintain the same TSO order as in e.*

We define a multi-threaded prefix of Exec as a subset F of operations in Exec such that F is closed under the reads-from relation, and $F = e_1'$ where e_1' is a multi-threaded prefix of e_1; or $F = e_1' C_1 ... C_k e_{k+1}'$, where $k < n$ and e_i' is a multi-threaded prefix of e_i for all $1 \le i \le k+1$.

Definition 2. *An execution Exec with n crash events is robust if for all $1 < i \le n+1$, there exists a multi-threaded prefix F_i of $H_i = e_1 C_1 ... C_{i-1} e_i$ such that*

1. *the last sub-execution of F_i is e_i ;*
2. *if $st\langle x\rangle \in e_j$ is read from by a load in later sub-executions for some $j \le i$, then $st\langle x\rangle \in F_i$.*
3. *F_i is a valid execution under strict persistency in which all stores in e_j have committed to the cache before the crash C_j occurs for all $1 \le j < i$.*

Each multi-threaded prefix of *Exec* preserves the sequenced-before and reads-from relations, the happens-before relation over stores, and the TSO order in *Exec*. The *happens-before relation over stores* is defined in Section 3.4. Definition 2 requires that each portion of the execution in *Exec* up to some crash event (*i.e.*, $H_i = e_1 C_1 ... C_{i-1} e_i$) is equivalent to the multi-threaded prefix F_i in that the last sub-execution of F_i has the same behavior as that of H_i. Intuitively, it means at any point of the execution, the most recent

sub-execution has the same behavior as that of some strictly persistent execution. For store operations $st\langle x\rangle \in e_j$ that are not included by any of multi-threaded prefixes F_i's, where $i > j$, their effects are either not written to the persistent memory or not read from by loads in sub-executions later than e_j in *Exec*. However, if $st\langle x\rangle \in e_j$ is read from by loads in later sub-executions, then it must be included in all of F_i's where $j \leq i$. Lastly, each F_i is an execution under strict persistency when all stores in F_i are written to the persistent memory.

The following definition presents the notion of *robustness* for programs:

Definition 3. *A program P is robust to a weak persistency model if every execution Exec of program P is robust.*

3.3 Persistent Lock-Free Data Structures

As prior studies note [5, 19, 50, 92], *strict persistency* guarantees recoverability for lock-free data structures. Thus, robustness is a sufficient criterion to correctly port lock-free data structures to persistent memory. The key observation is that a crash of a lock-free data structure under the strict persistency model is equivalent to a crash-free execution in which one set of threads runs the pre-crash execution and stop at their respective crash locations and then after those threads stop, the second set of threads runs the post-crash execution. Lock-freedom guarantees progress for such execution, and thus robustness plus lock-freedom suffices to ensure crash consistency.

The robustness definition is generic and can be applied to any program, including single-threaded, log-free, and lock-based multi-threaded programs, in addition to lock-free programs. For persistency strategies other than lock-free programs, robustness can still be a useful tool for finding any potential flush/fence bugs even though robustness is not sufficient to guarantee crash consistency for such programs. Broadly speaking, the domain of applicability for PSan is PM programs that attempt to persist data across crashes.

3.4 Clock Vectors and Sequence Numbers

Our algorithm for checking robustness requires tracking the happens-before relation and the TSO order, so we will cover some basics on how we use clock vectors to track the happens-before relation [25] over stores and sequence numbers to track the TSO order.

Clock vectors have an initial value $\bot_{CV}$, a union operator $\cup$, a comparison operator $\leq$, and a per-thread increment operator inc_τ that is invoked every time a thread performs a store. These are defined as follows:

$$\bot_{CV} = \lambda\tau.0,$$
$$CV_1 \cup CV_2 \triangleq \lambda\tau.max(CV_1(\tau), CV_2(\tau)),$$
$$CV_1 \leq CV_2 \triangleq \forall\tau.CV_1(\tau) \leq CV_2(\tau),$$
$$inc_\tau(CV) = \lambda u.\ \text{if } u == \tau \text{ then } CV(u) + 1 \text{ else } CV(u).$$

States:

$Tid \triangleq \mathbb{Z}$ $\quad CV \triangleq Tid \to \mathbb{Z}$ $\quad \mathbb{CV} \triangleq Tid \to CV$ $\quad \mathbb{SCV} \triangleq store \to CV$
$seq : \mathbb{Z}$ $\quad \mathbb{SEQ} \triangleq store \to \mathbb{Z}$

[LOAD]
$$\frac{st\langle x, \tau_s\rangle \xrightarrow{rf} ld\langle x, \tau\rangle \qquad \mathbb{CV}' = \mathbb{CV}[\tau \mapsto \mathbb{CV}(\tau) \cup \mathbb{SCV}(st\langle x, \tau_s\rangle)]}{\langle \mathbb{CV}, \mathbb{SCV}, \mathbb{SEQ}, seq\rangle \Rightarrow^{ld\langle x,\tau\rangle, st\langle x,\tau_S\rangle} \langle \mathbb{CV}', \mathbb{SCV}, \mathbb{SEQ}, seq\rangle}$$

[STORE ISSUE]
$$\frac{\mathbb{CV}' = \mathbb{CV}[\tau \mapsto inc_\tau(\mathbb{CV}(\tau))] \quad \mathbb{SCV}' = \mathbb{SCV}[st\langle x, \tau\rangle \mapsto \mathbb{CV}'(\tau)] \quad \mathbb{SEQ}' = \mathbb{SEQ}[st\langle x, \tau\rangle \mapsto 0]}{\langle \mathbb{CV}, \mathbb{SCV}, \mathbb{SEQ}, seq\rangle \Rightarrow^{st\langle x,\tau\rangle} \langle \mathbb{CV}', \mathbb{SCV}', \mathbb{SEQ}', seq\rangle}$$

[STORE COMMIT]
$$\frac{seq' = seq + 1 \qquad \mathbb{SEQ}' = \mathbb{SEQ}[st\langle x, \tau\rangle \mapsto seq']}{\langle \mathbb{CV}, \mathbb{SCV}, \mathbb{SEQ}, seq\rangle \Rightarrow^{st\langle x,\tau\rangle} \langle \mathbb{CV}, \mathbb{SCV}, \mathbb{SEQ}', seq'\rangle}$$

[CRASH]
$$\frac{seq' = 0 \qquad \mathbb{CV}' = \texttt{reset}(\mathbb{CV})}{\langle \mathbb{CV}, \mathbb{SCV}, \mathbb{SEQ}, seq\rangle \Rightarrow^{crash} \langle \mathbb{CV}', \mathbb{SCV}, \mathbb{SEQ}, seq'\rangle}$$

Figure 3. Algorithm for updating clock vectors that track the happens-before relation over stores and sequence numbers that record the TSO order.

Each store has a clock vector associated with it, and each thread has its own clock vector. We define a map $\mathbb{CV}$ that maps a thread identifier to the thread's clock vector and write $\mathbb{CV}(\tau)$ to denote the clock vector for thread τ. We define $\mathbb{SCV}$ as a map from a store to store's clock vector.

In order to keep track of the TSO order, we define a sequence number for each store operation, representing the order the stores take effect in the cache. We maintain a map $\mathbb{SEQ}$ that maps a store to its sequence number.

Figure 3 presents the algorithm for updating clock vectors and sequence numbers. The sequence counter *seq* is a strictly increasing global counter, which is initialized to 0. The [LOAD] rule applies when a load reads from a store and merges the clock vector of the thread performing the load with the clock vector of the store being read from. The [STORE ISSUE] rule applies when a thread τ performs a store, *i.e.*, inserting the store into the thread's store buffer. It updates the thread τ's clock vector using the inc_τ operator, initializes the store's clock vector, and initializes the store's sequence number to 0. The [STORE COMMIT] rule applies when a store leaves its store buffer. It increments the counter *seq* by 1, and assigns the store's sequence number as the counter's current value. When a crash event occurs, the [CRASH] rule resets the sequence number counter *seq* to 0 and the map $\mathbb{CV}$ to an empty map. For two stores $st\langle x\rangle$ and $st\langle y\rangle$ in the same sub-execution, if $\mathbb{SCV}(st\langle x\rangle) \leq \mathbb{SCV}(st\langle y\rangle)$, then the store $st\langle x\rangle$ happens before the store $st\langle y\rangle$.

Given a store $st\langle x, \tau\rangle$ and its clock vector $\mathbb{SCV}(st\langle x, \tau\rangle)$, we define the *clock of the store* as $\mathbb{SCV}(st\langle x, \tau\rangle)(\tau)$, the τ-th component of its clock vector. We will use a helper function `getcl` throughout the paper that takes a store as input and

returns the clock of the store. Because the inc_τ operator is only applied to thread τ, and a load operation in thread τ may only update components of thread τ's clock vector other than the τ-th component, every store in a thread has a unique clock. *Note that the clock of stores orders stores in a single thread in a sub-execution by when they are issued, while the sequence number orders stores in a sub-execution by when they commit their values to the cache.*

4 Basic Ideas

PSan builds on the open-source Jaaru infrastructure [37] for simulating the x86 persistent memory model. Jaaru's frontend takes as input the PM program source and generates an instrumented binary. The instrumented binary is executed by Jaaru, and Jaaru generates an execution trace. Jaaru assumes as input a set of test cases that explore a program's PM data structures. These can potentially be generated by existing test data generation tools [1, 4, 6, 9, 10, 33, 34, 64, 70, 87, 90, 91]. Jaaru generates executions of PM programs, and then PSan checks these executions for robustness violations using Jaaru's plugin interface.

PSan reports robustness violations to users, which can help users find bugs in the uses of flush and drain operations. PSan can also be helpful for debugging known bugs. When an assertion violation or other error is detected, Jaaru provides developers with the trace. This trace can contain millions of operations, and it can be difficult to understand which ones are relevant to the crash. PSan can quickly relate bugs in the uses of flush and fence operations to the individual memory operation that is either missing a flush operation or has an incorrectly placed flush operation. PSan then suggests to users one or more bug fixes.

4.1 Checking Equivalence

PSan's approach for identifying equivalent strictly persistent executions computes a set of strictly persistent executions that are consistent with the behavior of the weakly persistent execution thus far. The basic approach relies on computing *potential crash intervals* that describe the set of equivalent strictly persistent executions. We model potential crash intervals using constraints. If the constraints become unsatisfiable, then no such equivalent strictly persistent pre-crash execution exists and the program is not robust. At this point, PSan would then report a robustness violation.

During the post-crash execution of the program, PSan updates the constraints to compute a potential crash interval for the pre-crash execution. The constraint set is initially empty to indicate that any strictly persistent pre-crash execution is consistent with the behavior of the initially empty post-crash execution. Each load in the post-crash execution potentially narrows the set of strictly persistent pre-crash executions that are consistent with the post-crash execution.

For each potential crash interval constraint, the beginning of a range corresponds to a unique store, and so does the end of a range. We use the *clocks of stores* defined in Section 3.4 to mark the beginnings and ends of ranges. *Note that although the clocks of stores are used to mark the beginning and end ranges of potential crash interval constraints, a constraint really means that an equivalent strictly persistent execution should crash after the store corresponding to the beginning of the range commits to the cache and before the store corresponding to the end of the range commits to the cache.*

```
1    x = 1;
2    y = 2;
3    x = 3;
4    y = 4;
5    x = 5;
```

(a) Pre-crash execution

```
r1 = y;
r2 = x;
```

(b) Post-crash execution

Figure 4. An example of non-robust program with missing flush and drain operations. x and y are initialized to 0.

Figure 4 presents an example that we will use to present our basic approach. The left column in Figure 4 shows the code of the pre-crash execution and the right column shows the code of the post-crash execution. Section 6.1 elaborates on how PSan inserts crash points in the program. The clocks of stores in the pre-crash execution are listed on the left of Figure 4-a.

Consider an execution in which `r1 = 2` and `r2 = 5`. Figure 5 shows such an example and illustrates the process of checking for an equivalent strictly persistent execution. At the beginning of the post-crash execution, the potential crash interval constraint set is empty. After the post-crash execution reads 2 from y, this constrains an equivalent strictly persistent pre-crash execution to have crashed after the assignment `y = 2` commits to the cache, but before `y = 4` commits to the cache. Therefore, the potential crash interval constraint $[2, 4)$ is added to the constraints. When the post-crash execution reads 5 from x, this constrains an equivalent strictly persistent pre-crash execution to have crashed after the store `x = 5` commits to the cache and implies the potential crash interval constraint $[5, \infty)$ should be added to the constraints. However, the combination of the prior interval constraint $[2, 4)$ and the new interval constraint $[5, \infty)$ is unsatisfiable. Thus, there is no equivalent pre-crash execution

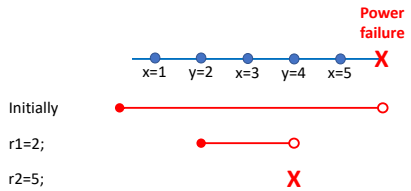

Figure 5. Constraints for execution of code in Figure 4 where `r1 = 2` and `r2 = 5`.

under strict persistency. This execution is possible under the x86 persistency model because there is no flush and drain operation for y after y = 4.

4.2 Supporting Threads

We next discuss the basic ideas of how we generalize our approach for updating potential crash interval constraints to the multi-threaded context.

4.2.1 Per-Thread Crash Intervals. Naïvely applying potential crash interval constraints to a multi-threaded execution trace using TSO order is overly restrictive. Figure 6 presents an example that demonstrates the issue with this approach. We assume that the store x = 1 is TSO ordered before the store y = 1 in the pre-crash execution and this could potentially be observed by pre-crash threads. Consider the execution where r1 = 0 and r2 = 1.

This execution is robust, because it is equivalent to a strictly persistent execution where thread τ_1 does not perform any operation, thread τ_2 executes y = 1, and then the program crashes. Then the post-crash execution of the strictly persistent execution would read r1 = 0 and r2 = 1.

In the naïve approach, we inspect the trace of the pre-crash execution to determine where an equivalent execution should crash. Since clocks of stores do not order stores in different threads, sequence numbers have to be used in the constraints. r1 = x = 0 yields the constraint $[0, seq_{\texttt{x = 1}})$, because an equivalent strictly persistent execution must crash before the store x = 1. Similarly, r1 = y = 1 yields the constraint $[seq_{\texttt{y = 1}}, \infty)$. However, the combination of the two constraints $[0, seq_{\texttt{x = 1}}) \wedge [seq_{\texttt{y = 1}}, \infty)$ is unsatisfiable.

To solve this issue, each thread requires its own potential crash interval constraints, since each thread can make different progress when a program crashes. Therefore, we define potential crash interval constraints $\mathbb{C}$ as a map from a thread identifier to a potential crash interval constraint for the thread. The map $\mathbb{C}$ is satisfiable if and only if each interval constraint in its range is satisfiable. Each $\mathbb{C}(\tau)$ is initially empty.

4.2.2 Persistency Closure under Happens-Before. Another aspect of the simple approach in Section 4.1 is that it only updates potential crash interval constraints based on the TSO ordering between stores at the same memory location. This simple approach is not enough to detect robustness violations in the multi-threaded context. More specifically, if a store is made persistent in a robust execution, then all stores that are read from and that happen before this store must also be made persistent. However, the simple approach cannot detect robustness violations in executions where a store that has been read from and that happens before a persistent store is not made persistent.

Figure 7 presents an example that shows such robustness violations. This example is also interesting because it shows that simply adding flush operations after each store is not always sufficient to guarantee robustness. Figure 7-(a) and 7-(b) present the pre-crash execution code for thread τ_1 and thread τ_2. Figure 7-(c) shows the code for the post-crash execution. We assume that both x and y are initialized to 0, and that they reside in different cache lines. Consider the execution where thread τ_1 executes x = 1 and is paused by the operating system before executing the corresponding flush. Then, thread τ_2 reads r1 = x = 1, stores y = r1 = 1, and flushes y. If the program crashes at this point, the post-crash execution can read r2 = 0, but r3 = 1. Such an execution is not feasible under strict persistency.

When the post-crash execution reads r2 = 0, it can be inferred that the thread τ_1 of an equivalent strictly persistent execution must have crashed before the store x = 1 commits to the cache. Therefore, we have $\mathbb{C}(\tau_1) = [0, \texttt{getcl(x = 1)})$. Similarly, when the load r3 = y reads from the store y = r1, it can be inferred that the thread τ_2 of the equivalent strictly persistent execution must have crashed after the store y = r1 commits to the cache, and $\mathbb{C}(\tau_2) = [\texttt{getcl(y = r1)}, \infty)$. At this point, both $\mathbb{C}(\tau_1)$ and $\mathbb{C}(\tau_2)$ are satisfiable, failing to detect the robustness violation in this execution.

This execution exhibits a robustness violation because the store y = r1 is made persistent, but the store x = 1 that happens before it is not. This robustness violation can be fixed if x = 1 is forced to be persistent before y = r1 by adding a flush instruction after the load r1 = x in thread τ_2.

It is worth noting that if we require that stores that are not read from and are TSO ordered before a persistent store be made persistent in a robust execution, then this condition is too strong in that it would classify some robust executions as non-robust. For example, the execution in Figure 6 is robust, but x = 1 is not persistent even though it is TSO ordered before y = 1, and y = 1 is made persistent.

4.3 Implications for Updating Constraints

In this section, we will present implications for updating potential crash interval constraints in executions with a single crash event. Every time a load $ld\langle x\rangle$ in the post-crash execution reads from a store $st\langle x, \tau_1\rangle$ in the pre-crash execution, PSan updates constraints based on the following implications:

1. Observed stores must have executed: When a load $ld\langle x\rangle$ in the post-crash execution reads from a store $st\langle x, \tau_1\rangle$ in the pre-crash execution, we can infer that an equivalent strictly persistent execution must have crashed after the store $st\langle x, \tau_1\rangle$ commits for thread τ_1:

$$st\langle x, \tau_1\rangle \xrightarrow{rf} ld\langle x\rangle \Rightarrow \mathbb{C}(\tau_1) := [\texttt{getcl}(st\langle x, \tau_1\rangle), \infty) \wedge \mathbb{C}(\tau_1). \quad (4.1)$$

2. Newer stores must have not executed: If there is a second store $st\langle x, \tau_2\rangle$ that is TSO ordered after the $st\langle x, \tau_1\rangle$, then the equivalent strictly persistent execution must have

```
x = 1;
flush x;
```

(a) Thread τ_1 in pre-crash execution

```
y = 1;
flush y;
```

(b) Thread τ_2 in pre-crash execution

```
r1 = x;
r2 = y;
```

(c) Post-crash execution

Figure 6. x and y reside in different cache lines and are initialized to 0. We assume that in the pre-crash execution, a third thread observes that x = 1 is TSO ordered before y = 1. Can the execution read r1 = 0 and r2 = 1?

```
x = 1;
flush x;
```

(a) Thread τ_1 in pre-crash execution

```
r1 = x;
y = r1;
flush y;
```

(b) Thread τ_2 in pre-crash execution

```
r2 = x;
r3 = y;
```

(c) Post-crash execution

Figure 7. An example of just adding flushes after stores is not always sufficient to provide robustness. x and y are initialized to 0. x and y reside in different cache lines. Can the execution read r1 = 1, r2 = 0, and r3 = 1?

crashed before $st\langle x, \tau_2\rangle$ commits for thread τ_2, because otherwise, $ld\langle x\rangle$ would read from $st\langle x, \tau_2\rangle$ in the strictly persistent execution instead:

$$st\langle x, \tau_1\rangle \xrightarrow{rf} ld\langle x\rangle \wedge st\langle x, \tau_1\rangle \xrightarrow{tso} st\langle x, \tau_2\rangle \Rightarrow \mathbb{C}(\tau_2) := [0, \texttt{getcl}(st\langle x, \tau_2\rangle)) \wedge \mathbb{C}(\tau_2). \quad (4.2)$$

3. An execution prefix is closed under happens before: If there is any store $st\langle y, \tau_3\rangle$ that happens before $st\langle x, \tau_1\rangle$ in the pre-crash execution, then the equivalent strictly persistent execution must have crashed after $st\langle y, \tau_3\rangle$ commits for thread τ_3, because $st\langle y, \tau_3\rangle$ must have been executed before $st\langle x, \tau_1\rangle$:

$$st\langle x, \tau_1\rangle \xrightarrow{rf} ld\langle x\rangle \wedge st\langle y, \tau_3\rangle \xrightarrow{hb} st\langle x, \tau_1\rangle \Rightarrow \mathbb{C}(\tau_3) := [\texttt{getcl}(st\langle y, \tau_3\rangle), \infty) \wedge \mathbb{C}(\tau_3). \quad (4.3)$$

4.4 Supporting Multiple Crash Events

So far, our discussion has only focused on executions with one crash event. In an execution *Exec* with n crash events, the execution has $n + 1$ sub-executions. Therefore, each crash event should have its own potential crash interval constraints, and we define map C that maps a sub-execution e to the potential crash interval constraints for the crash event immediately following the sub-execution. For a complete execution, C would map the last sub-execution to an empty set of constraints, because there is no crash event after the last sub-execution.

In an ongoing execution, we refer to the sub-execution after the last crash event that has occurred so far as the current sub-execution. When a load in the current sub-execution reads from a store in a previous sub-execution e, PSan would update the potential crash interval constraints for the sub-execution e. However, if a load in the current sub-execution reads from a store in a previous sub-execution that does not immediately precede the current sub-execution, then some additional constraints would apply, because the store that is read from cannot be overwritten by any store in sub-executions later than e. We present these additional constraints in Section 5.1.

```
x = 1;
y = 1;
```

(a) sub-execution e_1

```
y = 2;
r = x;
```

(b) sub-execution e_2

```
s = y;
```

(c) sub-execution e_3

Figure 8. A single-threaded program with three sub-executions. Both sub-executions e_1 and e_2 are followed by crash events. x and y reside in different cache lines and are initialized to 0. The execution reads r = 0 and s = 1.

Figure 8 presents an example of a single-threaded execution with two crash events and three sub-executions. Although this example is single-threaded, the general idea applies to multi-threaded programs. Both sub-executions e_1 and e_2 are followed by crash events. The load r = x = 0 reads from the initial value of x, and the load s = y = 1 reads from the store y = 1 in the first sub-execution.

Right after the crash event following sub-execution e_2, the execution is robust so far. Since the program is single-threaded, we will omit the thread identifier in the notation. The load r = x = 0 updates $C(e_1)$ as $C(e_1) = [0, \texttt{getcl}(\texttt{x = 1}))$, because the first sub-execution of an equivalent strictly persistent execution must crash before x = 1 commits to the cache, and $C(e_2)$ has no constraints. Then when the load s = y reads from y = 1, $C(e_1)$ becomes $[0, \texttt{getcl}(\texttt{x = 1})) \wedge [\texttt{getcl}(\texttt{y = 1}), \infty)$, because the first sub-execution of the equivalent execution must crash after y = 1 commits to the cache. Also, $C(e_2)$ becomes $[0, \texttt{getcl}(\texttt{y = 2}))$, because the second sub-execution of the equivalent execution must crash before y = 2 commits to the cache. Otherwise, the older store y = 1 would be overwritten. However, the constraints in $C(e_1)$ are not satisfiable, and such equivalent execution does not exist.

Note that a misinterpretation of the constraint $C(e_2) = [0, \texttt{getcl(y = 2)})$ would suggest that the second sub-execution should be empty. Then since `r = x` is not executed, $C(e_1)$ becomes $[\texttt{getcl(y = 1)}, \infty)$, resulting in satisfiable constraints. However, this is not the case. First of all, the constraint $C(e_2) = [0, \texttt{getcl(y = 2)})$ suggests that the second sub-execution of the equivalent execution should crash before `y = 2` *commits to the cache*, not necessarily before `y = 2` is executed. Second, even if the second sub-execution of the equivalent execution crashes before `y = 2` is executed, it does not affect the original weakly persistent execution that was used to derive the map C, and so we do not remove the implications of the load `r = x` from $C(e_1)$.

5 Algorithm

```
a ∈ Reg    v ∈ Val    τ ∈ TId
Prog  ::= TId ⟶(fin) Com
Com   ::= Exp | PCom
        | let a := Com in Com
        | if (Com) then {Com} else {Com}
        | repeat Com
PCom  ::= load(x) | store(x,Exp) | CAS(x,Exp,Exp)
        | FAA(x,Exp) | mfence | sfence
        | flushopt x | flush x
Exp   ::= v | a | Exp op Exp
```

Figure 9. A simple concurrent programming language.

We present our algorithm for detecting robustness violations with respect to the simple concurrent language used by Px86$_{\text{sim}}$ [88], as described in Figure 9. We assume that `Reg` is a finite set of registers (local variables), `Val` is a finite set of values, and `TId` $\subseteq \mathbb{N}$ is a finite set of thread identifiers. An expression `Exp` is either a register, a value, or the result of applying an arithmetic operation on two expressions. We define a multi-threaded program `Prog` as a function mapping each thread to the sequential program that the thread executes. The sequential fragment of the language is given by the `Com` grammar, which includes primitive commands `PCom`, expressions, assignments to local variables, conditional statements, and loops. The `load(x)` denotes an atomic read from location `x`, and the `store(x,Exp)` denotes an atomic write to location `x`. The `CAS(x,Exp,Exp)` denotes the atomic compare-and-swap. The `FAA(x,Exp)` denotes the atomic fetch-and-add operation. Our analysis treats RMW operations in the same fashion as a load immediately followed by a store. The `mfence` and `sfence` denote a memory fence and a store fence, respectively. Lastly, `flushopt` and `flush` denote persist instructions, persisting the cache line where location `x` resides.

5.1 Operational Semantics

Figure 10 presents our algorithm in operational semantics as an extension to the Px86$_{\text{sim}}$ operational model. *We present a correctness proof for the algorithm in the Supplemental Material of our paper.* Before performing the analysis in Figure 10, the algorithm in Figure 3 for computing clock vectors and sequence numbers is applied to the corresponding operations. After the analysis in Figure 10, we extend the transitions for the Px86$_{\text{sim}}$ operational model [88].

We use the following notations in the algorithm:

- $\texttt{getexec}(st\langle x, \tau\rangle)$ returns the sub-execution that contains the store $st\langle x, \tau\rangle$;
- $\texttt{next}(st\langle x, \tau\rangle, e)$ returns the smallest set of stores that includes (1) the first store to the location x in each thread that is TSO ordered after store $st\langle x, \tau\rangle$ in the sub-execution $\texttt{getexec}(st\langle x, \tau\rangle)$ and (2) the first store to the location x in each thread in any sub-execution that follows $\texttt{getexec}(st\langle x, \tau\rangle)$ and precedes e.
- $\texttt{nextop}(i)$ returns the instruction that follows i in the execution;
- $\texttt{top}(Exec)$ returns the last sub-execution in $Exec$, *i.e.*, the current sub-execution;
- C maps a sub-execution e to its mapping $\mathbb{C}_e$ from threads to potential crash intervals.

States:

$$C \triangleq Exec \to \mathbb{C} \qquad \mathbb{C} \triangleq \texttt{TId} \to \textit{Constraint List}$$

[LOAD-PREV]

$$\frac{\begin{array}{c} st\langle x,\tau\rangle \xrightarrow{rf} ld\langle x\rangle \quad \hat{e} = \texttt{getexec}(st\langle x,\tau\rangle) \quad e_c = \texttt{top}(Exec) \\ ld\langle x\rangle \in e_c \quad \hat{e} \neq e_c \quad \{st\langle x,\tau_1\rangle_1, ..., st\langle x,\tau_n\rangle_n\} = \texttt{next}(st\langle x,\tau\rangle, e_c) \\ \forall i \in \{1,...,n\}.\hat{e}_i = \texttt{getexec}(st\langle x,\tau_i\rangle_i), \sigma_i = \mathbb{SCV}(st\langle x,\tau_i\rangle_i)(\tau_i) \\ C_0 = C[\hat{e} \mapsto \{\langle \tau', C(\hat{e})(\tau') \wedge [\mathbb{SCV}(st\langle x,\tau\rangle)(\tau'), \infty)\rangle \mid \tau' \in \texttt{TId}\}] \\ \forall i \in \{1,...,n\}.C_i = C_{i-1}[e_i \mapsto C_{i-1}(\hat{e}_i)[\tau_i \mapsto C_{i-1}(\hat{e}_i)(\tau_i) \wedge [0, \sigma_i))]] \end{array}}{\langle ld\langle x\rangle, C\rangle \implies \langle \texttt{nextop}(ld\langle x\rangle), C_n\rangle}$$

Figure 10. Semantics for checking robustness violations.

We only check for robustness violations when a load in the current sub-execution reads from a store in a previous sub-execution. The clock vector $\mathbb{SCV}(st\langle x, \tau\rangle)$ has information about the last store in each of the other threads that happens before $st\langle x, \tau\rangle$, because for each $\tau' \neq \tau$, $\mathbb{SCV}(st\langle x, \tau\rangle)(\tau')$ is exactly the clock of the last store in thread τ' that happens before $st\langle x, \tau\rangle$. When $\tau' = \tau$, $\mathbb{SCV}(st\langle x, \tau\rangle)(\tau')$ is the clock of $st\langle x, \tau\rangle$. Therefore, C_0 is the result of applying implications 4.1 and 4.3. Then the last line in Figure 10 iteratively applies the implication 4.2 for each store in the set $\texttt{next}(st\langle x, \tau\rangle, e_c)$.

5.2 Suggesting Fixes for Robustness Violations

We next discuss how PSan suggests fixes for robustness violations. In general, there are two ways to fix a robustness violation. The first is to use flush and/or drain operations to force the cache to write back a cache line to persistent memory. The second is to leverage the existing cache coherence mechanism to enforce the desired ordering by locating a pair of stores for which an ordering violation is observed on the same cache line.

Each identified bug is defined by a pair of stores: the first store is ordered earlier in the happens-before relation than

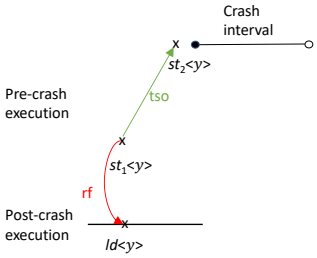

Figure 11. Reading from a store that is too old.

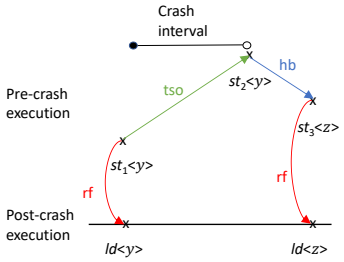

Figure 12. Reading from a store that is too new.

the second store, but only the second store was persisted and observed by loads in post-crash executions. PSan gives this pair of stores to users. The bug fix is a little more complicated because these stores could potentially be in different threads and it is possible, for example, that the thread that executes the first store stops immediately after the store, and some other thread reads from this store and later performs a second store. In this case, we cannot prevent this robustness violation by adding a flush after the first store since that thread stops. We have to fix this bug by adding a flush after the load. Thus, PSan defines a fix as a set of flush intervals that cover operations that happen between the pair of stores.

There are two cases in which a robustness violation may be reported — the first case is when the most recent load reads from a store that is too old to be consistent with the strict persistency model, and the second case is when the most recent load reads from a store that is too new. We first discuss the first case in more detail.

Reading from Too Old of Store. Figure 11 presents a robustness violation that occurs when the most recent load $ld\langle y\rangle$ reads from a store $st_1\langle y\rangle$ that is too old. This occurs because the program is missing a flush on some newer store $st_2\langle y\rangle$ to the same memory location. Our algorithm detects this when the presence of the later store $st_2\langle y\rangle$ causes the algorithm to move the end of the crash interval backward past the beginning of the interval. A single load can potentially reveal multiple stores $st_2\langle y\rangle$ that are missing flush operations. This set of stores are the stores $st\langle x, \tau_i\rangle_i$ such that the computation of the maps C_i in the load rule of our operational semantics computes a new unsatisfiable interval.

The fix for this bug is to insert a flush and a drain that happen after the store $st_2\langle y\rangle$ and happen before the beginning of some potential crash interval. Specifically, PSan computes for each thread a potential flush window that starts at the first operation in that thread that happens after $st_2\langle y\rangle$ and continues until the beginning of that thread's crash interval. We distinguish the interval for the thread that performed $st_2\langle y\rangle$, and call this interval the primary fix interval. While all the suggested fixes will eliminate the robustness violation, we believe the primary fix interval is typically the desired fix. However, the primary fix interval may not always exist as seen in the scenario in Figure 7 in which a thread crashes between performing a store and flushing and draining the store, but a second thread observes the presence of that store and then persist stores of their own. In this case, the primary fix interval would be empty, and PSan would produce an alternate interval for that second thread.

Alternatively, to fix this bug by colocating fields on the same cache line, PSan would compute the store that sets the beginning of the crash interval shown in Figure 11. The store $st_2\langle y\rangle$ must be made persistent before that store, and thus developers must modify the memory layout to ensure that both stores write to the same cache line.

Reading from Too New of Store. Figure 12 presents an execution in which the most recent load $ld\langle z\rangle$ reads from a store $st_3\langle z\rangle$ that is too new to be consistent with the strict persistency model. This occurs because a previous load $ld\langle y\rangle$ read from a store that was too old since some store $st_2\langle y\rangle$ was missing an appropriate flush operation. Our algorithm detects this violation when the store $st_3\langle z\rangle$ causes the beginning of the crash interval to be move forward past the end of the crash interval.

The fix for this bug is to insert a flush and a drain operation such that $st_2\langle y\rangle$ happens before the flush and drain operation and the flush and drain operation happens before $st_3\langle z\rangle$. We must first compute the store $st_2\langle y\rangle$. We implement this by recording for each crash interval the store that sets that its end. If the store at the end of an interval happens before $st_3\langle z\rangle$, then this store is a store $st_2\langle y\rangle$. There can be multiple such stores. For each thread and each store $st_2\langle y\rangle$, we report an interval such that $st_2\langle y\rangle$ happens before operations in the interval and operations in the interval happen before $st_3\langle z\rangle$. We distinguish the interval for the thread that executed $st_2\langle y\rangle$ as a primary fix. Similar to the previous case, the primary interval is typically the desired fix. But the interval can be empty if $st_2\langle y\rangle$ happens before $st_3\langle z\rangle$ only if some other thread in the pre-crash execution reads from $st_2\langle y\rangle$.

Alternatively, to fix this bug by colocating fields on the same cache line, the store $st_2\langle y\rangle$ must be made persistent before the store $st_3\langle z\rangle$, and thus developers must modify the memory layout to ensure that both x and y are located on the same cache line.

Implementation. The algorithm as described only detects robustness violations on the current execution. Our implementation is built on the Jaaru model checker and at every load, it selects a store for that load to read from. Before selecting a store for a load to read from, PSan checks each possible store that the load can read from to see if it will

create a robustness violation. PSan reports any detected violation. A straightforward application of the algorithm can only detect a single robustness violation in an execution. PSan can detect multiple robustness violations in a single execution by forcing loads to read from stores that do not cause robustness violations. This allows PSan to continue the execution past the first detected robustness violation so that PSan can detect additional robustness violations.

6 Evaluation

In this section, we evaluate the usefulness and effectiveness of PSan in finding persistency bugs in a set of benchmarks. We start by describing the benchmarks and the configuration of our system. Then, we describe our evaluation methodology and analyze the bugs found by PSan. Finally, we discuss our observations from our experiments.

System Setup. PSan was implemented atop the open-source Jaaru model checker for persistent memory [37]. Our experiments were carried out on an Ubuntu 18.04 machine with a 6 core 3.7 GHz Intel i7-8700K processor and 32GB RAM.

6.1 Methodology

We first tested PSan on the RECIPE [63] collection of PM indexes based on B+-trees, tries, radix trees, and hash tables [43, 63, 77]. CCEH [77] is an efficient hash table for persistent memory. FAST_FAIR [43] is an efficient implementation of B+-tree. We used all of these data structures (*i.e.*, P-ART, P-BwTree, P-CLHT, and P-Masstree) in our experiment except P-HOT because it does not compile with LLVM. We recompiled each of these programs with Jaaru's LLVM compiler pass to instrument memory accesses and cache operations. Each program has a test driver that performs operations on the data structure.

We also evaluated PSan on three popular real-world frameworks and applications: PMDK [47], Memcached [20], and Redis [60]. PMDK is the most active open-source library for accessing persistent memory and is developed and maintained by *Intel*. This well-tested library simplifies accessing persistent memory and debugging PM applications. PMDK incorporates a wide range of libraries from direct APIs to access persistent memory, *i.e.*, *libpmem*, to object transactional APIs, *i.e.*, *libpmemobj*. Similar to prior works, we used five PMDK data structure examples to evaluate our tool, BTree, CTree, RBTree, Hashmap atomic, and Hashmap tx. Memcached is a high-performance distributed memory caching system implemented by *Lenovo* to use persistent memory. This in-memory key-value store uses low-level *libpmem* APIs to efficiently store data in persistent memory. To evaluate PSan with Memcached, we implemented a client that issues insertion and lookup requests. Redis is an industrial high-performance cache server and in-memory database developed by *Intel*. Redis is capable of caching data on DRAM and persisting it in persistent memory through PMDK's transactional APIs. Similar to Memcached, we implemented our own client to modify and lookup data.

PSan supports two different exploration strategies that target different types of applications: (1) random search mode in which PSan explores random executions with random crash points and (2) model checking mode in which PSan systematically inserts crashes before each fence-like operation and after the last operation of the program and then, explores all values that each load can read.

In our data structure benchmark experiments, *i.e.*, CCEH, FAST_FAIR, and RECIPE, we used both model checking mode as well as random execution mode with 10,000 executions. We used a similar configuration for evaluating PMDK examples. However, for Redis and Memcached we just used random mode since these benchmarks require an outside client, which makes model checking challenging.

6.2 Bug Detection

During our experiment, PSan found a total of 48 bugs in benchmarks, and 17 of them were not reported by any of the state-of-the-art testing frameworks. 13 bugs were related to robustness violations in the memory management code of the benchmarks. Table 2 reports only violations/bugs that are not in the memory allocation code due to space constraints. Violations with * are known bugs. We reported these violations to the developers of these tools and so far, developers of CCEH and FAST_FAIR have confirmed these violations are real bugs. The RECIPE developers acknowledged the reported bugs but did not fix them, since these bugs are related to memory allocators and garbage collectors, and the code for memory allocators has to change regardless. For each of these violations, PSan reports the variable that needs a flush instruction and the precise range where the flush needs to be inserted. In our experiment, we simply applied PSan's suggestions and reran the program until no robustness violations were reported.

After analyzing each reported robustness violation, we categorized them into three different types:

Missing Flushes/Fences. Table 2 presents all memory locations that participated in robustness violations. Note that some of these violations refer to different usages of the same variable in different functions or executions. All the robustness violations except #9 are due to missing fence/flush instructions. 12 robustness violations caused program failures in our experiment and the rest had no visible manifestations. We examined the code and verified for each violation that the bugs could cause data corruption, data loss, or memory leak.

Cache-line Alignment Bugs. PSan identified one robustness violation that would likely not be fixed with flush or fence instructions, *i.e.*, #9 in Table 2, in FAST_FAIR benchmark. In this benchmark, the `header` class is used at the

Table 2. Robustness violations.

#	Benchmark	Field	Cause of Robustness Violation
1	CCEH	*sema*	locking *sema* in *Segment::Insert*
2	CCEH	*sema*	unlocking *sema* in *Segment::Insert*
3*	CCEH	*key*	writing to *key* in *Segment::Insert*
4*	CCEH	*Directory::_[i]*	writing to *_[i]* in *CCEH* constructor
5*	CCEH	*Directory::_*	writing to *_* in *CCEH* constructor
6*	CCEH	*CCEH*	writing to *CCEH* fields in *CCEH* constructor
7	FAST_FAIR	*switch_counter*	incrementing it in *page::insert_key*
8	FAST_FAIR	*last_index*	updating it in *page::insert_key*
9	FAST_FAIR	*dummy*	unalignment caused by *header* class
10	FAST_FAIR	*entry::ptr*	writing to *ptr* in *insert_key*
11*	FAST_FAIR	*entry::ptr*	writing to *ptr* in *entry* constructor
12*	FAST_FAIR	*leftmost_ptr*	writing to *leftmost_ptr* in *header* constructor
13*	FAST_FAIR	*btree::root*	writing to *root* in *btree* constructor
14	P-ART	*typeVersion-LockObsolete*	locking it in *N::writeLockOrRestart*
15	P-ART	*typeVersion-LockObsolete*	locking it in *N::lockVersionOrRestart*
16	P-ART	*typeVersion-LockObsolete*	unlocking it in *N::writeUnlock*
17	P-ART	*nodesCount*	updating it in *DeletionList::add*
18	P-ART	*N16::keys*	updating it in *N16::insert*
19	P-ART	*N16::count*	updating it in *N16::insert*
20*	P-ART	*N4::keys*	updating it in *N4::insert*
21*	P-ART	*N4::children*	updating it in *N4::insert*
22*	P-ART	*deletionLists*	writing to *deletionLists* in *Epoche* constructor
23*	P-ART	*Tree::root*	writing to *root* in *Tree* constructor
24	P-BwTree	*next*	updating it in *GrowChunk* function
25*	P-BwTree	*gc_metadata_p*	writing to *gc_metadata_p* address in *GCMetaData::PrepareThreadLocal*
26*	P-BwTree	*gc_metadata_p*	writing to content of *gc_metadata_p* in *GCMetaData::PrepareThreadLocal*
27*	P-BwTree	*tail*	writing to *tail* in *AllocationMeta*
28*	P-BwTree	*epoch_manager*	writing to *epoch_manager* in *BwTree* constructor
29*	P-CLHT	*version_list*	writing to *clht_t::version_list* in *clht_gc_thread_init*
30*	P-CLHT	*num_buckets*	writing to *clht_t::num_buckets* in *clht_hashtable_create*
31*	P-CLHT	*table*	writing to *clht_t::table* in *clht_hashtable_create*
32	PMDK	*PMEMobjpool*	*memcpy* operation on pool object in *libpmemobj* library
33	PMDK	*ulog*	storing *ulog* in *libpmemobj* library
34	PMDK	*ulog_entry_base*	*memcpy* in applying modifications on a single *ulog_entry_base*
35	PMDK	*ulog_entry_base*	applying *ULOG_OPERATION_OR* on a single *ulog_entry_base*

beginning of the page class [43]. The problem is that the developers did not carefully consider C++ object layout semantics. They neglected the fact that a word-aligned 8-bit field has 8 bits of padding following it when it is followed by the 16-bit field. Consequently, the header class is larger than expected and results in the rest of the page class not having the expected cache line alignment and thus breaks code that relies on stores to different fields in the page class writing to the same cache line to maintain ordering.

Memory Management Bugs. In addition to robustness violations in Table 2, PSan found 9 more robustness violations in P-ART and 4 more in P-BwTree. PSan found these violations in memory management code such as garbage collection and the memory allocation implementation. As mentioned in the paper [63], the RECIPE benchmark implementations focused on providing a platform to measure performance and did not fully implement the crash recovery and memory management components. These 13 reported robustness violations are real robustness violations, but there are more significant bugs in the code than just missing flush and drain instructions; fixing them requires more fundamental changes in the design of the memory management component.

While robustness is a sufficient condition for an execution to be free of bugs related to missing flush and fence operations, PSan, like all dynamic tools, can miss reporting a flush/fence bug if it does not explore an execution that reveals the missing flush/fence.

6.3 Performance

We next ran 100 random executions with both PSan and Jaaru, the underlying model checker, to report the overhead of PSan. Table 3 reports the average times taken to run one random execution for each of the benchmarks. PSan and Jaaru have comparable execution times because checking robustness introduces minimal overheads. This table also reports the total number of executions that PSan explored to find all reported bugs. Overall, it takes less than a minute to explore all executions used to find bugs for a benchmark and an average of 13.1 seconds per benchmark.

Table 3. Execution times for PSan and Jaaru (the underlying model checking infrastructure). PSan incurs minimal overhead compared to Jaaru.

Benchmark	Jaaru Time (s)	PSan Time (s)	# total executions
CCEH	0.050	0.051	1068
Fast_Fair	0.036	0.038	19
P-ART	0.045	0.047	348
P-BwTree	0.032	0.032	93
P-CLHT	0.142	0.143	6
P-Masstree	0.035	0.037	93

6.4 Discussion

Harmless Violations. While the proposed approach to correctness can handle many persistent data structures, there are design patterns that can cause false positives. These design patterns include *link-and-persist* [21], *pointer tagging* [65], and checksums. These design patterns all allow post-crash executions to safely observe low-level violations of robustness without compromising high-level safety. In particular, during our evaluation, we observed that PM programs that use checksums can safely read from data that has only been made partially persistent because the checksum will fail and the program will safely discard the data. Programs that use checksums are not robust by our prior definition because their post-crash executions may observe robustness violations. However, the values read by the loads that cause the robustness violations are discarded when a checksum check fails. PSan supports these patterns by using annotations. In particular, PSan uses these annotations to postpone the processing of the loads from a given checksum computation until the checksum validation completes successfully. If the checksum validation fails, those loads

operations are discarded. In Table 2, violations #33 - #35 are caused by checksums validating redo logs. These violations are harmless because the program safely discards the data when checksum fails, while such harmless violations could be avoided by checksum annotations.

Comparison with Other Tools. Of the six tools that can potentially detect ordering violations, only two tools, Jaaru [37] and Witcher [28], are both available and do not require us to annotate the expected ordering properties to be checked. Thus, we limited our comparison to Jaaru and Witcher. Jaaru found 18 persistency bugs in CCEH, FAST_FAIR, and RECIPE benchmarks, of which 15 are related to missing proper persistency mechanisms. Jaaru's developers had to manually examine each bug and reason about the execution traces to fix each persistency bug. On the contrary, PSan automatically reported the exact variable that needed a flush instruction and the precise location where the flush needed to be inserted. PSan reported 20 bugs that were not identified by Jaaru. Witcher reported 4 ordering bugs in our evaluated benchmarks and for each bug, Witcher requires developers' manual efforts to reason about the root cause of intricate crash states. One of these bugs was also found by PSan. PSan did not report the rest of these bugs since Witcher used different test driver programs to exercise the RECIPE benchmarks, while we used the programs from Jaaru's distribution of the RECIPE. While we would like to perform an evaluation on the exact same programs, this is problematic. We could not run PSan's programs on Witcher, because Witcher's distribution does not contain support for finding correctness bugs. We could not run Witcher's programs on PSan, because they do not have any code that runs after a crash. PSan reported 31 bugs that could not be found by Witcher.

Note that not being able to find all bugs reported by other tools on the same set of benchmarks evaluated by PSan and these tools is primarily due to the implementations of these tools that have particular dependencies on program versions, inputs, environments, etc., *not* a limitation of using robustness as a correctness criterion. As discussed earlier, robustness subsumes all ordering-related constraints and PSan should report all ordering bugs for given executions that are caused by missing flush and fence instructions.

7 Conclusion

This paper presents robustness, a sufficient correctness condition for the use of flush and drain operations in persistent memory programs. We implemented the first tool that leverages this condition to localize persistency bugs in the program and suggests fixes. PSan found 48 bugs (including 17 new bugs) in 13 popular PM benchmarks.

Acknowledgments

We would like to thank our shepherd, Satish Narayanasamy, and the anonymous reviewers for their thorough and insightful comments that helped us substantially improve the paper.

This work is supported by the National Science Foundation grants CNS-1703598, OAC-1740210, CNS-1763172, CNS-1907352, CNS-2006437, CCF-2006948, CNS-2007737, CCF-2102940, CNS-2128653, and CNS-2106838, as well as ONR grants N00014-16-1-2913 and N00014-18-1-2037.

References

[1] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. 2010. Using Bounded Model Checking for Coverage Analysis of Safety-Critical Software in an Industrial Setting. *J. Autom. Reasoning* 45 (12 2010), 397–414.

[2] ARM. 2021. Arm Architecture Reference Manual Armv8, for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest.

[3] Joy Arulraj and Andrew Pavlo. 2017. How to Build a Non-Volatile Memory Database Management System *(SIGMOD '17)*. 1753–1758.

[4] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-Directed Dynamic Automated Test Generation *(ISSTA '11)*. 12–22.

[5] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-Volatile Memory *(ISMM 2016)*. 55–67.

[6] Bernard Botella, Mickael Delahaye, Stephane Hong-Tuan-Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. 2009. Automating structural testing of C programs: Experience with PathCrawler. 70–78.

[7] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding Robustness against Total Store Ordering, Luca Aceto, Monika Henzinger, and Jiří Sgall (Eds.). 428–440.

[8] Bill Bridge. 2021. Nvm-direct library. https://github.com/oracle/nvm-direct.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs *(OSDI'08)*. 209–224.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (Dec. 2008), 38 pages.

[11] Daniel Castro, Paolo Romano, and João Barreto. 2018. Hardware transactional memory meets memory persistency *(IPDPS '18)*. 368–377.

[12] Hokeun Cha, Moohyeon Nam, Kibeom Jin, Jiwon Seo, and Beomseok Nam. 2020. B3-Tree: Byte-Addressable Binary B-Tree for Persistent Memory. *ACM Trans. Storage* 16, 3, Article 17 (July 2020), 27 pages.

[13] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency *(OOPSLA '14)*. 433–452.

[14] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797.

[15] Xianzhang Chen, Edwin H.-M. Sha, Ahmad Abdullah, Qingfeng Zhuge, Lin Wu, Chaoshu Yang, and Weiwen Jiang. 2017. UDORN: A design framework of persistent in-memory key-value database for NVM. 1–6.

[16] 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 *(ASPLOS '20)*. 1077–1091.

[17] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. 2018. HiNFS: A Persistent Memory File System with Both Buffering and Direct-Access. *ACM Trans. Storage* 14, 1, Article 4 (April 2018), 30 pages.

[18] 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. 105–118.

[19] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. 2020. *Lazy Release Persistency.* 1173–1186.
[20] Inc. Danga Interactive. 2018. Memcached. https://github.com/lenovo/memcached-pmem.
[21] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures *(USENIX ATC '18)*. 373–385.
[22] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs *(ASPLOS 2021)*. 503–516.
[23] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory *(EuroSys '14)*. Article 15, 15 pages.
[24] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas, and Stefanos Kaxiras. 2021. TSOPER: Efficient Coherence-Based Strict Persistency. 125–138.
[25] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection *(PLDI '09)*. 121–133.
[26] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. 2020. Persist Level Parallelism: Streamlining Integrity Tree Updates for Secure Persistent Memory. 14–27.
[27] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. *Mirror: Making Lock-Free Data Structures Persistent.* 1218–1232.
[28] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores *(SOSP 2021)*. 100–115.
[29] Ning Gao, Zhang Liu, and Dirk Grunwald. 2017. DTranx: A SEDA-based Distributed and Transactional Key Value Store with Persistent Memory Log. arXiv:1711.09543 [cs.DB]
[30] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-Compatible Persistent Transactions *(PLDI 2020)*. 59–74.
[31] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. 2020. go-pmem: Native Support for Programming Persistent Memory in Go. 859–872.
[32] Ellis Giles, Kshitij Doshi, and Peter Varman. 2017. Continuous Checkpointing of HTM Transactions in NVM *(ISMM 2017)*. 70–81.
[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223.
[34] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. *Automated Whitebox Fuzz Testing.* Microsoft, San Diego, California, USA.
[35] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions *(PLDI 2018)*. 46–61.
[36] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. 652–665.
[37] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently Model Checking Persistent Memory Programs *(ASPLOS 2021)*. 415–428.
[38] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2022. Yashme: Detecting Persistency Races *(ASPLOS 2022)*. 830–845.
[39] Minjong Ha and Sang-Hoon Kim. 2020. InK: In-Kernel Key-Value Storage with Persistent Memory. *Electronics* 9, 11 (2020), 22 pages.
[40] Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety *(ASPLOS 2021)*. 429–442.
[41] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-Threaded Applications *(EuroSys '17)*. 468–482.
[42] Haixin Huang, Kaixin Huang, Litong You, and Linpeng Huang. 2018. Forca: Fast and Atomic Remote Direct Access to Persistent Memory. 246–249.
[43] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree *(FAST '18)*. 187–200.
[44] Intel. 2020. Third Generation Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-overview.html?wapkw=clwb.
[45] Intel. 2021. Memory Optimized for Data-centeric Workloads. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.
[46] Intel. 2021. Revolutionizing Memory and Storage. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.
[47] Intel Corporation. 2020. Persistent Memory Development Kit. https://pmem.io/pmdk/.
[48] Intel Corporation. 2021. Intel Inspector. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html.
[49] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging *(ASPLOS '16)*. 427–442.
[50] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model, Cyril Gavoille and David Ilcinkas (Eds.). 313–327.
[51] Jungi Jeong and Changhee Jung. 2021. PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency) *(ASPLOS 2021)*. 517–529.
[52] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient Hardware-Assisted Logging with Asynchronous and Direct-Update for Persistent Memory. 520–532.
[53] 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 *(SOSP '19)*. 494–508.
[54] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. 191–205.
[55] Tomasz Kapela. 2015. An introduction to pmemcheck (part 1) - basics. https://pmem.io/2015/07/17/pmemcheck-basic.html.
[56] Artem Khyzha and Ori Lahav. 2021. Taming X86-TSO Persistency. *Proc. ACM Program. Lang.* 5, POPL, Article 47 (Jan. 2021), 29 pages.
[57] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories *(ASPLOS '16)*. 399–411.
[58] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. 2017. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. 197–212.
[59] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System *(SOSP '17)*. 460–477.
[60] Redis Labs. 2020. Redis. https://github.com/pmem/redis.
[61] Ori Lahav and Roy Margalit. 2019. Robustness against Release/Acquire Semantics *(PLDI 2019)*. 126–141.
[62] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. 433–438.
[63] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes *(SOSP '19)*. 462–477.
[64] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs, Ganesh Gopalakrishnan and Shaz Qadeer

(Eds.). 609–615.

[65] Nan Li and Wojciech Golab. 2021. Brief Announcement: Detectable Sequential Specifications for Recoverable Shared Objects *(PODC'21)*. 557–560.

[66] Sumin Li and Linpeng Huang. 2020. LosPem: A Novel Log-Structured Framework for Persistent Memory. *J. Emerg. Technol. Comput. Syst.* 16, 3, Article 27 (May 2020), 17 pages.

[67] 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 *(ASPLOS '17)*. 329–343.

[68] 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 *(ASPLOS '17)*. 329–343.

[69] 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 *(MICRO-51)*. 258–270.

[70] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. PMFuzz: Test Case Generation for Persistent Memory Programs *(ASPLOS 2021)*. 487–502.

[71] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs *(ASPLOS '20)*. 1187–1202.

[72] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs *(ASPLOS '19)*. 411–425.

[73] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. 773–785.

[74] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory *(HotStorage'17)*. 4.

[75] Leonardo Marmol, Mohammad Chowdhury, and Raju Rangaswami. 2018. LibPM: Simplifying Application Usage of Persistent Memory. *ACM Trans. Storage* 14, 4, Article 34 (Dec. 2018), 18 pages.

[76] Yuri Meshman, Noam Rinetzky, and Eran Yahav. 2015. Pattern-Based Synthesis of Synchronization for the C++ Memory Model *(FMCAD '15)*. 120–127.

[77] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory *(FAST '19)*. 31–44.

[78] Dushyanth Narayanan and Orion Hodson. 2012. Whole-System Persistence *(ASPLOS XVII)*. 401–410.

[79] Ian Neal, Andrew Quinn, and Baris Kasikci. 2021. Hippocrates: Healing Persistent Memory Bugs without Doing Any Harm *(ASPLOS 2021)*. 401–414.

[80] Ian Neal, Ben Reeves, Ben Stoler, and Andrew Quinn. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application? 1047–1064.

[81] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud *(SOSP '11)*. 29–41.

[82] Peizhao Ou and Brian Demsky. 2015. AutoMO: Automatic Inference of Memory Order Parameters for C/C++11 *(OOPSLA 2015)*. 221–240.

[83] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory *(SIGMOD '16)*. 371–386.

[84] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. 265–276.

[85] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2015. Memory Persistency: Semantics for Byte-Addressable Nonvolatile Memory Technologies. *IEEE Micro* 35, 3 (2015), 125–131.

[86] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. *Proc. VLDB Endow.* 7, 2 (Oct. 2013), 121–132.

[87] Corina S. Pundefinedsundefinedreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing Nasa Software *(ISSTA '08)*. 15–26.

[88] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-x86 Architecture. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 11 (December 2019), 31 pages.

[89] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. 672–685.

[90] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C *(ESEC/FSE-13)*. 263–272.

[91] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex–White Box Test Generation for .NET, Bernhard Beckert and Reiner Hähnle (Eds.). 134–153.

[92] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory *(FAST'11)*. 5.

[93] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-System Interfaces to Storage-Class Memory *(EuroSys '14)*. Article 14, 14 pages.

[94] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory *(ASPLOS XVI)*. 91–104.

[95] 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 *(APSys '16)*. Article 18, 7 pages.

[96] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A file system for Storage Class Memory. 1–11.

[97] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems *(USENIX ATC '17)*. 349–362.

[98] Jian Xu and Steven Swanson. 2016. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories *(FAST'16)*. 323–338.

[99] 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 *(SOSP '17)*. 478–496.

[100] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. *Clobber-NVM: Log Less, Re-Execute More*. 346–359.

[101] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems *(FAST'15)*. 167–181.

[102] Baoquan Zhang and David H. C. Du. 2021. NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Tree with Accumulative Compaction. *ACM Trans. Storage* 17, 3, Article 23 (Aug. 2021), 26 pages.

[103] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. 897–912.

[104] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A Key-Value Store for Optane Persistent Memory *(EuroSys '21)*. 194–209.

[105] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 421–434.