UNIVERSITY OF CALIFORNIA,
IRVINE

Checking Robustness for Persistency Memory Programs

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical Engineering and Computer Science

by

Weiyu Luo

Dissertation Committee:
Professor Brian Demsky, Chair
Professor Rainer Dömer
Assistant Professor Sang-Woo Jun

2024

DEDICATION

To my parents and my family members who always supported me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to sincerely thank my advisor, Professor Brian Demsky, who played a key role in my Ph.D. journey. In past six years, I was lucky to benefit from his mentorship and experience that helped me grow and gain exceptional skill sets in software systems and programming languages. I sincerely thank him for selecting me to be one of his students and patiently teaching me how to think critically, do research, develop big systems, and present ideas. I would not be able to successfully finish my Ph.D. degree without his help and guidance.

I would like to thank professors and fellow researchers who helped me along the way. I thank Professor Aparna Chandramowlishwaran, Professor Rainer Dömer, Professor Isaac Scherson, and Professor Sang-Woo Jun for serving on my candidacy exam committee. I also thank Hamed Gorjiara, Rahmadi Trimananda, Alex Lee, and Simon Guo who have been my peer co-authors. Finally, I thank Peizhao Ou, Zachary Snyder, Ahmed Al Nahian, Seyed Amir Hossein Aqajari, Derek Yeh, Xiafa Wu, Keonho Lee, Conan Truong, and many others who have been my colleagues.

I would like to thank the developers of PMDK from Intel, Memcached, RECIPE, and Redis in particular Andy Rudoff, Piotr Balcer, Frank Hady, and Sekwon Lee that I extensively used their work in my research evaluation. I would like to thank the developers of LLVM and C++ language that I used these programming languages and compilers in my research in past 6 years.

Finally, I would like to thank University of California Irvine for granting me a one-year fellowship. Also, I would like to thank National Science Foundation grants CNS-1703598, OAC-1740210, CCF-2006948, CCF-2102940, and CCF-2220410. My doctoral research and the work presented in this dissertation would not be possible without the support from these institutions.

VITA

Weiyu Luo

EDUCATION

Doctor of Philosophy in Computer Engineering **2024**
University of California, Irvine *Irvine, California*

Master of Science in Computer Engineering **2021**
University of California, Irvine *Irvine, California*

Bachelor of Science in Mathematics **2018**
Pennsylvania State University *State College, Pennsylvania*

RESEARCH EXPERIENCE

Graduate Student Research **2008–2024**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2024**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

Checking Robustness to Weak Persistency Models **Jun 2022**
Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation

Stateful Dynamic Partial Order Reduction for Model Checking Event-Driven Applications that Do Not Terminate **Jan 2022**
Verification, Model Checking, and Abstract Interpretation: 23rd International Conference, VMCAI 2022

C11Tester: A Race Detector for C/C++ Atomics **Mar 2021**
Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems

SOFTWARE

PSan `https://plrg.ics.uci.edu/psan/`
A tool for checking robustness to weak persistency models

C11Tester `http://plrg.ics.uci.edu/c11tester/`
A tool for testing C/C++ atomics in real world code

ABSTRACT OF THE DISSERTATION

Checking Robustness for Persistency Memory Programs

By

Weiyu Luo

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Irvine, 2024

Professor Brian Demsky, Chair

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 thesis, we present robustness as a sufficient correctness condition to ensure that program executions are free from bugs resulting from missing flushes. We present two approaches to for checking robustness, one dynamic and one static. We develop an algorithm for checking robustness and have implemented this algorithm in the dynamic 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.

While dynamic tools can help developers find missing flush instructions, they typically require test cases that reveal the bug. Test cases can be onerous to write and can easily miss covering critical bug revealing executions. Therefore, we also present PMRobust, a static analysis that can ensure that persistent memory code is free from all missing flush bugs. PMRobust does not require any test cases and reports all missing flush bugs. PMRobust's analysis supports common persistent memory programming patterns and avoids reporting spurious bug reports for these patterns. We have evaluated PMRobust on persistent memory libraries and several persistent memory data structures. We have found a total of 80 bugs in popular PM benchmarks including 15 new bugs.

Chapter 1

Introduction and Background

1.1 Introduction

Persistent memory (PM) revolutionizes the storage-memory hierarchy [85, 93, 52]. This technology became commercially available with Intel's release of Optane DC Persistent Memory [51]. 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 [27, 65, 66, 80, 103, 106, 108, 109, 19, 59], databases [3, 67, 23, 81, 88], log-based systems [74, 18, 58, 73, 34, 48], key-value stores [17, 105, 107, 60, 112, 114, 45], and concurrent DRAM indexes [16, 111, 70, 90, 102, 116, 14] 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 [50], 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 [21, 13, 110, 35, 37, 75, 53, 64, 104, 10, 113, 99, 36, 82], locks [6, 15, 47, 55, 76], or synchronization-free regions [40] 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 [69, 61, 79, 78, 87, 43, 54, 77, 46, 25, 86, 33, 44] 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 [86] 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.

1.1.1 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 [91] 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 strict persistency. 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 1.1: 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 1.1 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.

1.2 Background on x86 Persistency Model

This section briefly overviews the Intel-x86 persistency semantics following the Px86_{sim} model in Raad *et al.* [96, 97]. For our purposes, the differences between Raad *et al.* [97] and Khyzha [62] 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.

Each core/thread on x86 has a store buffer that buffers stores to the cache to hide the store latency. Stores in the store buffer are written to the cache in order. The cache is volatile — a power loss event will cause cached data that has not been written back to persistent storage to be lost. Cache lines are written back to main memory non-deterministically when the cache needs the space for other data. The x86 architecture provides instructions to force the cache to write data back to persistent storage.

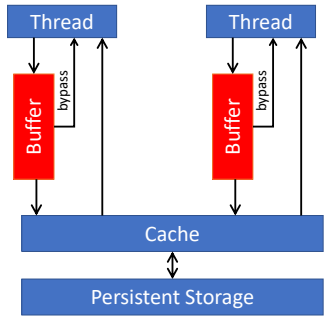


Figure 1.2: The x86-TSO storage system.

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

A key difference between these instructions is how they can be reordered across other instructions. Table 1.1 summarizes the instruction ordering constraints for persistent storage on x86-TSO. 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 immediately flushed to

Later in Program Order

Earlier in Program Order	Re	Wr	RMW	mf	sf	clfopt	clf
Read	✓	✓	✓	✓	✓	✓	✓
Write	✗	✓	✓	✓	✓	CL	✓
RMW	✓	✓	✓	✓	✓	✓	✓
mfence	✓	✓	✓	✓	✓	✓	✓
sfence	✗	✓	✓	✓	✓	✓	✓
clflushopt	✗	✗	✓	✓	✓	✗	CL
clflush	✗	✓	✓	✓	✓	CL	✓

Table 1.1: Reordering constraints in the $Px86_{sim}$. A ✓ indicates that the order between the two instructions is preserved, a ✗ indicates that the two instructions can be reordered, and a CL indicates that the order is preserved only if they both operate on the same cache line. 'mf', 'sf', 'clfopt', and 'clf' represent 'mfence', 'sfence', 'clflushopt', and 'clflush', respectively.

persistent memory. The `clflushopt` instruction is inserted into the store buffer also like store instructions, 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 `mfence` or locked `RMW` instructions. The store fence instruction `sfence` also orders `clflushopt` instructions relative to `clflush`, `clflushopt`, `clwb`, and store instructions. The `clwb` instruction only writes back the contents of the cache line and does not evict it from the cache and thus has better performance. However, from a semantics perspective, the `clwb` instruction is identical to `clflushopt` instruction [97], and thus we treat them identically in our discussions. While ARM has a persistency model [98], we are not aware of any commercially available persistent memory for ARM. Our basic approach should also be applicable to ARM. We expect that in the future standards bodies may develop persistent memory models to enable portable code much like language memory models such as C++11 [26]. Indeed, there has been some research on portable persistent memory models [63].

Chapter 2

Dynamic Approach: PSan

This section presents 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 2.1: A weakly-persistent execution that reads `r1 = 1` and `r2 = 2` is not robust.

To illustrate, consider the executions in Figure 2.1, 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 2.1, PSan determines a flush instruction must be inserted after `x = 2` to fix the robustness violation.

This chapter 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.

2.1 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 [92, 91, 28, 57, 41, 31]. Among them, *strict persistency* is the most conservative and intuitive model which integrates memory persistency into memory

consistency [91]. 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 $st\langle x\rangle$.

2.1.1 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 $st\langle y\rangle$.

2.1.2 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 1.1, 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 2.1. *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 $1 \leq i \leq k+1$.

Definition 2.2. *An execution Exec with n crash events is robust if for all $1 < i \leq n+1$, 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 $j \leq i$, then $st\langle x\rangle \in F_i$; and*

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 $1 \leq j < i$.*

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 2.1.4. Definition 2.2 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 2.3. *A program P is robust to a weak persistency model if every execution Exec of program P is robust.*

2.1.3 Persistent Lock-Free Data Structures

As prior studies note [102, 6, 56, 22], *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.

2.1.4 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 [29] 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:

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

Each store has a clock vector associated with it, and each thread has its own clock vector.

States:

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

[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]

$$\frac{\begin{array}{c}\mathbb{CV}' = \mathbb{CV}[\tau \mapsto inc_\tau(\mathbb{CV}(\tau))] \\ \mathbb{SCV}' = \mathbb{SCV}[st\langle x, \tau\rangle \mapsto \mathbb{CV}'(\tau)] \qquad \mathbb{SEQ}' = \mathbb{SEQ}[st\langle x, \tau\rangle \mapsto 0]\end{array}}{\langle \mathbb{CV}, \mathbb{SCV}, \mathbb{SEQ}, seq\rangle \Rightarrow^{st\langle x,\tau\rangle} \langle \mathbb{CV}', \mathbb{SCV}', \mathbb{SEQ}', seq\rangle}$$

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

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 2.2 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.*

2.2 Basic Ideas

Figure 2.3 presents an overview of the PSan system. PSan builds on the open-source Jaaru infrastructure [43] 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

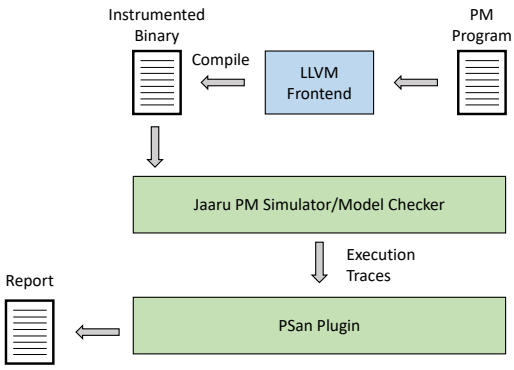


Figure 2.3: System Overview

generated by existing test data generation tools [77, 1, 11, 12, 8, 4, 71, 101, 100, 38, 39, 94]. 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.

2.2.1 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 2.1.4 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 2.4: An example of non-robust program with missing flush and drain operations. `x` and `y` are initialized to 0.

Figure 2.4 presents an example that we will use to present our basic approach. The left column in Figure 2.4 shows the code of the pre-crash execution and the right column shows the code of the post-crash execution. Section 2.4.1 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 2.4-a.

Consider an execution in which `r1 = 2` and `r2 = 5`. Figure 2.5 shows such an example and

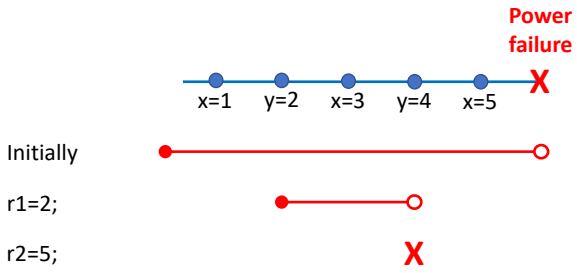


Figure 2.5: Constraints for execution of code in Figure 2.4 where `r1 = 2` and `r2 = 5`.

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 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`.

2.2.2 Supporting Threads

```
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 2.6: `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 2.7: 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` $= 1$?

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

Per-Thread Crash Intervals

Naïvely applying potential crash interval constraints to a multi-threaded execution trace using TSO order is overly restrictive. Figure 2.6 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.

Persistency Closure under Happens-Before

Another aspect of the simple approach in Section 2.2.1 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 2.7 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 2.7-(a) and 2.7-(b) present the pre-crash execution code for thread τ_1 and thread τ_2. Figure 2.7-(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 2.6 is robust, but `x = 1` is not persistent even though it is TSO ordered before `y = 1`, and `y = 1` is made persistent.

2.2.3 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:

$$st\langle x, \tau_1\rangle \xrightarrow{rf} ld\langle x\rangle \Rightarrow \mathbb{C}(\tau_1) := [\texttt{getcl}(st\langle x, \tau_1\rangle), \infty) \wedge \mathbb{C}(\tau_1). \quad (2.1)$$

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 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:

$$st\langle x, \tau_1\rangle \xrightarrow{rf} ld\langle x\rangle \wedge st\langle x, \tau_1\rangle \xrightarrow{tso} st\langle x, \tau_2\rangle \Rightarrow \mathbb{C}(\tau_2) := [0, \texttt{getcl}(st\langle x, \tau_2\rangle) \wedge \mathbb{C}(\tau_2). \quad (2.2)$$

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 $st\langle x, \tau_1\rangle$:

$$st\langle x, \tau_1\rangle \xrightarrow{rf} ld\langle x\rangle \wedge st\langle y, \tau_3\rangle \xrightarrow{hb} st\langle x, \tau_1\rangle \Rightarrow \mathbb{C}(\tau_3) := [\texttt{getcl}(st\langle y, \tau_3\rangle), \infty) \wedge \mathbb{C}(\tau_3). \quad (2.3)$$

2.2.4 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 $\mathcal{C}$ 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, $\mathcal{C}$ 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 2.3.1.

```
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 2.8: 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 2.8 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 $\mathcal{C}(e_1)$ as $\mathcal{C}(e_1) = [0, \texttt{getcl(x = 1)})$, because the first sub-execution of an equivalent strictly persistent execution must crash before `x = 1` commits to the cache, and $\mathcal{C}(e_2)$ has no constraints. Then when the load `s = y` reads from `y = 1`, $\mathcal{C}(e_1)$ becomes

$[0, \texttt{getcl(x = 1)}) \wedge [\texttt{getcl(y = 1)}, \infty)$, because the first sub-execution of the equivalent execution must crash after `y = 1` commits to the cache. Also, $\mathcal{C}(e_2)$ becomes $[0, \texttt{getcl(y = 2)})$, 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 $\mathcal{C}(e_1)$ are not satisfiable, and such equivalent execution does not exist.

Note that a misinterpretation of the constraint $\mathcal{C}(e_2) = [0, \texttt{getcl(y = 2)})$ would suggest that the second sub-execution should be empty. Then since `r = x` is not executed, $\mathcal{C}(e_1)$ becomes $[\texttt{getcl(y = 1)}, \infty)$, resutling in satisfiable constraints. However, this is not the case. First of all, the constraint $\mathcal{C}(e_2) = [0, \texttt{getcl(y = 2)})$ 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 $\mathcal{C}$, and so we do not remove the implications of the load `r = x` from $\mathcal{C}(e_1)$.

2.3 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 2.9: A simple concurrent programming language.

We present our algorithm for detecting robustness violations with respect to the simple concurrent language used by Px86_{sim} [97], as described in Figure 2.9. 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`, expresssions, 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.

2.3.1 Operational Semantics

Figure 2.10 presents our algorithm in operational semantics as an extension to the Px86_{sim} operational model.

We present a correctness proof for the algorithm in Section 2.7. Before performing the analysis in Figure 2.10, the algorithm in Figure 2.2 for computing clock vectors and sequence numbers is applied to the corresponding operations. After the analysis in Figure 2.10, we extend the transitions for the Px86_{sim} operational model [97].

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;
- $\texttt{top}(\mathit{Exec})$ returns the last sub-execution in Exec, *i.e.*, the current sub-execution;
- $\mathcal{C}$ maps a sub-execution e to its mapping $\mathbb{C}_e$ from threads to potential crash intervals.

States:

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

[LOAD-PREV]

$$\frac{\begin{array}{c} st\langle x, \tau\rangle \xrightarrow{rf} ld\langle x\rangle \qquad \hat{e} = \texttt{getexec}(st\langle x, \tau\rangle) \\ e_c = \texttt{top}(\mathit{Exec}) \qquad ld\langle x\rangle \in e_c \qquad \hat{e} \neq e_c \qquad \{st\langle x, \tau_1\rangle_1, ..., st\langle x, \tau_n\rangle_n\} = \texttt{next}(st\langle x, \tau\rangle, e_c) \\ \forall i \in \{1, ..., n\}.\hat{e}_i = \texttt{getexec}(st\langle x, \tau_i\rangle_i), \sigma_i = \mathbb{SCV}(st\langle x, \tau_i\rangle_i)(\tau_i) \\ \mathcal{C}_0 = \mathcal{C}[\hat{e} \mapsto \{\langle \tau', \mathcal{C}(\hat{e})(\tau') \wedge [\mathbb{SCV}(st\langle x, \tau\rangle)(\tau'), \infty)\rangle \mid \tau' \in \texttt{TId}\}] \\ \forall i \in \{1, ..., n\}.\mathcal{C}_i = \mathcal{C}_{i-1}[e_i \mapsto \mathcal{C}_{i-1}(\hat{e}_i)[\tau_i \mapsto \mathcal{C}_{i-1}(\hat{e}_i)(\tau_i) \wedge [0, \sigma_i))]] \end{array}}{\langle ld\langle x\rangle, \mathcal{C}\rangle \implies \langle \texttt{nextop}(ld\langle x\rangle), \mathcal{C}_n\rangle}$$

Figure 2.10: 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, $\mathcal{C}_0$ is the result of applying implications 2.1 and 2.3. Then the last line in Figure 2.10 iteratively applies the implication 2.2 for each store in the set $\texttt{next}(st\langle x, \tau\rangle, e_c)$.

2.3.2 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 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

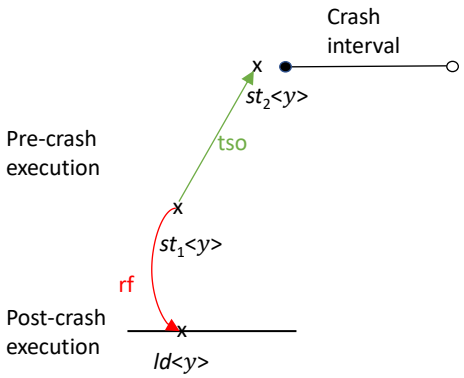


Figure 2.11: Reading from a store that is too old.

is too new. We first discuss the first case in more detail.

Reading from Too Old of Store. Figure 2.11 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 $\mathcal{C}_i$ 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 2.7 in which a thread crashes between performing a store and flushing and draining the store, but a second thread observes the

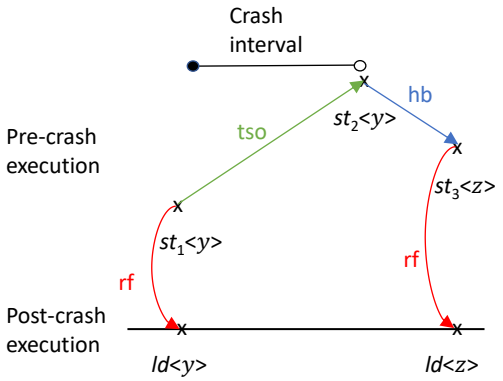


Figure 2.12: Reading from a store that is too new.

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 2.11. 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 2.12 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.

2.4 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 [43]. 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.

2.4.1 Methodology

We first tested PSan on the RECIPE [70] collection of PM indexes based on B+-trees, tries, radix trees, and hash tables [84, 49, 70]. CCEH [84] is an efficient hash table for persistent memory. FAST_FAIR [49] 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 [53], Memcached [23], and Redis [67]. 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.

2.4.2 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 2.1 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 2.1 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 2.1, in FAST_FAIR benchmark. In this benchmark, the `header` class is used at the beginning of the `page` class [49]. 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 2.1, PSan

Table 2.1: 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*

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 [70], 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.

2.4.3 Performance

We next ran 100 random executions with both PSan and Jaaru, the underlying model checker, to report the overhead of PSan. Table 2.2 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 2.2: 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

2.4.4 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* [24], *pointer tagging* [72], 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 2.1, 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 [43] and Witcher [33], 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.

2.5 Related 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 [9, 89, 68, 83] 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 2.3: Comparison with other tools; robustness subsumes ordering heuristics/conditions used in existing tools.

Tool	**Persistent Order**
PSan	Robustness
Witcher [33]	Dependence heuristic
PMDebugger [25]	User annotations
PMTest [79]	User annotations
XFDetector [78]	Commit store annotations
Jaaru [43]	Crash/assertion failure
Yat [69]	Crash/assertion failure
Agamotto [87]	Does not check order
Pmemcheck [61]	Does not check order
PMFuzz [77]	Just fuzzes input, uses Pmemcheck or XFDetector for checking
Hippocrates [86]	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, 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 2.3 summarizes the approaches other tools take to checking the order of PM stores. All these conditions are essentially instances of robustness violations. Witcher [33] 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 [79] and PMDebugger [25] 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 [78] requires that ordering constraints are specified implicitly by annotating a set of commit variables, otherwise it can report false positive. Jaaru [43] and Yat [69] only detect ordering bugs when the program crashes or asserts, and developers must manually localize the bug. Pmemcheck [61] and Agamotto [87] 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.

PSan can save significant manual effort compared to repairing bugs without a tool. While Hippocrates [86] 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 2.7). 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.*

2.6 Conclusion

This chapter 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.

2.7 Proof

For two sub-executions $e_i, e_j \subset Exec$, if e_i is earlier than e_j, then we write $e_i \prec e_j$; if e_i equals or is earlier than e_j, then we write $e_i \preceq e_j$.

Theorem 1. *Let P be a PM program. An execution $Exec$ of P under a weak persistency model is robust if and only if the algorithm in Figure 2.10 does not report a robust violation.*

Proof. Let $Exec$ be an execution with n crash events.

Forward We will prove the forward direction by contradiction. Suppose that the algorithm in Figure 2.10 reports a robustness violation and that $Exec$ is robust. Without loss of generality, we only need to consider the first robustness violation reported by the algorithm. Suppose that the first robustness violation is reported when a load $ld\langle x\rangle \in e_s$ reads from a store $st\langle x\rangle \in e_i$, where $e_i \prec e_s$, which imposes a constraint c that is incompatible with some existing constraint c_0 in $\mathcal{C}(e_j)(\tau_k)$. The constraint c is either of the form $[0, \alpha)$ or of the form $[\beta, \infty)$.

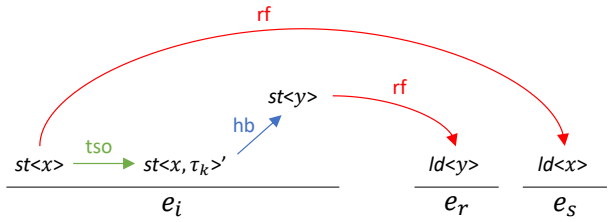


Figure 2.13: Illustration for forward direction *subcase 1a*.

Case 1: The constraint c is of the form $[0, \alpha)$. Then this constraint must be induced by some store $st\langle x, \tau_k\rangle' \in \texttt{next}(st\langle x\rangle, e_s)$ and α is the clock of $st\langle x, \tau_k\rangle'$, *i.e.*, $\alpha = \mathbb{SCV}(st\langle x, \tau_k\rangle')(\tau_k)$. Now we have two subcases: either $st\langle x, \tau_k\rangle' \in e_i$ or $st\langle x, \tau_k\rangle' \in e_j$, where $e_i \prec e_j$.

Subcase 1a: Suppose that $st\langle x, \tau_k\rangle' \in e_i$. Then we have $e_i = e_j$ and $st\langle x\rangle \xrightarrow{tso} st\langle x, \tau_k\rangle'$. The conflicting constraint c_0 must be of the form $[\beta, \infty)$ and $\beta > \alpha$. Therefore, it implies that

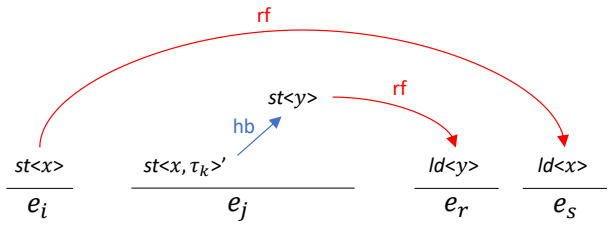


Figure 2.14: Illustration for forward direction *subcase 1b*.

the constraint c_0 was added to $\mathcal{C}(e_i)(\tau_k)$ when some store $st\langle y\rangle \in e_i$ was read from by a load $ld\langle y\rangle \in e_r$, where $e_i \prec e_r$, and $\beta = \mathbb{SCV}(st\langle y\rangle)(\tau_k)$. Since the constraint c_0 was added before c, the load $ld\langle y\rangle$ is either in the sub-execution e_s that contains $ld\langle x\rangle$ or in a sub-execution earlier than e_s, *i.e.*, $e_i \prec e_r \preceq e_s$. By the algorithm for computing clock vectors, there must be a store $st\langle z, \tau_k\rangle \in e_i$ that has the clock β. The store $st\langle z, \tau_k\rangle$ either happens before the store $st\langle y\rangle$ or is the store $st\langle y\rangle$. Now because α is the clock of $st\langle x, \tau_k\rangle'$, β is the clock of $st\langle z, \tau_k\rangle$, and $\beta > \alpha$, it implies that $st\langle x, \tau_k\rangle' \xrightarrow{hb} st\langle z, \tau_k\rangle$. Hence, we have $st\langle x, \tau_k\rangle' \xrightarrow{hb} st\langle y\rangle$.

Since both $st\langle x\rangle \in e_i$ and $st\langle y\rangle \in e_i$ are read from by loads in later sub-executions and $e_i \prec e_s$, we have $st\langle x\rangle, st\langle y\rangle \in F_s$ due to the condition 2 in Definition 2.2. Since F_s is closed under the happens-before relation, it implies that $st\langle x, \tau_k\rangle' \in F_s$ and $st\langle x\rangle \xrightarrow{tso} st\langle x, \tau_k\rangle'$ in F_s. In the case where all stores in F_s have committed to the cache before a crash occurs, $st\langle x, \tau_k\rangle'$ has committed to the cache. Therefore, the load $ld\langle x\rangle \in e_s$ that reads from $st\langle x\rangle$ in the execution *Exec* should not read from $st\langle x\rangle$ in F_s under strict persistency. Thus, we have derived a contradiction because the reads-from relation in F_s should be a subset of the relation in *Exec*.

Subcase 1b: Suppose that $st\langle x, \tau_k\rangle' \in e_j$, where $e_i \prec e_j$. This subcase is similar to *subcase 1a*. By the same reasoning, we can deduce that there exists a $st\langle y\rangle \in e_j$ such that $st\langle x, \tau_k\rangle' \xrightarrow{hb} st\langle y\rangle$ and that $st\langle y\rangle$ is read from by a load $ld\langle y\rangle \in e_r$, where $e_j \prec e_r \preceq e_s$. Hence, $st\langle x, \tau_k\rangle', st\langle y\rangle \in F_s$ because of the condition 2 in Definition 2.2 and that F_s is closed under the happens-before relation. In the case where all stores in F_s have committed to the cache before a crash occurs, $st\langle x, \tau_k\rangle'$ has committed to the cache. Now because $st\langle x\rangle \in e_i, st\langle x, \tau_k\rangle' \in e_j, ld\langle x\rangle \in e_s$ and

$e_i \prec e_j \preceq e_s$, $ld\langle x\rangle$ should not read from $st\langle x\rangle$ in F_s under strict persistency, leading to a contradiction.

Case 2: The constraint c is of the form $[\beta, \infty)$. Then $e_i = e_j$ and $\beta = \mathbb{SCV}(st\langle x\rangle)(\tau_k)$. The conflicting constraint c_0 must be of the form $[0, \alpha)$ and $\alpha < \beta$. Suppose that the conflicting constraint c_0 was added when a store $st\langle y\rangle$ was read from by a load $ld\langle y\rangle$ in a later sub-execution e_r. Then the constraint c_0 must be induced by some store $st\langle y, \tau_k\rangle' \in \texttt{next}(st\langle y\rangle, e_r)$, where $st\langle y, \tau_k\rangle' \in e_i$, $e_i \prec e_r$, and $\alpha = \mathbb{SCV}(st\langle y, \tau_k\rangle')(\tau_k)$. Since the constraint c_0 was added before c, we also have $e_r \preceq e_s$. We have two subcases: either $st\langle y\rangle \in e_i$ or $st\langle y\rangle \in e_l$, where $e_l \prec e_i$.

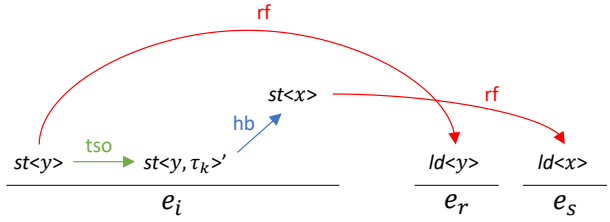


Figure 2.15: Illustration for forward direction *subcase 2a*.

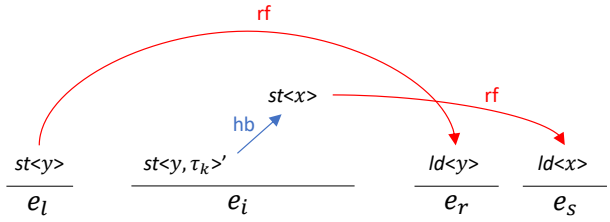


Figure 2.16: Illustration for forward direction *subcase 2b*.

Subcase 2a: Suppose that $st\langle y\rangle \in e_i$. Then we have $st\langle y\rangle \xrightarrow{tso} st\langle y, \tau_k\rangle'$. Since $\mathbb{SCV}(st\langle y, \tau_k\rangle')(\tau_k) = \alpha < \beta = \mathbb{SCV}(st\langle x\rangle)(\tau_k)$, by a similar argument as in *subcase 1a*, we can deduce that $st\langle y, \tau_k\rangle' \xrightarrow{hb} st\langle x\rangle$. Since $st\langle x\rangle \in e_i$ is read from by the load $ld\langle x\rangle \in e_s$, where $e_i \prec e_r \preceq e_s$, then $st\langle x\rangle \in F_r$. Similarly, $st\langle y\rangle \in F_r$. Since F_r is closed under the happens-before relation, we have $st\langle y, \tau_k\rangle' \in F_r$. In the case where all stores in F_r have committed to the cache before a crash occurs, $st\langle y, \tau_k\rangle'$ has committed to the cache. However, because of $st\langle y\rangle \xrightarrow{tso} st\langle y, \tau_k\rangle'$,

the load $ld\langle y\rangle \in e_r$ should not read from $st\langle y\rangle \in e_i$ in F_r under strict persistency, causing a contradiction.

Subcase 2b: Suppose that $st\langle y\rangle \in e_l$, where $e_l \prec e_i$. Similar to *subcase 2a*, we can deduce that $st\langle y, \tau_k\rangle' \xrightarrow{hb} st\langle x\rangle$. Since $st\langle x\rangle \in e_i$ is read from by the load $ld\langle x\rangle \in e_s$, where $e_i \prec e_r \preceq e_s$, we have $st\langle x\rangle \in F_r$. Similarly, $st\langle y\rangle \in F_r$. Since F_r is closed under the happens-before relation, we have $st\langle y, \tau_k\rangle' \in F_r$. In the case where all stores in F_r have committed to the cache before a crash occurs, $st\langle y, \tau_k\rangle'$ has committed to the cache. Because $st\langle y\rangle \in e_l$, $st\langle y, \tau_k\rangle' \in e_i$, and $e_l \prec e_i$, we can deduce that the load $ld\langle y\rangle \in e_r$ should not read from $st\langle y\rangle$ in F_r under strict persistency, which is a contradiction.

In conclusion, we have shown that if the algorithm in Figure 2.10 reports a robustness violation, then there does not exist a strictly persistent execution that is equivalent to *Exec*, and hence *Exec* is not robust.

Backward Suppose that the algorithm in Figure 2.10 does not report a robustness violation. Then the strategy is to construct a sequence of multi-threaded prefixes $F_2, ..., F_{n+1}$ as required by Definition 2.2. The proof is organized into the following 4 steps.

Step 1: Construction of multi-threaded prefixes.

We will construct F_s for some $1 < s \leq n+1$ as follows. If one such F_s can be constructed, then we can construct all others in a similar manner.

Since the map $\mathcal{C}$ is satisfiable, $\mathcal{C}(e_i)(\tau_j)$ is satisfiable for all $i = 1, ..., n+1$ and $j \in \texttt{TId}$. For each e_i, consider the constraints $\mathbb{C}_{i,j} := \mathcal{C}(e_i)(\tau_j)$. Since $\mathbb{C}_{i,j}$ is a joint and ($\wedge$) of a list of constraints of the form $[\alpha, \beta)$, we can treat each constraint as an interval and define $\overline{\mathbb{C}_{i,j}}$ as the set intersection of the list of constraints in $\mathbb{C}_{i,j}$. Since $\mathbb{C}_{i,j}$ is satisfiable, then $\overline{\mathbb{C}_{i,j}}$ is an non-empty interval of the form $[\alpha, \beta)$ and has a lower bound which is the clock of some store performed by thread τ_j in the sub-execution e_i. If $\mathbb{C}_{i,j}$ is an empty set, we will let

$\overline{\mathbb{C}_{i,j}} = [0, \infty)$. We will use $\mathit{begin}(\overline{\mathbb{C}_{i,j}})$ to denote the lower bound of $\overline{\mathbb{C}_{i,j}}$.

Then we can construct the multi-threaded prefix F_s in this way. For each e_i, where $1 \leq i < s$, we can get a multi-threaded prefix $e_i{}'$ by cutting off all operations that are sequenced after the store with the clock $\mathit{begin}(\overline{\mathbb{C}_{i,j}})$ in each thread $\tau_j \in \texttt{TId}$. We will take $e_s{}'$ to be e_s. Then we let F_s be $e_1{}'C_1...C_{s-1}e_s{}'$.

Step 2: Each $e_i{}'$ is a valid multi-threaded prefix of e_i.

By construction, $e_s{}'$ is a valid multi-threaded prefix for e_s. We also want to show that each $e_i{}'$, where $1 \leq i < s$, is a valid multi-threaded prefix for e_i. It is obvious that $e_i{}'$ is closed under the sequenced-before relation by construction. So we only need to show that $e_i{}'$ is closed under the happens-before relation over stores. If we can show that $e_i{}'$ is closed under the reads-from relation where both the source and destination of the reads-from edge are in $e_i{}'$, then the closure under the happens-before relation follows as a consequence.

To prove it by contradiction, suppose that there exists a pair of load $ld\langle x\rangle$ and store $st\langle x, \tau_j\rangle$ such that $st\langle x, \tau_j\rangle, ld\langle x\rangle \in e_i$, $st\langle x, \tau_j\rangle \xrightarrow{rf} ld\langle x\rangle$ in e_i, $st\langle x, \tau_j\rangle \notin e_i{}'$ but $ld\langle x\rangle \in e_i{}'$. Then $\mathbb{SCV}(st\langle x, \tau_j\rangle)(\tau_j) > \mathit{begin}(\overline{\mathbb{C}_{i,j}})$, and there exists some store $st\langle y, \tau_k\rangle \in e_i$ that is sequenced after $ld\langle x\rangle$ and that is read from by a load $ld\langle y\rangle$ in some sub-execution later than e_i. Then, we have $st\langle x, \tau_j\rangle \xrightarrow{hb} st\langle y, \tau_k\rangle$ in e_i, and $\mathbb{SCV}(st\langle y, \tau_k\rangle)(\tau_j) \geq \mathbb{SCV}(st\langle x, \tau_j\rangle)(\tau_j)$. When the store $st\langle y, \tau_k\rangle \in e_i$ is read from by the load $ld\langle y\rangle$ in a later sub-execution, the constraint $[\mathbb{SCV}(st\langle y, \tau_k\rangle)(\tau_j), \infty)$ should be added to $\mathbb{C}_{i,j}$, according to the algorithm in Figure 2.10. Then we have $\mathit{begin}(\overline{\mathbb{C}_{i,j}}) \geq \mathbb{SCV}(st\langle y, \tau_k\rangle)(\tau_j) \geq \mathbb{SCV}(st\langle x, \tau_j\rangle(\tau_j)) > \mathit{begin}(\overline{\mathbb{C}_{i,j}})$, which is a contradiction. Therefore, $e_i{}'$ is closed under the reads-from relation where both the source and destination of the reads-from edge are in $e_i{}'$. Then $e_i{}'$ is also closed under the happens-before relation and is a valid multi-threaded prefix of e_i for all $1 \leq i < s$.

Step 3: F_s is a valid multi-threaded prefix.

Next, in order to show that $F_s = e_1{'}C_1...C_{s-1}e_s{'}$ is a multi-threaded prefix of $e_1C_1...C_{s-1}e_s$, we need to show that F_s is closed under the reads-from relation. Since we have already shown that for all $1 \leq i < s$, $e_i{'}$ is closed under the reads-from relation where both the source and destination of the reads-from edge are in $e_i{'}$ (this also holds for $e_s{'}$ as $e_s{'} = e_s$), we only need to show that F_s is closed under the reads-from relation where the source and destination of the reads-from edge are in different sub-executions. In other words, suppose that $st\langle x, \tau_k\rangle \in e_i$, $ld\langle x\rangle \in e_j$, $st\langle x, \tau_k\rangle \xrightarrow{rf} ld\langle x\rangle$, and $ld\langle x\rangle \in e_j{'}$, where $e_i \prec e_j \preceq e_s$. Then we need to show that $st\langle x, \tau_k\rangle \in e_i{'}$. Since $st\langle x, \tau_k\rangle \in e_i$ is read from by the load $ld\langle x\rangle$ in the later sub-execution e_j, the constraint $[\mathbb{SCV}(st\langle x, \tau_k\rangle)(\tau_k), \infty)$ was added to $\mathbb{C}_{i,k}$. Therefore, we have $begin(\overline{\mathbb{C}_{i,k}}) \geq \mathbb{SCV}(st\langle x, \tau_k\rangle)(\tau_k)$, and it follows that $st\langle x, \tau_k\rangle \in e_i{'}$, based on the construction of $e_i{'}$. Thus, F_s is closed under the reads-from relation , and hence F_s is a valid multi-threaded prefix of $e_1C_1...C_{s-1}e_s$.

Step 4: F_s satisfies Definition 2.2.

We still need to show that F_s satisfies Definition 2.2. The condition 1 in Definition 2.2 is satisfied trivially by the construction of F_s. The same reasoning in the proof of *Step 4* can be applied to show that the condition 2 also holds. We will prove by contradiction that the condition 3 holds.

Suppose that for all $1 \leq i < s$, all stores in $e_i{'}$ have committed to the cache before the crash C_i occurs, but F_s is not an execution under strict persistency. Then there exist $st\langle x, \tau_k\rangle \in e_j{'}$, $st\langle x, \tau_l\rangle{'} \in e_j{'}$, and $ld\langle x\rangle \in e_r{'}$, where $e_j{'} \prec e_r{'} \preceq e_s{'}$, such that $st\langle x, \tau_k\rangle \xrightarrow{rf} ld\langle x\rangle$ and $st\langle x, \tau_k\rangle \xrightarrow{tso} st\langle x, \tau_l\rangle{'}$. Without loss of generality, assume that $st\langle x, \tau_k\rangle$ is immediately TSO ordered before $st\langle x, \tau_l\rangle{'}$. Then $st\langle x, \tau_l\rangle{'} \in \texttt{next}(st\langle x, \tau_k\rangle, e_r)$. Since $st\langle x, \tau_l\rangle{'} \in e_j{'}$, it implies that $\mathbb{SCV}(st\langle x, \tau_l\rangle{'})(\tau_l) \leq begin(\overline{\mathbb{C}_{j,l}})$. However, when $st\langle x, \tau_k\rangle$ was read from by the load $ld\langle x\rangle$ in e_r, the algorithm in Figure 2.10 would add the constraint $[0, \mathbb{SCV}(st\langle x, \tau_l\rangle{'})(\tau_l))$ to $\mathbb{C}_{j,l}$, and therefore $begin(\overline{\mathbb{C}_{j,l}}) < \mathbb{SCV}(st\langle x, \tau_l\rangle{'})(\tau_l)$. This contradicts with the inequality $\mathbb{SCV}(st\langle x, \tau_l\rangle{'})(\tau_l) \leq begin(\overline{\mathbb{C}_{j,l}})$. Therefore, condition 3 in Definition 2.2 also holds.

In conclusion, since each multi-threaded prefix F_s where $1 < s \leq n + 1$ can be constructed in the above fashion, the execution *Exec* is robust. □

Chapter 3

Static Approach: PMRobust

3.1 Introduction

In this chapter, we propose a novel static analysis that can ensure the absence of flush-related bugs and requires no test cases. Robustness has been proposed as a model for the correct use of flush and drain operations in Chapter 2 [42]. As a reminder, the observation here is that 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 [91] ensures that the "persistency memory order is identical to volatile memory order". Robustness ensures that any execution of a program under a weak persistency model is equivalent to some execution of the program under strict persistency. It thus suffices to ensure that the correct usage of flush and drain operations as additional flush and drain operations will not alter the set of possible post-crash program executions.

The PSan tool presented in Chapter 2 relied on a dynamic analysis combined with random execution generation or model checking to find robustness violations for persistent memory programs [42]. PSan suffers from the same limitations as all dynamic analysis—it requires

test cases and may miss bugs that are not revealed by the test cases. PMRobust finds all flush and fence bugs and does not require any test cases to find bugs. The cost of these stronger guarantees is that the static analysis has a higher risk of reporting false positives.

3.2 PMRobust

This section discusses key ideas behind our approach to verifying the safety of PM code in C/C++. We first discuss the requirements that robustness places on PM programs. We then present an approach to verifying that programs are robust to weak persistency models.

3.2.1 Ensuring data is correctly flushed

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

(a) Pre-crash execution

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

(b) Post-crash execution

Figure 3.1: Assume that `x` = `y` = `0` initially. If the post-crash execution observes `r2` = `1`, strict persistency requires that `r1` = `1`.

Figure 3.1 presents an example that illustrates the requirements of strict persistency. Strict persistency requires that the persistency order for stores respect the happens-before relation. This means that the store `x` = `1` must be persistency ordered before the store `y` = `1` and the execution in which `r1` = `0` and `r2` = `1` is forbidden.

It may initially appear that robustness would require a flush instruction after every store to persistent memory. However, it turns out that robustness only requires that the persistency order for stores respect the happens-before relation when post-crash executions can potentially observe a violation of strict persistency. For example if post-crash executions only read from `y`, then the program is robust even if `y` = `1` is made persistent and `x` = `1` is not. This observation is most relevant for a newly created persistent object that has not yet become reachable from the roots of persistent data structures. It suffices to wait to flush stores to a newly created persistent object until the object is inserted into a persistent data structure.

We next present a sufficient set of requirements on flush and drain operations to ensure robustness. Figure 3.2 presents a finite state machine that captures how to implement

robustness using flush and drain operations for the x86-TSO persistency model. We refer to a state in this finite state machine as an *escape persistency state*. The finite state machine captures the set of legal transitions for cache lines through escape persistency states. If there are two cache lines that contain objects that are escaped and non-clean, then this is a *robustness violation*. A key insight is differentiating between (1) memory locations that are *captured* by the local thread and thus stores to the memory location would not be visible if the program crashed and (2) memory locations that have *escaped* to become reachable from the roots of persistent data structures and thus stores would be visible if the program crashed.

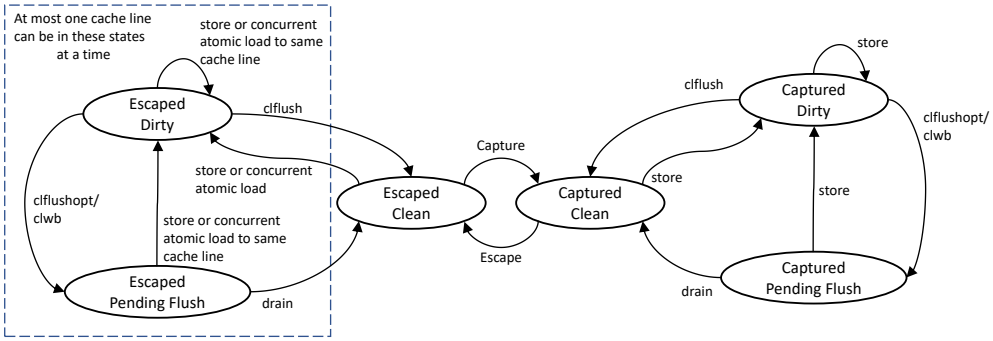


Figure 3.2: Using Flush & Drain Operations to Ensure Robustness

In general, persistent memory programs must ensure that data written to persistent memory before a crash was consistently written out before using the data. This check often takes the form of a commit store to a memory location that is read before the data to ensure the data's consistency. For example, the store at line 16 in Figure 3.4 is a commit store. Post-crash executions then read from this commit store to determine whether data is consistent. This commit store corresponds to the event that causes a memory location to escape. This insight has been used successfully by both the Jaaru [43] model checker and the XFDetector [78] persistency bug finding tool. In Jaaru's evaluation, violations of robustness typically corresponded to persistency bugs. The only exception we observed was the checksum pattern that uses a checksum to validate that data was read consistently.

Captured Objects: Memory locations can be captured (*i.e.*, there exists no path from the persistent data structure roots to the memory location). Stores to captured memory locations are not visible after a crash, and thus it is safe to delay flush instructions until immediately before the memory location escapes via insertion into a persistent data structure. This can have several benefits—first, it becomes possible to use optimized flush instructions like `clflushopt` or `clwb` on several cache lines and amortize the cost of the drain operation across multiple flush instructions. Second, if the program performs stores in a non-sequential order, it is possible to handle multiple non-consecutive store instructions to the same cache line with one flush instruction.

Escaped Objects: Memory locations have escaped if there is a path of references from a persistent data structure root to the memory location and the necessary conditions have been set so that data structure code may read from the memory location. Escaped memory locations require enforcing stronger persistency order constraints—the persistency order of stores to escaped memory locations must match the happens-before order. If consecutive stores happen to be to the same cache line, this happens automatically due to cache coherence. If they are to different cache lines, it is necessary to flush the first store before the second store.

Thread 1:

```
1   x = 1;
2   clflush(&x);
```

Thread 2:

```
3   r1 = x;
4   y = r1;
```

(a) Pre-crash execution

```
1   r2 = x;
2   r3 = y;
```

(b) Post-crash execution

Figure 3.3: Assume that `x` = `y` = `0` initially and all accesses are atomic. Can `r1` = `1`, `r2` = `0`, and `r3` = `1`?

Atomic loads require extra care to ensure robustness. Consider the example in Figure 3.3. If the pre-crash execution crashes before the `clflush` instruction in Thread 1 completes,

but after the store to `y` in Thread 2 has been made persistent, it is possible for `r2` = `0` and `r3` = `1`, violating strict consistency. Robustness in this case requires a `clflush` instruction to the cache line of `x` after Thread 2 reads from `x`. This example also has an implication for stores to escaped memory locations inside of critical sections—the store must be flushed before the lock is released.

3.2.2 Verifying Robustness

```
1  struct Node {
2    int data;
3    struct Node * next;
4  };
5
6  struct Stack {
7    struct Node * top;
8  };
9
10 void push(struct Stack *s, int val) {
11   struct Node * head = s->top;
12   struct Node * n = pmalloc(sizeof(struct Node));
13   n->data = val;
14   n->next = head;
15   clflush(n);
16   s->top = n;
17   clflush(s->top);
18 }
```

Figure 3.4: A Simple Persistent Stack

We begin with a simple example that we use to illustrate key concepts in our approach to verifying that persistent memory code correctly flushes data. Figure 3.4 presents a single-threaded persistent stack. The `push` method adds a new value on the top of the stack. The `push` procedure calls `pmalloc` to allocate a new stack node, stores the value `val` to the node, and updates the node's `next` field. Then it flushes the new node, update the top of the stack to reference the new node. Finally, the procedure flushes the update to the top of the stack.

In this example, the stack `s` and node `n` are the persistent variables and have one of the states in Figure 3.2. The stack `s` is the root of the persistent data structure and is escaped initially. When node `n` is created, it initially has the state $\langle \textit{captured}, \textit{clean} \rangle$ for both fields. The node `n` has the state $\langle \textit{captured}, \textit{dirty} \rangle$ for both fields after the stores at lines 13 and 14.

Both fields' states are changed to ⟨*captured*, *clean*⟩ after `n` is flushed. The commit store at line 16 makes `n` escaped as `n` is reachable from the persistent data structure root, and the state of `s` to ⟨*escaped*, *dirty*⟩. However, we are in an safe state, as only `s` is escaped and dirty. Finally, `s` is also flushed and becomes ⟨*escaped*, *clean*⟩ when the procedure ends.

Verification

We next discuss our proposed approach to verifying the safety of PM programs. Our approach builds on the finite state formulation for ensuring robustness from Section 3.2.1. The basic idea is to use a static analysis to compute at each program point a mapping from memory locations to set of potential escape persistency states.

The transfer function for an action A is then defined by applying the persistent state transitions from Figure 3.2 to the individual escape persistency states of the input set to generate the set of output escape persistency states. The transfer function is monotonic and so the analysis is a fixed-point dataflow analysis over a method's control flow graph.

The analysis must check several correctness properties. The first is to verify that the program does not take a forbidden transition that would violate robustness such as allowing a non-clean object to escape or have multiple dirty escaped cache lines. We compute the effect of method calls by extending escape persistency states with extra information that tracks their corresponding initial states when the method was first called. When the analysis of the method is complete, the static analysis has determined how the method changes each of the possible escape persistency states.

Loads pose an interesting challenge because they allow another thread to observe a store before it is made persistent, and that thread may then store a value that was derived from the value returned by the load. The later store can potentially be persisted before the initial store and thus a crash can leave the persistent memory in an inconsistent state. For example,

in Figure 3.3, it is possible for `y = r1` to be persisted before `x = 1`, and then the post-crash execution would read `r2 = 0` and `r3 = 1`.

This problem can be solved by inserting a flush instruction immediately after the load, but this incurs an overhead. We consider two cases for loads:

1. **Non-atomic Loads:** In the case that the load is a non-atomic load, there is no issue as long as the original store is flushed before its mutex is released and thus before it can be read. We require that non-atomic stores be flushed before any release operation such as an unlock is performed.

2. **Atomic Loads:** Atomic operations allow accessing memory that is not protected by a lock. However, atomic accesses can also be protected by a lock, which could be a common use case for PM code if the atomics are used to guarantee store atomicity in the case of a crash. However, if an atomic memory location is not protected by a mutex, we must add a flush instruction after any atomic loads. We address this issue by having atomic loads change the persistency state to dirty. This forces the thread to flush the data before performing other visible stores. Note that robustness violations from loads are not always bugs. There are design patterns that can cause false positives. These design patterns include *link-and-persist* [24], *pointer tagging* [72], and checksums. These design patterns allow post-crash executions to safely observe low-level violations of robustness without compromising high-level safety.

3.2.3 Relaxing Robustness for Checksums & Counters

A usage pattern that is sometimes used in persistent memory programs is to write some data and a checksum, and then persist both the data and the checksum. When accessing the data, the PM program first verifies the checksum before using the data. While this pattern is

safe, it can yield executions that are not equivalent to any execution under strict consistency and thus violates robustness. PMRobust includes annotations that developers can use to specify that an object is accessed using the checksum pattern. PMRobust then relaxes its robustness checks.

Another use case that could potentially occur in systems programming is that some memory locations may hold atomic values that can tolerate losing updates in the presence of a crash. A potential example is atomic counters that might only need to be flushed during shutdown or periodically. A crash might cause these counters to have stale values, but this may be acceptable in some cases.

3.3 Intraprocedural Analysis

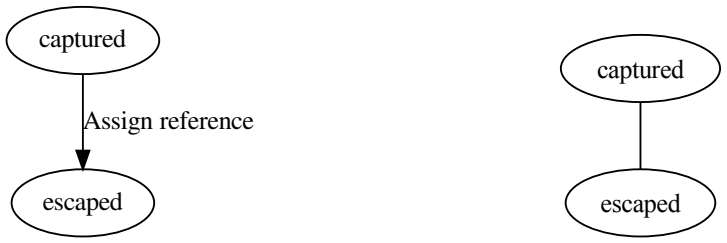
In this section, we first discuss the core intraprocedural analysis. Later, in Section 3.4 we will extend this analysis for the interprocedural context and to handle arrays. The intraprocedural part is implemented as a standard forward dataflow analysis. The algorithm maintains a program state at each instruction.

3.3.1 Preliminaries

The set of instructions that we analyze are non-atomic loads, non-atomic stores, atomic loads, atomic stores, atomic RMWs, assignments, and flush and fence instructions. An object can occupy multiple cache lines and thus an object reference $r \in R$ can be used depending on the field to access one of several different cache lines. Objects are by default not aligned to cache lines, and thus the static analysis will not necessarily know whether two different fields reside on the same cache line. Thus, we model a memory location $m \in M$ as the combination $l = \langle r, n \rangle \in \mathcal{L}$ of a reference $r \in R$ and a non-negative offset $n \in \mathbb{Z}^{0+}$ from that reference.

We next describe our core analysis approach. For each persistent memory location, our analysis must compute: (1) whether a reference to that memory location may have escaped to persistent memory, and (2) the status of any stores to that memory, *i.e.*, whether they have been flushed to persistent memory. We decompose these into two analysis problems. Although the original finite state machine in Figure 3.2 combines both properties into a single FSM, we have separated the two properties into two finite state machines to simplify the presentation.

Figure 3.5a presents a finite state machine that summarizes the dynamic semantics for whether a memory location has escaped and Figure 3.5b presents a lattice for our escape analysis. We use an escape state $e \in \mathcal{E}$ to represent one of the two escape values, *i.e.*, $\{\text{captured}, \text{escaped}\}$, from the escape analysis lattice. Here we need a may-escape analysis because we need to conservatively determine whether a reference may have escaped. Thus we have the escaped value lower on the lattice, and a merge of a escaped value with a captured value would yield the escaped state. The core analysis computes a map $\mathcal{G}_{\mathcal{E}} \subseteq R \times \mathcal{E}$ from memory locations to escape states at each program point. The meet operator $\sqcap : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ is defined by $e_1 \sqcap e_2 = \text{lower}(e_1, e_2)$, which returns the lower of the two lattice values. We write $e_1 \succeq e_2$ if e_1 is higher than or equal to e_2 in the lattice.

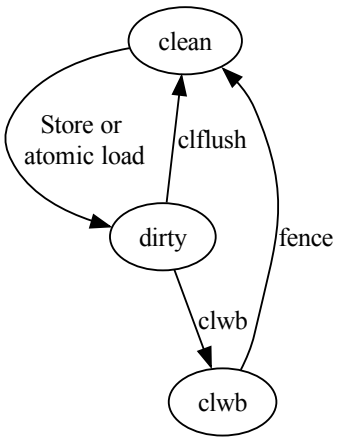


(a) Finite State Machine (b) Lattice

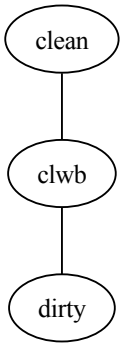
Figure 3.5: Lattice and FSM for Escape Analysis

Figure 3.6a presents a finite state machine that summarizes the dynamic semantics for persistency state of a memory location. and Figure 3.6b presents a lattice for our persistency state analysis. We use a persistency state $p \in \mathcal{P}$ to represent one of the three persistency state values, *i.e.*, $\{\text{clean}, \text{clwb}, \text{dirty}\}$. The lattice is ordered in this fashion, because we need to know whether a memory location may require a fence instruction (clwb) or whether it may require a fence and flush instruction (dirty). For example, if a reference is clean on one path to a node and clwb on a different path to the node, the analysis must conservative assume it is clwb at the merge point.

The core analysis computes a map $\mathcal{G}_{\mathcal{P}} \subseteq \mathcal{L} \times \mathcal{P}$ from memory locations to persistency states. The meet operator $\sqcap : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ and the ordering operator $\succeq$ for $\mathcal{P}$ are defined similar to the ones for $\mathcal{E}$.



(a) Finite State Machine



(b) Lattice

Figure 3.6: Lattice and FSM for Persistency State Analysis

`statement`	$\mathcal{G}'_{\mathcal{E}} = (\mathcal{G}_{\mathcal{E}} - \text{KILL}) \cup \text{GEN}$
`y=x`	$\begin{aligned} U &= \mathcal{G}_{\mathcal{A}}(\mathtt{x}) \cup \{\mathtt{y}\} \\ \mathcal{G}'_{\mathcal{A}} &= \{\langle r, S\rangle \mid \langle r, S\rangle \in \mathcal{G}_{\mathcal{A}} \wedge r \notin U\} \cup \\ &\quad \{\langle r, U\rangle \mid r \in U\} \\ \mathcal{G}_{\mathcal{E}}{}' &= \{\langle r, e\rangle \mid \langle r, e\rangle \in \mathcal{G}_{\mathcal{E}} \wedge r \neq \mathtt{y}\} \cup \\ &\quad \{\langle \mathtt{y}, e\rangle \mid \langle \mathtt{x}, e\rangle \in \mathcal{G}_{\mathcal{E}}\} \end{aligned}$
`*y=x` or `*y=&x->f`	$\begin{aligned} U &= \mathcal{G}_{\mathcal{A}}(\mathtt{x}) \\ \mathcal{G}_{\mathcal{E}}{}' &= \{\langle r, e\rangle \mid \langle r, e\rangle \in \mathcal{G}_{\mathcal{E}} \wedge r \notin U\} \cup \\ &\quad \{\langle r, \text{escaped}\rangle \mid \langle r, e\rangle \in \mathcal{G}_{\mathcal{E}} \wedge r \in U\} \end{aligned}$
`y=*x`	$\begin{aligned} U &= \mathcal{G}_{\mathcal{A}}(\mathtt{y}) \setminus \{\mathtt{y}\} \\ \mathcal{G}'_{\mathcal{A}} &= \{\langle r, S\rangle \mid \langle r, S\rangle \in \mathcal{G}_{\mathcal{A}} \wedge r \notin \mathcal{G}_{\mathcal{A}}(\mathtt{y})\} \cup \\ &\quad \{\langle \mathtt{y}, \{\mathtt{y}\}\rangle\} \cup \{\langle r, U\rangle \mid r \in U\} \\ \text{GEN} &= \{\langle \mathtt{y}, \text{escaped}\rangle\} \\ \text{KILL} &= \{\langle \mathtt{y}, *\rangle\} \end{aligned}$

Figure 3.7: Transfer Functions for Escape Analysis, where `x` and `y` point to PM locations

`statement`	$\mathcal{G}'_{\mathcal{P}} = (\mathcal{G}_{\mathcal{P}} - \text{KILL}) \cup \text{GEN}$
`x->f=v`	$\text{GEN} = \{\langle\langle \texttt{x}, \text{offset}(\texttt{f})\rangle, \text{dirty}\rangle\}$ $\text{KILL} = \{\langle\langle \texttt{x}, \text{offset}(\texttt{f})\rangle, *\rangle\}$
`y=x->f` where `f` is an atomic field	$\text{GEN} = \{\langle\langle \texttt{x}, \text{offset}(\texttt{f})\rangle, \text{dirty}\rangle\}$ $\text{KILL} = \{\langle\langle \texttt{x}, \text{offset}(\texttt{f})\rangle, *\rangle\}$
`flush(&x->f)`	$\text{GEN} = \{\langle\langle \texttt{x}, \text{offset}(\texttt{f})\rangle, \text{clean}\rangle\}$ $\text{KILL} = \{\langle\langle \texttt{x}, \text{offset}(\texttt{f})\rangle, *\rangle\}$
`clwb(&x->f)`	$\mathcal{G}'_{\mathcal{P}} = \{\langle l, p\rangle \mid \langle l, p\rangle \in \mathcal{G}_{\mathcal{P}} \wedge$ $(p \neq \text{dirty} \vee l \neq \langle \texttt{x}, \text{offset}(\texttt{f})\rangle)\} \cup$ $\{\langle l, \text{clwb}\rangle \mid \langle l, \text{dirty}\rangle \in \mathcal{G}_{\mathcal{P}} \wedge l = \langle \texttt{x}, \text{offset}(\texttt{f})\rangle\}$
`fence`	$\mathcal{G}'_{\mathcal{P}} = \{\langle l, p\rangle \mid \langle l, p\rangle \in \mathcal{G}_{\mathcal{P}} \wedge (p \neq \text{clwb})\} \cup$ $\{\langle l, \text{clean}\rangle \mid \langle l, \text{clwb}\rangle \in \mathcal{G}_{\mathcal{P}}\}$

Figure 3.8: Transfer Functions for Persistency State Analysis

3.3.2 Transfer Functions

Checking Whether Objects are Captured

In practice, it is difficult to precisely determine whether an object has escaped or not. We take a very simple approach to escape analysis — once a reference to a newly allocated struct or array is stored to anyplace other than a variable, we assume it has escaped. The key ideas of the analysis are that newly allocated object start in the captured state. For example, the statement `x=new` would result in the analysis computing that `x` is in the captured state at the program point immediately after this statement. This is stored in the map $\mathcal{G}_{\mathcal{E}}$. The analysis then computes the sets of variables that may reference the same object. After the statement `x=new`, the analysis would compute that `x` is the only variable to reference the memory it references. The analysis stores this information in the alias map $\mathcal{G}_{\mathcal{A}} \subseteq R \times \mathbb{P}(R)$ from references to the set of references that may alias, where $\mathbb{P}(R)$ denote the power set of R. If the value in a variable `x` is stored to some heap location, the variable `x` and all variables that may reference the same heap location are marked as escaped.

We next present the formalization of our escape analysis in Figure 3.7. Note that we use the form $\mathcal{G}'_{\mathcal{E}} = (\mathcal{G}_{\mathcal{E}} - \text{KILL}) \cup \text{GEN}$ if the GEN and KILL sets are constant sets, and we explicitly express the transfer functions as sets otherwise. We next discuss the transfer functions for key statements:

Assignment Instruction: When there is an assignment instruction `y=x`, `y` will now alias `x` and everything `x` aliased. We compute a set U that contains `y` and everything `x` aliased. Thus, for each element $r \in U$, we removed its old alias set S and replaced it with the new alias set U. At the same time, we update $\mathcal{G}_{\mathcal{E}}$ to assign `y` to have the same escape state as `x`. For example, line 11 in Figure 3.4 would create the alias `head` and `s`, and mark `head` as escaped.

Store Instruction: When we see a store instruction `*y = x` or `*y = &x->f` that stores the address `x` or the address of one of its fields `&x->f` to any location, we consider `x` and any address that `x` may alias as escaped. We compute a set U that contains all the references that `x` aliased. For each reference $r \in U$, we change its escape state to escaped and keep the escape state of other references in $\mathcal{G}_{\mathcal{E}}$ intact. For example, line 16 in Figure 3.4 makes `n` escaped.

Load Instruction: When a load instruction `y = *x` reads from `x`, and `y` points to some persistent memory, the analysis marks `y` as escaped. The reasoning is that if we had previously stored the address of some PM location `a` to `*x` (*i.e.*,`*x = a`), then `a` is considered as escaped due to the store instruction. Then when we load from `x`, the loaded value should also be marked as escaped. The * symbol in the KILL set $\{\langle \mathtt{y}, * \rangle\}$ denotes previous escape state that `y` has. Since `y` is overwritten, we compute the new alias map $\mathcal{G}'_{\mathcal{A}}$ as three pieces: 1) we compute U as the set that removes `y` from its alias set and update the alias set for each reference $r \in U$; 2) the alias set for `y` is the singleton $\{\mathtt{y}\}$; and 3) the alias sets for references $r \notin \mathcal{G}_{\mathcal{A}}(\mathtt{y})$ are the same. Note that while there may be multiple references to the same object as `y` references, precise reference information is not important to keep as all references will be marked as escaped.

Analyzing Persistency States

In this section, we describe how we model load, store, flush, and fence instructions in the persistency state analysis. Figure 3.8 presents the transfer functions for the persistency state analysis, where we assume `x` is a PM location. Similar to Section 3.3.2, we express transfer functions using GEN and KILL sets when they are constant sets. The reader may note that variables may alias but this analysis does not track aliasing information. The key observation is that aliasing does not create soundness issues — it simply means that the same variable that is used to perform a store must be used to flush the value. The lack of

aliasing information may result in false positives in cases where one alias is used to perform a store while another is used to flush the store. In Figure 3.8, we use $\langle$`x`, offset(`f`)$\rangle$) to denote the memory location of `x->f`.

Store/Atomic Store: When an atomic or non-atomic store writes to a field `x->f`, the KILL set removes the old persistency state of `x->f` (*i.e.*, $\langle$`x`, offset(`f`)$\rangle$), and the GEN set marks it as dirty. Similar to Figure 3.7, the * symbol in KILL sets denote previous states.

Atomic Load: As we mentioned in Section 3.2.2, an atomic load changes the state of the loaded variable or field to dirty. So the analysis removes the old persistency state of `x->f` and marks it as dirty.

Atomic RMW: Atomic RMWs are combinations of atomic loads and stores, so we apply the transfer functions of atomic load and store instructions.

Atomic CAS: An atomic CAS instruction is an atomic RMW instruction if it returns successfully, and is an atomic load otherwise. Since our transfer functions have the same effects when storing to a field `x->f` and when performing an atomic load from `x->f`, we consider atomic CAS instructions as atomic RMW instructions in our analysis.

Flush: When flushing the address of a field `x->f` with a `clflush` instruction, the KILL set removes the old persistency state of `x->f`, and the GEN set marks it as clean. Flushing the address of a field `x->f` with `clwb` or `clflushopt` instructions is subtle. In this case, the transfer function does not change the persistency states of locations other than `x->f`. The persistency state of location `x->f` is changed to `clwb` if its previous persistency state was dirty.

Fence: For fence instructions, the transfer function leaves the persistency states untouched for locations whose persistency states are not `clwb`. The locations whose persistency states are `clwb` are changed to the clean state.

3.3.3 Intraprocedural Error Reporting

PMRobust has two types of reports: errors and warnings. Warnings will be discussed in Sections 3.4.5 and 3.4.8. Error reporting falls into two categories: 1) unflushed variables at function exits; and 2) a store to an escaped PM location when a different PM location is already escaped and non-clean.

The first category of errors is checked at function exits. When we complete the analysis of a function, we get the program states at each function exit and take a union of the states by using meet operators. If the state of any PM location that is not a function parameter or the return value is escaped and non-clean, we report an error for the location.

The second category of errors is checked at each instruction. If two PM locations are escaped and non-clean at any point of time, this is an error. However, to avoid creating duplicate reports, we only report this type of error when a second PM location becomes escaped and non-clean, given that some PM location is already escaped and non-clean in the program state. To address robustness violations involving multiple threads, if there is a release operation to a non-persistent memory location (atomic store release or unlock) and there is an escaped and non-clean location, PMRobust reports an error and the source line number of the second store. Recall from Section 3.3.1, a PM location l is a pair $\langle r, n \rangle$ of a reference and an offset from the reference, so an escaped PM object with two or more fields being non-clean is also reported as an error.

3.4 Interprocedural Analysis

In this section, we first discuss extending the core analysis to be interprocedural. Then we discuss how we detect bugs that involve objects reachable from function parameters and bugs that involve multiple functions. Lastly, we present details about array support, how we

detect references to persistent memory, and theorems regarding PMRobust's soundness.

3.4.1 Context Sensitivity

To handle function calls, we extend our analysis with function call contexts. The context sensitivity is implemented using *function summary tables*. Each function maintains a function summary table that maps calling contexts to summarized cached results.

For a function with n parameters, a *calling context* is of the form $C \in (\mathcal{E} \times \mathcal{P})^n$. A calling context C can be thought of as a vector that stores the abstract escape and persistency state for each function parameter. While a function parameter may have m fields and each field can have a persistency state, we introduce an abstraction to collapse the m persistency states to a single abstract persistency state, by returning the lowest persistency state among the m persistency states. The purpose of the abstraction is to reduce the possible number of calling contexts and cached results. The details about the abstraction is discussed in Section 3.4.2.

For a function F with n parameters, A *cached result* that corresponds to a calling context C is of the form $R_{\langle F,C\rangle} \in (\mathcal{E} \times \mathcal{P})^{n+1}$. We use the notation $R_{\langle F,C\rangle}$ because the cached result depends on the calling context C and function F. Similar to a calling context, a cached result can also be thought of as a vector that stores the abstract escape and persistency state for each function parameter and the return value. In the case where function F does not have a return value, the last element in the cached result is ignored in the analysis.

For two calling context $C = \langle\langle e_1, p_1\rangle, ..., \langle e_n, p_n\rangle\rangle$ and $C' = \langle\langle e'_1, p'_1\rangle, ..., \langle e'_n, p'_n\rangle\rangle$ of some function F, we say that C is higher than C' if $e_i \succeq e'_i$ and $p_n \succeq p'_n$ for all i. The meet operator $\sqcap : (\mathcal{E} \times \mathcal{P})^{n+1} \to (\mathcal{E} \times \mathcal{P})^{n+1}$ for cached results is defined as the pairwise meet operation. While cached results can contain extra information about F, such as the *marksObjDirEsc* bit discussed in Section 3.4.4, we omit it in the representation here. To extend our alias analysis

to be interprocedural, we also store the aliasing information between function parameters and the return value to cached results.

The interprocedural analysis uses a worklist that stores pairs $\langle F, C \rangle$ of a function F and a calling context C that need analysis. The worklist is initialized to include all functions with the calling contexts of all parameters having the state $\langle \textit{captured}, \textit{clean} \rangle$. When we complete the analysis of a function with a calling context, we update the function summary table. Every time a function F's cache results are updated, we push all callers of F with their calling contexts to the worklist. There are three cases when processing a function call F with a calling context C:

- **Case 1:** We have already analyzed F with the calling context C, *i.e.*, the cached result $R_{\langle F,C \rangle}$ corresponding to $\langle F, C \rangle$ is present. Then the cached result is used to approximate the state of the parameters and the return value of F right after the call site. The approximation of states will be discussed in Section 3.4.2.

- **Case 2:** We have not analyzed F with the calling contexts C before but have analyzed F with the calling contexts higher that C. Then we take the cached results $R_{\langle F,C' \rangle}$ for all calling contexts C' higher than C, and use the merged result of this set of cached results via meet operator for approximation. Finally, we push the pair $\langle F, C \rangle$ to the worklist. This choice ensures that we maintain monotonicity when processing call sites (and thus preserve the termination guarantees for dataflow analysis) — if the analysis lowers the incoming analysis states for the parameters, the returned analysis state will either not change or be lower.

- **Case 3:** We have not analyzed F with the calling contexts C or any calling context higher that C before. We approximate all parameters and the return value of F as having the state $\langle \textit{captured}, \textit{clean} \rangle$. Then, we push the pair $\langle F, C \rangle$ to the worklist.

3.4.2 Approximating Calling Context Persistency States

In this section, we discuss how calling contexts are computed from program states, and how program states are approximated based on cached results. At function call sites, we compute calling contexts from program states. If cached results are present, then we use the abstract states in the cached results to approximate the program state of function parameters and the return value right after the call.

When analyzing a function with a calling context, we approximate the initial program state of the parameters according to the calling context. When the analysis completes for a function, we first get the program states at each function exit, and then take a union to compute a final program state. Finally, we compute the abstract escape and persistency state for each parameter and the return value and store the abstract states in the function's cached result for the given calling context.

At each program point, a variable `x` that has n fields has a program state $ps \in \mathcal{E} \times \mathcal{P}^n$, where each field has a persistency state in Figure 3.6b. Thus, to reduce the number of possible states in calling contexts and cached results, we introduce an abstraction $Abs : \mathcal{E} \times \mathcal{P}^n \to \mathcal{E} \times \mathcal{P}$ define by $Abs(\langle e, p_1, ..., p_n \rangle) = \langle e, \text{lowest}(p_1, ..., p_n) \rangle$ that maps a variable's program state to an abstract escape and persistency state. In other words, applying Abs on the program state of `x` preserves the escape state but only keeps the lowest persistency state in fields of `x`. Meanwhile, when approximating a variable's program state with an abstract state, we use $AbsRev : \mathcal{E} \times \mathcal{P} \to \mathcal{E} \times \mathcal{P}^n$ define by $AbsRev(\langle e, p \rangle) = \langle e, (p, ..., p) \rangle$ that approximates the persistency state of each field as the abstract persistency state.

3.4.3 Objects Reachable from Parameters

When a function F has a parameter `p` whose field `f` points to some PM object `q`, the calling contexts or function cached results of F does not have information about the persistency state of `q`. Thus, if F accesses `p->f`, dereferences it, and modifies the content of `q`, the changes to `q` are not reflected in the cached results of F.

To deal with this issue, we note that when `p->f` is dereferenced, a load instruction is performed, *i.e.*, there is some instruction `y = *p->f`. So instead of recording the changes to `*p->f` in the persistency state of `p`, we treat `y` as a new label with initially clean states. By the transfer functions in Figure 3.7, `y` is considered as escaped due to the load instruction. If the function F writes to `y`, we require the content of `y` be flushed before F returns. Otherwise, we report an error.

```
1  void F(int &a) {
2    a = 1;
3    flush(&a);
4  }
5
6  void main() {
7    x = 1;
8    F(y);
9  }
```

(a) A buggy execution

```
1  void F(int &a) {
2    a = 1;
3    flush(&a);
4  }
5
6  void main() {
7    x = 1;
8    F(x);
9  }
```

(b) A bug-free execution

Figure 3.9: Assume that `x` and `y` reside on different cache lines and are escaped and clean initially.

3.4.4 Stores in Function Calls

A naive implementation of context sensitivity can sometimes miss bugs caused by stores in methods. Figure 3.9a presents such a buggy execution. Assume both `x` and `y` are initially escaped and clean. Although `f` flushes `a` right after the store, the `main` function still has a bug, as the execution has two cache lines that contain escaped and non-clean objects right after line 2. Therefore, to detect such bugs, we add an *marksObjDirEsc* bit to function

cache results to indicate if a function with a clean calling context makes an escaped object non-clean. We also set this bit if a function with a clean calling context performs a release operation to catch robustness violations involving multiple threads.

Figure 3.9b shows an execution similar to the one in Figure 3.9a but is bug-free. The key observation is that the parameter of `f` in Figure 3.9b is escaped and non-clean, while the parameter of `f` in Figure 3.9a is not. We generalize the condition to functions with multiple parameters.

Theorem 2. *Suppose function F calls $G(x_1, ..., x_n)$ and some objects are escaped and non-clean in the program state of F right before calling G. If the marksObjDirEsc bit is set in G under the calling context, while none of $x_1, ..., x_n$ is escaped and non-clean, then this is a robustness violation.*

Proof. If the *marksObjDirEsc* bit is set in G under the calling context, and none of G's parameters is escaped and dirty, then G must make some object O escaped and dirty, and O is different from the objects that are already escaped and dirty before calling G in F. Therefore, calling G causes a robustness violation. □

Theorem 3. *Suppose function F calls $G(x_1, ..., x_n)$ and no objects are escaped and non-clean in the program state of F right before calling G. Then if calling G causes any robustness violation, the violation will be detected while analyzing G with the calling context.*

Proof. Since no objects are escaped and dirty in the program state of F before calling G, the calling context of G has all of its elements being $\langle \text{captured}, \text{clean} \rangle$. So if calling G causes any robustness violation in F, the violation will be detected while analyzing G with the calling context. □

Theorem 4. *Suppose function F calls $G(x_1, ..., x_n)$ and some objects are escaped and non-clean in the program state of F right before calling G. Suppose some of $x_1, ..., x_n$ is escaped*

and non-clean. Then if there is a robustness violation caused by calling G from F, some robustness violation is reported.

Proof. Suppose that there is one escaped and dirty object O in the program state of F before calling G. Then O must be one of G's parameters. Therefore, the only case that causes a robustness violation is where G makes some other object escaped and dirty before flushing O. This violation can be detected when analyzing g with its calling context.

Now suppose that there are more than one escaped and dirty objects in the program state of F before calling G. Without loss of generality, assume there are exactly two such objects O_1 and O_2. Suppose O_1 is passed into G while O_2 is not. If G makes any object other that O_1 escaped and dirty, then this is the same case as the previous paragraph. If G stores to O_1, there is a robustness violation between the pair O_1 and O_2. However, a robustness violation already exists between the pair before calling G. □

3.4.5 Handling Arrays

We next discuss how PMRobust treats arrays. Our key idea is to abstract array writes as a pair of an array reference and an index. Formally, we model an array element $l = \langle r, n \rangle \in \mathcal{L}_a$ as a reference $r \in R$ and a non-negative index $n \in \mathbb{Z}^{0+}$ from that reference. We abstract the array index using the variable that provided the value used by the array dereference operation. Thus, our abstraction is only able to track dirty array elements as long as the original index variable exists. To ensure soundness, when a function writes to some PM array, we conservatively require the written element to be flushed before the function returns.

We conservatively assume all array elements have escaped and only compute a map $\mathcal{G}_{\mathcal{P}_a} \in \mathcal{L}_a \times \mathcal{P}$ from array elements to persistency states. Figure 3.10 presents the transfer functions for the array persistency analysis. Figure 3.10 is similar to Figure 3.8 except that the addresses

`statement`	$\mathcal{G}'_{\mathcal{P}_a} = (\mathcal{G}_{\mathcal{P}_a} - \text{KILL}) \cup \text{GEN}$
`a[i]=v`	$\text{GEN} = \{\langle\langle \texttt{a}, \texttt{i}\rangle, \text{dirty}\rangle\}$ $\text{KILL} = \{\langle\langle \texttt{a}, \texttt{i}\rangle, *\rangle\}$
`flush(&a[i])`	$\text{GEN} = \{\langle\langle \texttt{a}, \texttt{i}\rangle, \text{clean}\rangle\}$ $\text{KILL} = \{\langle\langle \texttt{a}, \texttt{i}\rangle, *\rangle\}$
`y=a[i]` where `a[i]` is atomic	$\text{GEN} = \{\langle\langle \texttt{a}, \texttt{i}\rangle, \text{dirty}\rangle\}$ $\text{KILL} = \{\langle\langle \texttt{a}, \texttt{i}\rangle, *\rangle\}$
`clwb(&a[i])`	$\mathcal{G}'_{\mathcal{P}_a} = \{\langle l, p\rangle \mid \langle l, p\rangle \in \mathcal{G}_{\mathcal{P}_a} \wedge (p \neq \text{dirty} \vee l \neq \langle \texttt{a}, \texttt{i}\rangle\})\cup$ $\{\langle l, \text{clwb}\rangle \mid \langle l, \text{dirty}\rangle \in \mathcal{G}_{\mathcal{P}_a} \wedge l = \langle \texttt{a}, \texttt{i}\rangle\}$
`fence`	$\mathcal{G}'_{\mathcal{P}_a} = \{\langle l, p\rangle \mid \langle l, p\rangle \in \mathcal{G}_{\mathcal{P}_a} \wedge (p \neq \text{clwb})\}\cup$ $\{\langle l, \text{clean}\rangle \mid \langle l, \text{clwb}\rangle \in \mathcal{G}_{\mathcal{P}_a}\}$

Figure 3.10: Transfer Functions for Array Persistency State Analysis

being stored to and loaded from are replaced by array elements. At a function exit, we report warnings if the map $\mathcal{G}_{\mathcal{P}_a}$ contains any element whose persistency state is not clean.

3.4.6 Detecting References to Persistent Memory

PMRobust uses a CFL-reachability-based alias analysis introduced by Zheng and Rugina[115] to distinguish persistent memory locations from non-persistent locations. First, the analysis requires a set of user-configured persistent memory allocators, and the pointers returned by these allocators are identified as the initial set of persistent memory pointers. The aliases of known persistent memory (PM) pointers are iteratively computed and added to the set until a fixed point is reached, ensuring all potential PM pointers are eventually included. The PM locations in a program are then locations pointed to by the computed PM pointers. Alternatively, this procedure could be viewed as a taint analysis, with the taint representing the capability to point to persistent memory, which is propagated by the transition rules of the CFL. The CFL-reachability-based formulation of the aliasing relation is precise and enables a demand-driven algorithm, in which only pointers that point to persistent memory or can reach persistent memory (through a series of dereferences and offsets) are explored. This helps our analysis avoid calculating aliasing relations between a large number of non-persistent memory pointers, such as those pointing to the stack.

Our version of the alias analysis is adapted from an existing implementation from the LLVM-8 codebase[30]. It uses function summary to speed up the analysis on large programs and eagerly calculates alias relations between all pointers. We modified the analysis to only explore aliases of PM pointers following the original demand-driven formulation[115]. The implementation is context-sensitive due to the use of function summaries. We added type-based field-sensitivity to the analysis, where a PM pointer status is tracked for each offset of a struct type. Whenever an offset of a concrete struct is identified as a PM pointer, the status

is set to true for the offset of the struct type and propagates to the offset of all structs of that type. This approach does not sacrifice much precision compared to full field-sensitivity, as persistent memory programs that aim to be correct mostly likely use specialized data types for persistent memory. Using the same data type for both volatile and persistent memory is indeed rarely seen in the benchmarks we evaluate in Section 3.5.

3.4.7 Interprocedural Error Reporting

The error reporting mechanism for interprocedural analysis is the same as the intraprocedural one, except that robustness violations that involve multiple functions are also reported. In this section, we present a lemma and a theorem about PMRobust's soundness.

Lemma 5. *If a PM location* `x` *is reachable from persistent data structure roots, then the escape analysis will mark* `x` *and its alias as escaped.*

Proof. We will prove the statement for intraprocedural analysis first, and then prove for the interprocedural analysis.

Case 1: If `x` is stored to some data structure via `*y = x`, then `x` and all elements in its alias set $\mathcal{G}_{\mathcal{A}}(\texttt{x})$ are marked as escaped by the second transfer function in Figure 3.7. Furthermore, if `x` is later loaded from `y`, say `a = *y`, then `a` is an alias of `x`. Note that although `a` is not in the alias set $\mathcal{G}_{\mathcal{A}}(\texttt{x})$ of `x`, `a` is also marked as escaped by the third transfer function in Figure 3.7.

Case 2: If `x` is added to the alias set $\mathcal{G}_{\mathcal{A}}(\texttt{a})$ of some variable `a` via `a = x`, and later `a` is made escaped via `*y = a`, then the second transfer function in Figure 3.7 marks the entire set $\mathcal{G}_{\mathcal{A}}\texttt{a}$ as escaped.

Case 3: If `x` has already escaped, and later an alias of `x` is created via `y = x`, then the first

transfer function in Figure 3.7 marks `y` as escaped.

For interprocedural analysis, we only need to consider two cases.

Case 1: Suppose `x` has already escaped in some function `H`, then `y` becomes an alias of `x` in some function `G(x, y, ...)`, where `H` is the caller of `G`. Note that the function cached results also contain the may-alias information between function parameters and the return value. So the analysis will use the cached result to mark `y` as escaped.

Case 2: Suppose `y` becomes an alias of `x` in some function `F(x, y, ...)`, and then `x` escapes in some function `G(x, ...)` (or `y` escapes in `G(y, ...)`), where `F` and `G` share the same caller `H`. Right after calling `F` in `H`, `H` has the information that `y` may alias `x`. If `x` escapes in `G(x, ...)`, then when the analysis uses the cached result of `G` to mark `x` as escaped, it also marks the variables that may alias `x` as escaped. The case is similar when we have `y` escape in `G(y, ...)` instead of `x` escaping in `G(x, ...)`. □

```
1 void F(...) {
2   x = 1;
3   y = 1;
4 }
```

(a) Subcase A.1

```
1 void F(){
2   x = 1;
3   G(y, ...);
4 }
```

(b) Subcase A.2

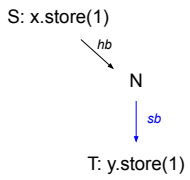
```
1 void F() {
2   G(x, ...);
3   y = 1;
4 }
```

(c) Subcase A.3

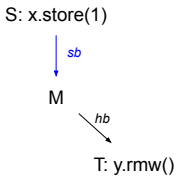
```
1 void F() {
2   G1(x, ...);
3   G1(y, ...);
4 }
```

(d) Subcase A.4

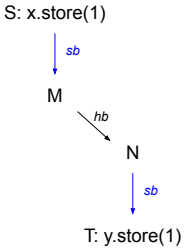
Figure 3.11: Assume that `x` and `y` are PM locations that reside on different cache lines and are escaped and clean initially.



(a) Subcase B.1



(b) Subcase B.2



(c) Subcase B.3

Figure 3.12: Assume that `x` and `y` are PM locations that reside on different cache lines and are escaped and clean initially, where *sb* represents the sequenced-before relation, and *hb* represents the happens-before relation.

Theorem 6. *If a program has a robustness violation, then* PMRobust *will report some error or warning.*

Proof. A robustness violation must involve two stores S and T that write to PM locations `x` and `y` on different cache lines such that `x` and `y` are reachable from persistent data structure roots, that S happens before T, and that `x` is not flushed before the store to `y`. By Lemma 5, we can assume that both `x` and `y` have been marked as escaped. We next enumerate the cases and show that our analysis reports some error or warning. For simplicity, we assume `x` and `y` are two different PM objects. The same proof will apply if they are fields of the same PM object that reside on different cache lines.

Case A. We first consider the case where the stores S and T are in the same thread. Figure 3.11 presents a few subcases. Without loss of generality, we assume that functions `G`, $\texttt{G}_1$, $\texttt{G}_2$ write to their first parameters without flushing them. Although there can be chains of function calls in practice, those cases are not different from the cases presented in Figure 3.11. *Subcase A.1* is obvious, as the analysis detects two escaped and dirty objects at `y = 1` and reports an error. For *Subcase A.2*, the analysis also reports some robustness violation according to Theorem 4. *Subcase A.3* is similar to *Subcase A.1*, as we assume `G` writes to its first parameter. *Subcase A.4* is similar to *Subcase A.2*.

Case B. Now we consider the case where the stores S and T are in two different thread. Since we assume S happens before T, there are three possible subcases as presented in Figure 3.12. The $\overset{hb}{\rightarrow}$ edges represent the happens-before relation, and the $\overset{sb}{\rightarrow}$ edges represent the sequenced-before relation or the program order.

In *Subcase B.1*, since N is sequenced before T, we can assume that operations N and T are in the same function, say F. Since N happens after S, S is an atomic store, and N is either

an atomic load or atomic RMW that reads from `x`. In either case, by the transfer functions in Figure 3.8, `x` is dirty right after the operation N. Thus, there are two escaped and dirty objects after the store T, and the analysis reports a bug while analyzing the function F.

In *Subcase B.2*, since M happens before T that performs a store to `y`, T is an atomic RMW operation, and M can be an atomic store or atomic RMW that stores to `y`. Since S is sequenced before M, we can assume that S and M are in the same function, say F. By the same reasoning as *Subcase B.1*, there are two escaped and dirty objects after the operation M, and the analysis reports a bug while analyzing the function F.

In *Subcase B.3*, the operations M and N establish a happens-before relation. Although, the operations S and T are not necessarily atomics, that does not change the proof. There are three possibilities. Subcase B.3.1) If M and N are atomic operations that read from or write to persistent memory locations, then it is similar to *Subcase B.1* and *Subcase B.2*. Subcase B.3.2) If M and N are atomic operations on non-persistent memory locations, then M must be a release operation and we require that (a) all cache lines must be clean before doing any release operation and (b) any function call in which all parameters are clean in the calling context that contains a release operation marks itself as having done a release operation, and thus all cache lines must be clean before making such a function call. Subcase B.3.3) If M and N are locking operations where M is an release and N is an acquire, it is the same as Subcase B.3.2.

Other Cases. We will discuss some other cases in this paragraph. Note that the above proof also applies if `x` and `y` are array elements. If `x` is some object reachable from some function parameter, then Section 3.4.3 mentions the strategy where we create a new label when `x` is dereferenced. The new label is marked as escaped by the transfer functions in Figure 3.7. Thus, the proof above also applies. Lastly, if `x` is some address computed by pointer arithmetic, then we will throw a warning right after the store to `x`. □

3.4.8 Limitations

Our treatment of pointer arithmetic is very imprecise and report a warning when there are stores to or loads from PM addresses computed by pointer arithmetic. We also do not implement support for global pointers as they are rarely used to reference persistent memory.

3.5 Evaluation

In this section, we evaluate the effectiveness and performance of our analysis at uncovering persistency robustness violations on a suite of benchmarks. We start by describing the benchmarks and our system configuration. We then discuss our evaluation methodology and present the analysis report from each of the benchmarks. Lastly, we discuss our findings from the reports and address sources of imprecision in our analysis.

System Setup: PMRobust was implemented as an analysis pass in LLVM 8.0.0. Our experiments were carried out on a Ubuntu 20.04.6 machine with a 10 core 3.7 GHz Intel i9-10900K processor and 128GB RAM.

3.5.1 Methodology

We tested PMRobust on the RECIPE[70] collection of benchmarks consisting of B+-trees, tries, radix trees, and hash table variants that were specialized to function as PM indexes. Of the five data structures in the RECIPE benchmarks, one (P-HOT) failed to compile with LLVM, and we used the remaining four (P-ART, P-BwTree, P-CLHT, P-Masstree) in the evaluation. In addition, we included CCEH[84], an efficient PM hashtable; and FAST_FAIR[49], an efficient implementation of B-Tree for PM, in our evaluation.

We also evaluated our tool on PMDK[53]. PMDK uses checksums and thus can safely eliminate many flush operations and *thus is not really amenable to analysis by* PMRobust. We included PMDK as it is commonly used in prior work, but as expected it results in a large number of false positives. We omitted both Memcached as it also makes use of checksums and thus would show similar results, and Redis because it is redundant as it uses the same PMDK library.

Our analysis is designed to start from a main function, and incrementally analyze all functions reachable from it, and therefore an entry point containing a main function is needed for each data structures we test on. For this purpose, we used test programs provided by the data structure implementer to ensure all appropriate user-facing API calls (and all functions reachable from them) are covered.

3.5.2 Bug Detection

Table 3.1: New Robustness Violation Bugs

#	Benchmark	Location	Cause of Robustness Violation
14	P-ART	N16::key	atomic load of key in N16::getChildren
15	P-ART	N16::children	atomic load of key in N16::getChildren
16	P-ART	N16::children	atomic load of key in N16::getChild
17	P-ART	N256::children	atomic load of key in N16::getChildren
18	P-ART	N4::children	atomic load of key in N4::getChild
19	P-ART	N4::children	atomic load of key in N4::getChildren
20	P-ART	N48::children	atomic load of key in N48::getChild
21	P-ART	N48::children	atomic load of key in N48::getChildren
22	P-ART	N48::childIndex	atomic load of childIndex in N48::deleteChildren
52	PMDK	stats_persistent:: heap_curr_allocated	atomic load of heap_curr_allocated in STATS_CTL_HANDLER
53	PMDK	stats_persistent:: heap_curr_allocated	increment heap_curr_allocated in palloc_heap_action_on_process
54	PMDK	stats_persistent:: heap_curr_allocated	subtract from heap_curr_allocated in palloc_heap_action_on_process
56	PMDK	dst	applying ULOG_ENTRY_AND on dst in ulog_entry_apply
57	PMDK	dst	applying ULOG_ENTRY_SET on dst in ulog_entry_apply
58	PMDK	dst	applying ULOG_ENTRY_BUF_COPY on dst in ulog_entry_apply

In our evaluation, PMRobust found a total of *80* bugs, of which *60* are not memory management related and can be found in Section 3.8, and *20* are memory management bugs, and will be addressed separately. Compared to PSan [42], a dynamic analysis that checks for

the kind of persistency violation, we found 6 new bugs in PMDK and 9 new bugs in P-ART of the RECIPE benchmark, listed in Table 3.1. It is important to note that we found new bugs in benchmark suites have been heavily analyzed by many bug finding tools.

As far as we are aware these bugs have not been reported by other studies in the literature. The new bugs #14-22 are unflushed atomic loads, which are only violations if a thread loads data stored by another thread, a scenario that does not occur in P-ART's test program and thus not explored by PSan. Similarly, #56-58 are located on different branches of the same switch statement as the previously reported bug #55. They have not been found before most likely because they are not called by the test programs. These new reports provide evidence for the advantages of the static approach. Since PMRobust does not currently support function pointers, there is one bug reported by PSan that our tool failed to report.

All of the violations are due to missing flushes after the dirty-state-inducing operation to the memory location listed, which is either a write or an atomic load. The violations caused by missing flushes after locking and unlocking operations (*e.g.*, #11, #12, #13) would in fact cause deadlocks after recovering from a crash. To address this problem, the RECIPE paper assumes that "the locks used in the index are non-persistent, and that the locks are re-initialized after a crash". However, these assumptions are not implemented in their codebase, and PMRobust is right to identify them as violations with respect to the implementation.

Note that the number of reports do not necessarily correspond to the number of bugs. A function with missing flushes could cause multiple violations when they are used in multiple places in the program, causing multiple bug reports. For example, this is the case for `N::writeLockOrRestart`, reported as #11. This function is used for acquiring the write lock on tree nodes in P-ART, which caused twelve bug reports from separate function calls. For the sake of clarity, we only list such functions once in the table. A template function with missing flushes could also be reported multiple times if they are instantiated multiple times in the LLVM IR.

Memory Management Bugs PMRobust found *20* bugs in the memory management part of the benchmark code responsible for memory allocation and garbage collection. As mentioned in the paper, the RECIPE repository is implemented to demonstrate the performance of the proposed technique for constructing PM indexes, and does not fully implement crash recovery in all parts of the code, in particular in its memory management code. As a result, ensuring crash consistency in the RECIPE memory management code requires more fundamental changes than adding missing flushes found by PMRobust.

Comparison with Agamotto Agamotto [87] is a tool that finds multiple types of persistency bugs. The violations we check for correspond to their missing flush correctness bugs, of which Agamotto report 2 in PMDK and 1 in RECIPE, whereas PMRobust reports 9 in PMDK and 51 in RECIPE.

3.5.3 False Positives

Table 3.2: Report False Positive Rate

Benchmark	# of Reports	# of False Positives	% of False Positives
P-ART	291	176	60%
P-BwTree	113	49	43%
P-CLHT	7	5	71%
P-Masstree	57	32	56%
CCEH	51	31	61%
FAST_FAIR	160	111	69%
PMDK	174	162	93%

We report the false positive rate of our tool on each benchmark in Table 3.2. The false positive reports come from the following sources: (1) not accounting for RECIPE's state inconsistency repair mechanism, which tolerates certain violations; (2) not accounting for certain conditional flushes (*e.g.*, flushes under if statements) that are guaranteed to execute

at runtime; (3) not accounting for transient runtime state stored in persistent memory; (4) not accounting for cache alignment where different fields of an PM object or different PM objects can be on the same cache line; (5) over-approximation of escaped states; and (6) over-approximation of violations due to atomic loads.

Of the above sources, RECIPE's inconsistency repair mechanism and conditional flushes are inherently dynamic, and to handle them statically would require extensive manual annotations. Storing unflushed transient states into persistent memory is a practice seen in performance-critical libraries like PMDK, which accounts for most of the false positives there. This is used for example to store a volatile mutex together the persistent memory pool it protects. These runtime states are either protected by a checksum, or are always reinitialized during recovery. Extensive annotations would be needed to rule out the transient states in persistent memory, which indicates a fundamental mismatch between these libraries and our approach.

While modeling of escaped states could be made more precise by depending on more sophisticated static techniques — *e.g.*, a flow-sensitive points-to analysis that determines whether each memory location is reachable from a persistent root at each program point, we chose not to investigate them in the scope of this paper as they add extra complexity and are not central to design of our analysis. Similarly, more precise modeling of atomic loads might involve techniques such as information-flow tracking, as atomically loaded values before a crash could influence the post-crash state through indirect information flow, which we leave as future work.

Overall, the false positive rates of our analysis are due to the inherent imprecision of static analyses, and the approximations we made for practical purposes. The high false positive rates of static checking in general also manifest in many practical type systems, such as the linear type system of Rust that conservatively reject many safe programs. Despite these false positives, the advantages of the static approach are (1) that it is able to report all potential violations on all paths rather than only on paths explored at runtime and (2) there is no

need to construct extensive test suites to reveal bugs. We believe that these tradeoffs are likely to be worthwhile for persistent data structures that store critical data.

3.5.4 Performance

We ran PMRobust 10 times on each of the benchmarks and present the average analysis times in Table 3.3. The analysis is reasonably fast for all of the benchmarks, and terminates in less than 20 seconds for PMDK, the largest of them. The code size of each benchmark is included for reference.

Table 3.3: Average analysis time of PMRobust over 10 runs

#	Benchmark	Time(s)	Code Size(KLOC)
1	P-ART	7.49	2.7
2	P-BwTree	10.19	10.9
3	P-CLHT	0.60	17.6
4	P-Masstree	9.02	2.5
5	CCEH	1.10	4.4
6	FAST_FAIR	3.58	2.2
7	PMDK	16.41	51.5*

* only includes data structures and sublibraries used in the evaluation

3.6 Related Work

Persistent Memory Bug Detection. There is work on checking/testing PM programs to find bugs. In particular, XFDetector [78] uses a finite state machine to track the consistency and persistency of persistent data. PMTest [79] lets developers annotate a program with checking rules to infer the persistency status of writes and ordering constraints between writes. Pmemcheck [61] checks how many stores were not made persistent and detects memory overwrites using binary rewriting. Yat [69] is an attempt to model check persistent memory. It injects failures before fence operations and eagerly enumerates all post-failure states to detect potential bugs. Agamotto [87] finds bugs in persistent memory programs by using symbolic execution. It uses a priority-based static analysis to steer program execution to program states that frequently modify PM. Jaaru [43] takes a constraint-based approach to enumerating executions that can drastically reduce the number of post-failure executions. PMDebugger [25] is a debugger developed on top of Valgrind that tracks operations to find persistency bugs. Hippocrates [86] is a tool for automatically fixing persistency bugs by analyzing crash information. Although these tools are able to find many bugs, none of these tools can assure the absence of flush/fence bugs. POG [95] and Pierogi [5] provide logics that can be used to manually reason about program behaviors.

Programming Models for PM. There is work on building systems that allow developers to use PM in a reliable way without knowing the details of PM. For example, a line of work [13, 35, 37, 75] proposes to use (software or hardware) transactions to provide (failure and thread) atomicity. Another line of work [7, 15, 47, 55, 76] advocates use of locks or synchronization-free regions [40]. Memento [20] provides detectable checkpointing—it extends standard checkpoint with support to allow the system to be able to detect the status of in flight operations when the crash occurred. These approaches typically incur large overheads to support the necessary logging.

Constructive Approaches. In addition to these general-purpose debugging tools and programming models, there is a rich literature on systematic transforms for lock-free data structures to use persistent memory [102, 6, 56, 22, 32, 24]. Most of these constructive approaches leverage different techniques to deduce flush and fence instructions for lock-free programs. More recently, Mirror [32] 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. [56] 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.

3.7 Conclusion

This paper presents an analysis that can statically check for violations of robustness, a sufficient condition for the correct usage of flush and fence operations in persistent memory programs. PMRobust is the first static analysis tool for checking for bugs with the use of flush and fence operations and is the only tool that can verify the absence of flush and fence bugs on all program executions. PMRobust found 80 bugs in popular PM benchmarks.

3.8 Bug Listing

All non-memory bugs found by PMRobust in the evaluation are listed in Table 3.4.

Table 3.4: Robustness Violation Bugs

#	Benchmark	Location	Cause of Robustness Violation
1	P-ART	Tree::loadKey	store to loadKey in constructor of Tree
2	P-ART	Tree::epoche	store to epoche in constructor of Tree
3	P-ART	node.children[i]	atomic load of node.children[i] in Tree::lookup
4	P-ART	node.children[i]	atomic load of n.children[i] in Tree::lookupRange
5	P-ART	n.children	atomic load of n.children in Tree::lookupRange
6	P-ART	N4::key	store to key in N4::insert
7	P-ART	N4::children	store to children in N4::insert
8	P-ART	N4::compactCount	store to compactCount in N4::insert
9	P-ART	N16::compactCount	store to compactCount in N16::insert
10	P-ART	N48::compactCount	store to compactCount in N48::insert
11	P-ART	N:: typeVersionLockObsolete	locking in N::writeLockOrRestart
12	P-ART	N:: typeVersionLockObsolete	locking in N::lockVersionOrRestart
13	P-ART	N:: typeVersionLockObsolete	unlocking in N::writeUnlock
14*	P-ART	N16::key	atomic load of key in N16::getChildren
15*	P-ART	N16::children	atomic load of key in N16::getChildren
16*	P-ART	N16::children	atomic load of key in N16::getChild
17*	P-ART	N256::children	atomic load of key in N16::getChildren
18*	P-ART	N4::children	atomic load of key in N4::getChild
19*	P-ART	N4::children	atomic load of key in N4::getChildren
20*	P-ART	N48::children	atomic load of key in N48::getChild
21*	P-ART	N48::children	atomic load of key in N48::getChildren
22*	P-ART	N48::childIndex	atomic load of childIndex in N48::deleteChildren
23	P-BwTree	AllocationMeta::next	store to AllocationMeta::next in AllocationMeta::GrowChunk
24	P-BwTree	AllocationMeta::tail	store to AllocationMeta::tail in AllocationMeta::TryAllocate
25	P-BwTree	BwTree::this	store to various fields of Bwtree in its constructor
26	P-CLHT	bucket	store to bucket in clht_put
27	P-Masstree	masstreeptr	store to masstreeptr in initOrRecoverPersistentData
28	P-Masstree	new_sibling	store to new_sibling by calling its constructor in leafnode::inter_insert
29	P-Masstree	p.version_	store to p.version_ by calling p-¿inter_insert in leafnode::split
30	P-Masstree	leafnode::wlock	store to wlock in constructor of leafnode
31	P-Masstree	leafnode::wlock	store to wlock in leafnode::lock
32	P-Masstree	leafnode::wlock	store to wlock in leafnode::unlock
33	P-Masstree	leafnode::wlock	store to wlock in leafnode::try_lock
34	CCEH	Directory::_	store to _ in constructor of Directory
35	CCEH	Directory::_[i]	store to _[i] in constructor of Directory
36	CCEH	Directory::lock	locking in Directory::Acquire
37	CCEH	Directory::lock	unlocking in Directory::Release
38	CCEH	Directory::sema	locking in Directory::Insert
39	CCEH	Directory::sema	unlocking in Directory::Insert
40	CCEH	Directory::key	writing to key in in Directory::Insert
41	FAST_FAIR	btree::this	store to various fields in constructor of btree
42	FAST_FAIR	header::this	store to various fields in constructor of header
43	FAST_FAIR	page::entry::ptr	store to entry::ptr in constructor of entry
44	FAST_FAIR	page::switch_counter	store to switch_counter in btree::insert_key(uint64_t, char*, int*, bool, bool)
45	FAST_FAIR	page::entry::ptr	store to entry::ptr in btree::insert_key(uint64_t, char*, int*, bool, bool)
46	FAST_FAIR	page::last_index	store to last_index in btree::insert_key(uint64_t, char*, int*, bool, bool)
47	FAST_FAIR	page::switch_counter	store to switch_counter in btree::insert_key(key_item*, char*, int*, bool, bool)
48	FAST_FAIR	page::entry::ptr	store to entry::ptr in btree::insert_key(key_item*, char*, int*, bool, bool)
49	FAST_FAIR	page::last_index	store to last_index in btree::insert_key(key_item*, char*, int*, bool, bool)
50	FAST_FAIR	page::switch_counter	store to switch_counter in btree::remove_key
51	FAST_FAIR	page::last_index	store to last_index in btree::remove_key
52*	PMDK	stats_persistent:: heap_curr_allocated	atomic load of heap_curr_allocated in STATS_CTL_HANDLER
53*	PMDK	stats_persistent:: heap_curr_allocated	increment heap_curr_allocated in palloc_heap_action_on_process
54*	PMDK	stats_persistent:: heap_curr_allocated	subtract from heap_curr_allocated in palloc_heap_action_on_process
55	PMDK	dst	applying ULOG_ENTRY_OR on dst in ulog_entry_apply
56*	PMDK	dst	applying ULOG_ENTRY_AND on dst in ulog_entry_apply
57*	PMDK	dst	applying ULOG_ENTRY_SET on dst in ulog_entry_apply
58*	PMDK	dst	applying ULOG_ENTRY_BUF_COPY on dst in ulog_entry_apply
59	PMDK	dst	memcpy to dst in ulog_store
60	PMDK	dst	memcpy to dst in ulog_entry_apply

* New bugs found by PMRobust

Bibliography

[1] D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina. Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. *J. Autom. Reasoning*, 45:397–414, 12 2010.

[2] ARM. Arm architecture reference manual armv8, for a-profile architecture. `https://developer.arm.com/documentation/ddi0487/latest`, September 2021.

[3] J. Arulraj and A. Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1753–1758, New York, NY, USA, 2017. Association for Computing Machinery.

[4] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 12–22, New York, NY, USA, 2011. Association for Computing Machinery.

[5] E. V. Bila, B. Dongol, O. Lahav, A. Raad, and J. Wickerson. View-based owicki–gries reasoning for persistent x86-tso. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*, page 234–261, Berlin, Heidelberg, 2022. Springer-Verlag.

[6] H.-J. Boehm and D. R. Chakrabarti. Persistence programming models for non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 55–67, New York, NY, USA, 2016. Association for Computing Machinery.

[7] H.-J. Boehm and D. R. Chakrabarti. Persistence programming models for non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 55–67, New York, NY, USA, 2016. Association for Computing Machinery.

[8] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of c programs: Experience with pathcrawler. In *2009 ICSE Workshop on Automation of Software Test*, pages 70–78, Vancouver, BC, Canada, 2009. Institute of Electrical and Electronics Engineers.

[9] A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In L. Aceto, M. Henzinger, and J. Sgall, editors, *Automata, Languages and Programming*, pages 428–440, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[10] B. Bridge. Nvm-direct library. `https://github.com/oracle/nvm-direct`, September 2021.

[11] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, USA, 2008. USENIX Association.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), Dec. 2008.

[13] D. Castro, P. Romano, and J. Barreto. Hardware transactional memory meets memory persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, pages 368–377, Vancouver, BC, Canada, 2018. Institute of Electrical and Electronics Engineers.

[14] H. Cha, M. Nam, K. Jin, J. Seo, and B. Nam. B3-tree: Byte-addressable binary b-tree for persistent memory. *ACM Trans. Storage*, 16(3), July 2020.

[15] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. Association for Computing Machinery.

[16] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, Feb. 2015.

[17] X. Chen, E. H.-M. Sha, A. Abdullah, Q. Zhuge, L. Wu, C. Yang, and W. Jiang. Udorn: A design framework of persistent in-memory key-value database for nvm. In *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, Hsinchu, Taiwan, 2017. Institute of Electrical and Electronics Engineers.

[18] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Y. Chen, J. Shu, J. Ou, and Y. Lu. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage*, 14(1), Apr. 2018.

[20] K. Cho, S. Jeon, A. Raad, and J. Kang. Memento: A framework for detectable recoverability in persistent memory. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[21] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS XVI*, pages 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[22] M. Dananjaya, V. Gavrielatos, A. Joshi, and V. Nagarajan. *Lazy Release Persistency*, pages 1173–1186. Association for Computing Machinery, New York, NY, USA, 2020.

[23] I. Danga Interactive. Memcached. `https://github.com/lenovo/memcached-pmem`, November 2018.

[24] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 373–385, USA, 2018. USENIX Association.

[25] B. Di, J. Liu, H. Chen, and D. Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 503–516, New York, NY, USA, 2021. Association for Computing Machinery.

[26] N. Douglas. P1026R0: A call for a data persistence (iostream v2) study group. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1026r0.pdf`, 2018.

[27] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[28] P. Ekemark, Y. Yao, A. Ros, K. Sagonas, and S. Kaxiras. Tsoper: Efficient coherence-based strict persistency. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 125–138, Seoul, Korea, 2021. Institute of Electrical and Electronics Engineers.

[29] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. Association for Computing Machinery.

[30] L. Foundation. Llvm-8. `https://github.com/llvm/llvm-project/tree/release/8.x`, August 2019.

[31] A. Freij, S. Yuan, H. Zhou, and Y. Solihin. Persist level parallelism: Streamlining integrity tree updates for secure persistent memory. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 14–27, Athens, Greece, 2020. Institute of Electrical and Electronics Engineers.

[32] M. Friedman, E. Petrank, and P. Ramalhete. *Mirror: Making Lock-Free Data Structures Persistent*, pages 1218–1232. Association for Computing Machinery, New York, NY, USA, 2021.

[33] X. Fu, W.-H. Kim, A. P. Shreepathi, M. Ismail, S. Wadkar, D. Lee, and C. Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, SOSP 2021, pages 100–115, New York, NY, USA, 2021. Association for Computing Machinery.

[34] N. Gao, Z. Liu, and D. Grunwald. Dtranx: A seda-based distributed and transactional key value store with persistent memory log, 2017.

[35] K. Genç, M. D. Bond, and G. H. Xu. Crafty: Efficient, HTM-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 59–74, New York, NY, USA, 2020. Association for Computing Machinery.

[36] J. S. George, M. Verma, R. Venkatasubramanian, and P. Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 859–872, Boston, MA, USA, July 2020. USENIX Association.

[37] E. Giles, K. Doshi, and P. Varman. Continuous checkpointing of HTM transactions in NVM. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 70–81, New York, NY, USA, 2017. Association for Computing Machinery.

[38] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[39] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing, November 2008.

[40] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 46–61, New York, NY, USA, 2018. Association for Computing Machinery.

[41] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. Relaxed persist ordering using strand persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665, Valencia, Spain, 2020. Institute of Electrical and Electronics Engineers.

[42] H. Gorjiara, W. Luo, A. Lee, G. H. Xu, and B. Demsky. Checking robustness to weak persistency models. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 490–505, New York, NY, USA, 2022. Association for Computing Machinery.

[43] H. Gorjiara, G. H. Xu, and B. Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 415–428, New York, NY, USA, 2021. Association for Computing Machinery.

[44] H. Gorjiara, G. H. Xu, and B. Demsky. Yashme: Detecting persistency races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 830–845, New York, NY, USA, 2022. Association for Computing Machinery.

[45] M. Ha and S.-H. Kim. Ink: In-kernel key-value storage with persistent memory. *Electronics*, 9(11), 2020.

[46] M. Hoseinzadeh and S. Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 429–442, New York, NY, USA, 2021. Association for Computing Machinery.

[47] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482, New York, NY, USA, 2017. Association for Computing Machinery.

[48] H. Huang, K. Huang, L. You, and L. Huang. Forca: Fast and atomic remote direct access to persistent memory. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 246–249, Orlando, FL, USA, 2018. Institute of Electrical and Electronics Engineers.

[49] D. Hwang, W.-H. Kim, Y. Won, and B. Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 187–200, USA, 2018. USENIX Association.

[50] Intel. 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`, June 2020.

[51] Intel. Memory optimized for data-centeric workloads. `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`, 2021.

[52] Intel. Revolutionizing memory and storage. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`, 2021.

[53] Intel Corporation. Persistent memory development kit. https://pmem.io/pmdk/, 2020.

[54] Intel Corporation. Intel inspector. https://software.intel.com/content/www/us/en/develop/tools/onea components/inspector.html, 2021.

[55] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. Association for Computing Machinery.

[56] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In C. Gavoille and D. Ilcinkas, editors, *Distributed Computing*, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[57] J. Jeong and C. Jung. Pmem-spec: Persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 517–529, New York, NY, USA, 2021. Association for Computing Machinery.

[58] J. Jeong, C. H. Park, J. Huh, and S. Maeng. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 520–532, Fukuoka, Japan, 2018. Institute of Electrical and Electronics Engineers.

[59] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[60] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, Feb. 2019. USENIX Association.

[61] T. Kapela. An introduction to pmemcheck (part 1) - basics. `https://pmem.io/2015/07/17/pmemcheck-basic.html`, July 2015.

[62] A. Khyzha and O. Lahav. Taming x86-tso persistency. *Proc. ACM Program. Lang.*, 5(POPL), Jan. 2021.

[63] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 481–493, New York, NY, USA, 2017. Association for Computing Machinery.

[64] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411, New York, NY, USA, 2016. Association for Computing Machinery.

[65] H. Kumar, Y. Patel, R. Kesavan, and S. Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, Feb. 2017. USENIX Association.

[66] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[67] R. Labs. Redis. `https://github.com/pmem/redis`, August 2020.

[68] O. Lahav and R. Margalit. Robustness against release/acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 126–141, New York, NY, USA, 2019. Association for Computing Machinery.

[69] P. Lantz, S. Dulloor, S. Kumar, R. Sankaran, and J. Jackson. Yat: A validation framework for persistent memory software. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association.

[70] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. RECIPE: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[71] G. Li, I. Ghosh, and S. P. Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 609–615, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[72] N. Li and W. Golab. Brief announcement: Detectable sequential specifications for recoverable shared objects. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 557–560, New York, NY, USA, 2021. Association for Computing Machinery.

[73] S. Li and L. Huang. Lospem: A novel log-structured framework for persistent memory. *J. Emerg. Technol. Comput. Syst.*, 16(3), May 2020.

[74] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. Association for Computing Machinery.

[75] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. Association for Computing Machinery.

[76] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *Proceedings of the 51st Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO-51, pages 258–270, Virtual Event , Greece, 2018. Institute of Electrical and Electronics Engineers.

[77] S. Liu, S. Mahar, B. Ray, and S. Khan. Pmfuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 487–502, New York, NY, USA, 2021. Association for Computing Machinery.

[78] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.

[79] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[80] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[81] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'17, page 4, USA, 2017. USENIX Association.

[82] L. Marmol, M. Chowdhury, and R. Rangaswami. Libpm: Simplifying application usage of persistent memory. *ACM Trans. Storage*, 14(4), Dec. 2018.

[83] Y. Meshman, N. Rinetzky, and E. Yahav. Pattern-based synthesis of synchronization for the c++ memory model. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 120–127, Austin, Texas, 2015. FMCAD Inc.

[84] M. Nam, H. Cha, Y.-R. Choi, S. H. Noh, and B. Nam. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST '19, pages 31–44, USA, 2019. USENIX Association.

[85] D. Narayanan and O. Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. Association for Computing Machinery.

[86] I. Neal, A. Quinn, and B. Kasikci. Hippocrates: Healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021, pages 401–414, New York, NY, USA, 2021. Association for Computing Machinery.

[87] I. Neal, B. Reeves, B. Stoler, and A. Quinn. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064, Banff, Alberta, November 2020. USENIX Association.

[88] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. Association for Computing Machinery.

[89] P. Ou and B. Demsky. Automo: Automatic inference of memory order parameters for c/c++11. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 221–240, New York, NY, USA, 2015. Association for Computing Machinery.

[90] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[91] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276, Minneapolis, MN, USA, 2014. Institute of Electrical and Electronics Engineers.

[92] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro*, 35(3):125–131, 2015.

[93] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the nvram era. *Proc. VLDB Endow.*, 7(2):121–132, Oct. 2013.

[94] C. S. Pundefinedsundefinedreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. Association for Computing Machinery.

[95] A. Raad, O. Lahav, and V. Vafeiadis. Persistent owicki-gries reasoning: a program logic for reasoning about persistent programs on intel-x86. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[96] A. Raad, L. Maranget, and V. Vafeiadis. Extending intel-x86 consistency and persistency: formalising the semantics of intel-x86 memory types and non-temporal stores. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[97] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis. Persistency semantics of the Intel-x86 architecture. *Proceedings of the ACM on Programming Languages*, 4(POPL), December 2019.

[98] A. Raad, J. Wickerson, and V. Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), Oct. 2019.

[99] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685, Waikiki, HI, USA, 2015. Institute of Electrical and Electronics Engineers.

[100] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. Association for Computing Machinery.

[101] N. Tillmann and J. de Halleux. Pex–white box test generation for .net. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, pages 134–153, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[102] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[103] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[104] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[105] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[106] X. Wu and A. L. N. Reddy. Scmfs: A file system for storage class memory. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Seattle, WA, USA, 2011. Institute of Electrical and Electronics Engineers.

[107] F. Xia, D. Jiang, J. Xiong, and N. Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 349–362, USA, 2017. USENIX Association.

[108] J. Xu and S. Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, USA, 2016. USENIX Association.

[109] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. Association for Computing Machinery.

[110] Y. Xu, J. Izraelevitz, and S. Swanson. *Clobber-NVM: Log Less, Re-Execute More*, pages 346–359. Association for Computing Machinery, New York, NY, USA, 2021.

[111] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, USA, 2015. USENIX Association.

[112] B. Zhang and D. H. C. Du. Nvlsm: A persistent memory key-value store using log-structured merge tree with accumulative compaction. *ACM Trans. Storage*, 17(3), Aug. 2021.

[113] L. Zhang and S. Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 897–912, Renton, WA, July 2019. USENIX Association.

[114] W. Zhang, X. Zhao, S. Jiang, and H. Jiang. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 194–209, New York, NY, USA, 2021. Association for Computing Machinery.

[115] X. Zheng and R. Rugina. Demand-driven alias analysis for c. *SIGPLAN Not.*, 43(1):197–208, jan 2008.

[116] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen. Dptree: Differential indexing for persistent memory. *Proc. VLDB Endow.*, 13(4):421–434, Dec. 2019.