UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Verifying Correctness of Persistent Memory Programs

Permalink
https://escholarship.org/uc/item/7bh0k9s4

Author
Gorjiara, Hamed

Publication Date
2022

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Verifying Correctness of Persistent Memory Programs

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Hamed Gorjiara

Dissertation Committee:
Professor Brian Demsky, Chair
Professor Guoqing Harry Xu
Professor Nader Bagherzadeh

2022

© 2022 Hamed Gorjiara

DEDICATION

To my parents and my family who always supported me

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **viii**

ACKNOWLEDGMENTS **ix**

VITA **xi**

ABSTRACT OF THE DISSERTATION **xiii**

1 Introduction **1**
1.1 Primary Challenge of Using Persistent Memory 2
1.2 Model Checker 4
1.2.1 Jaaru: Model checking Persistent Memory Programs 6
1.3 Robustness 8
1.3.1 Correctness Criteria for Flush Operations 9
1.3.2 PSan: Checking Robustness with Constraints 11
1.4 Persistency Race 13
1.4.1 Yashme: Detecting Persistency Races 17
1.5 Organization 19

2 Background **21**
2.1 Overview of x86 Persistent Memory Storage 21

3 Model Checker **24**
3.1 Basic Ideas 25
3.1.1 Constraint-Refinement 25
3.1.2 Leveraging Commit Stores for Additional Efficiency 28
3.1.3 System Overview 31
3.2 Model Checking Algorithm 33
3.2.1 Discussion 41
3.3 Evaluation 42
3.3.1 Bug Detection 43
3.3.2 Jaaru Bug Reporting 46
3.3.3 Performance 46

3.3.4 Key Takeaway 48

4 Robustness **49**
4.1 Preliminaries 50
4.1.1 Strict Persistency 51
4.1.2 Robustness Condition 51
4.1.3 Persistent Lock-Free Data Structures 52
4.1.4 Clock Vectors and Sequence Numbers 53
4.2 Basic Ideas 56
4.2.1 Checking Equivalence 56
4.2.2 Supporting Threads 58
4.2.3 Implications for Updating Constraints 62
4.2.4 Supporting Multiple Crash Events 63
4.3 Algorithm 65
4.3.1 Operational Semantics 65
4.3.2 Suggesting Fixes for Robustness Violations 67
4.4 Evaluation 71
4.4.1 Methodology 71
4.4.2 Bug Detection 73
4.4.3 Performance 75
4.4.4 Discussion 75

5 Persistency Race **80**
5.1 Motivation 81
5.1.1 Example Persistency Race 81
5.1.2 Severity of Persistency Races 82
5.1.3 Empirical Validation 84
5.2 Yashme Overview 85
5.2.1 Basics 87
5.2.2 Key Idea: Expanding the Detection Window 90
5.3 Algorithm Preliminaries 93
5.3.1 Persistency Races in Execution Prefixes 94
5.4 Race Detection Algorithm 96
5.5 Implementation 100
5.6 Evaluation 101
5.6.1 Methodology 101
5.6.2 Race Detection 102
5.6.3 Optimization & Performance 105
5.6.4 Bug Reporting and Confirmation 106
5.6.5 Discussion 107

6 Related Work **109**
6.1 Programming Models for Persistent Memory 109
6.2 Crash Consistency Detection 109
6.2.1 Model Checking 110

6.2.2 Persistent Memory Testing Tools 111
6.2.3 Relation of Robustness to Prior Work 112
6.3 Constructive Approaches 114
6.4 Fix Suggestion 115

7 Conclusion **116**
7.1 Summary 116
7.2 Limitations and Future Directions 117

Bibliography **120**

Appendix A Jaaru Artifact Evaluation **137**

Appendix B PSan Artifact Evaluation **145**

Appendix C Yashme Artifact Evaluation **154**

LIST OF FIGURES

Page

1.1 (a) An example of normal program without proper flush operations (b) with proper flush operation. In this example we assumed `x` and `y` are located in different cache lines. 3
1.2 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. 11
1.3 A weakly-persistent execution that reads `r1 = 1` and `r2 = 2` is not robust. . 12
1.4 A persistency race example. We assume `pmobj->val` is initially 0 and both executions are single threaded. `gcc` optimization level `O1` and above generate ARM64 code for this example that can print `0x12345678`. PRIx64 is a macro for printf that prints a 64-bit integer as hex. 14

2.1 An x86-TSO storage system. 22

3.1 Pre-failure execution of a simple PM program. We assume that `x` and `y` reside on the same cache line. The blue line represents the total order in which stores are written to the cache. The red line shows the interval for the last time the cache line containing `x` and `y` may be written back to persistent memory. . . 27
3.2 Post-failure (recovery) execution of the program that reads the value 4 from `x`. This *refines* the interval for the most recent writeback of the cache line to be between the store `x` = `4` and the store `x` = `6`. 27
3.3 An example program with a commit store. 28
3.4 Jaaru system overview. 30
3.5 An example of Jaaru's runtime system. 32
3.6 Algorithm for executing instructions. 35
3.7 Algorithm for evicting store and flush buffers. 35
3.8 Algorithm for BuildMayReadFrom. 37
3.9 Algorithm for DoRead. 38
3.10 The main model checking Algorithm. 40
3.11 Bugs found in PMDK. *Bugs with a * are new bugs.* Only the second bug was reported before in XFDetector [107]. 44
3.12 Bugs were found by Jaaru in every program of RECIPE. *Bugs with a * are new bugs.* 45

3.13 Jaaru's state space reduction. Reported are the number of times Jaaru executes a program (**JExec.**), time Jaaru takes to finish exploration (**JTime**), number of failure injection points (**FPoints**), and the number of program executions Yat needs to eagerly explore pre-failure stores (**Yat Execs.**). 47

4.1 Algorithm for updating clock vectors that track the happens-before relation over stores and sequence numbers that record the TSO order. 54
4.2 An example of non-robust program with missing flush and drain operations. `x` and `y` are initialized to 0. 57
4.3 Constraints for execution of code in Figure 4.2 where r1 = 2 and r2 = 5. . . 58
4.4 `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`? 59
4.5 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`? 59
4.6 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. 63
4.7 A simple concurrent programming language. 65
4.8 Semantics for checking robustness violations. 66
4.9 Reading from a store that is too old. 69
4.10 Reading from a store that is too new. 70
4.11 Source code for bug #9 which leads to unaligned accesses by the program. . 74

5.1 The `Segment::Insert` method from the `CCEH` hashtable. The store to the `key` field commits an insertion into the table. This store is non-atomic and thus a poorly timed crash could cause the key to be partially written. 82
5.2 Example of using `clflush` to flush the store to x. 87
5.3 Example of using `clwb` to flush the store to x. 88
5.4 Example of coherence preventing persistency races. Assume that the variables `x` and `y` reside on the same cache line and that the store to `y` is an atomic release store. 89
5.5 Crash misses window for detecting persistency race using core algorithm. . . 90
5.6 Prefixes of pre-crash execution that are consistent with the post-crash execution. 91
5.7 Prefixes of pre-crash execution that are consistent with the post-crash execution after reading from `y` residing on the same cache line as `x`. 92
5.8 Algorithm for executing instructions. 98
5.9 Algorithm for evicting store and flush buffers. 99
5.10 Algorithm for loads. 100
5.11 The `CCEH::Get` method from the `CCEH` hashtable reads from the non-atomic `key` and `value` fields. 104

LIST OF TABLES

Page

2.1 Summary of reordering constraints in the $Px86_{sim}$ model. 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. These constraints are also used in Raad *et al.* [136]. 23

3.1 System configuration. 42

4.1 Robustness violations. 78
4.2 Execution times for PSan and Jaaru (the underlying model checking infrastructure). PSan incurs minimal overhead compared to Jaaru. 79

5.1 Ubiquity of persistency races. 83
5.2 Races found in CCEH, FAST_FAIR, and RECIPE benchmarks. 103
5.3 Races found in PMDK, Redis, and Memcached. 104
5.4 # races detected w/ and w/o prefix-based expansion for a single execution on RECIPE, PMDK, Memcached, and Redis benchmarks, as well as execution times for both Yashme and Jaaru (the underlying checking infrastructure). Yashme incurs minimal overhead compared to Jaaru. 106

6.1 Comparison with other tools; robustness subsumes ordering heuristics/conditions used in existing tools. 113

ACKNOWLEDGMENTS

First and foremost, I want express my gratitude regarding getting a chance to pursue my Ph.D. at UC Irvine, one of the greatest public school in the United States. Since I was a little child, I always wanted to make a difference and receive the highest education from a great school. I'm so grateful that I had a chance to fulfill this childhood dream and hopefully my contributions would persuade other fellow researchers to accomplish great achievements in the future.

Next I would like to sincerely thank Professor Brian Demsky, my advisor, 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 patiently teaching me how to think critically, do research, design and develop big systems, and present ideas. Definitely, without his help, I would not be able to successfully finish my Ph.D. degree. I'm so happy there are professors like him that dedicated their life to train good students to make a great impact on the society and make our world a better place.

Also, I would like to thank Professor Guoqing Harry Xu who tirelessly co-advised me in my research in past six years. I was greatly persuaded by his research vision and research ideas and without his help, I would not be able to successfully defend my Ph.D. degree.

I would like to thank professors and fellow researchers who helped me along the way. I thank Professor Aparna Chandramowlishwaran, Professor Nader Bagherzadeh, and Professor Sang-Woo Jun for serving as my candidacy exam committee. I also thank Weiyu Luo and Alex Lee who have been my student co-authors. Finally, I thank Peizhao Ou, Rahmadi Trimananda, Zachary Snyder, Weiyu Luo, Ahmed Al Nahian, Seyed Amir Hossein Aqajari, and many others who have been colleagues.

I would like to thank Hongkun Yang and Ivy Liu who have selected me for an internship opportunity at Google. I thank them for their guidance and mentorship during my internship and helping me grow my industrial coding skills. I am so grateful this internship experience at Google that persuaded me to join the industry after my graduation.

I would like to thank the ACM SIGPLAN for granting permission to include content that has been previously published in conference proceedings into this dissertation. I also thank ASPLOS and PLDI communities for accepting my research and giving a change to present it to the fellow researchers. I would like to thank Jay Lorch, Baptiste Lepers, Satish Narayanasamy and all anonymous reviewers who help me improve my research papers.

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.

My family has been the greatest supporter of my PhD journey. In particular, I am grateful for my parents, Hosnieh Mostafaee and Mohammadbagher Gorjiara, who have been my greatest supporter from the very beginning since I was child; my siblings, Mina Gorjiara, Bita Gorjiara, Tina Gorjiara, and Mohammad Gorjiara who always influenced to be successful and be a better version of myself; my nephews and niece, Aryan Fartash, Ryan and Arian Reshadi; my brothers and sisters in law, Siamak Fartash, Saeed M. Ali, Mehrdad Reshadi, and Hoorieh Hosseini; my uncle, George Sinaki; my cousin, Kiana Sinaki. I would like to thank my girl friend, Shawdee Mashayekh and her lovely family who made me feel like home in the U.S. and supported me through my PhD journey. I would like to thank my friends Mahdi Rafati, Ahoramazda Aryazand, Dordaneh Ashouriha and her kind family, Hooman Hashemi, Mohammad Jafari, Sepand Fallah, Ashkan Niktab, Alireza Samadian, Soroush Allahyari, Saeed Mehrabi, and Arman Asgharinia that their presence made the past 6 years more pleasant.

Finally, I would like to thank University of California Irvine for granting 2-year fellowship. Also, I would like to thank National Science Foundation (NSF) and Office of Naval Rsearch (ONR) for fully fund my research via grants CNS-1703598, OAC-1740210, CNS-1763172, CNS-1907352, CNS-2006437, CCF-2006948, CNS-2007737, CCF-2102940, CNS-2128653, CNS-2106838, N00014-16-1-2913, and N00014-18-1-2037. My doctoral research and the work presented in this dissertation would not be possible without the support from these institutions.

VITA

Hamed Gorjiara

EDUCATION

Doctor of Philosophy in Computer Engineering **2022**
University of California, Irvine *Irvine, California*

Master of Science in Computer Engineering **2019**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computer Engineering **2016**
University of Tehran *Tehran, Iran*

Work EXPERIENCE

Graduate Research Assistant **2016–2022**
University of California, Irvine *Irvine, California*

Software Engineering Intern **2021–2021**
Google LLC *Irvine, California*

Software Developer (Part-time) **2014–2016**
Research Institute of University of Tehran *Tehran, Iran*

TEACHING EXPERIENCE

Java instructor **2016–2016**
ACM branch of University of Tehran *Tehran, Iran*

Teacher Assistant **2014–2016**
University of Tehran *Tehran, Iran*

REFEREED JOURNAL PUBLICATIONS

Satune: synthesizing efficient SAT encoders **2020**
Proceedings of the ACM on Programming Languages (OOPSLA)

REFEREED CONFERENCE PUBLICATIONS

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

Yashme: detecting persistency races **February 2022**
Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems

Jaaru: Efficiently model checking persistent memory programs **April 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.

Yashme `https://plrg.ics.uci.edu/yashme/`
A tool for detecting persistency races in persistent memory programs.

Jaaru `https://plrg.ics.uci.edu/jaaru/`
A tool for model checking persistent memory programs.

Satune `https://plrg.ics.uci.edu/satune/`
A tool for synthesizing efficient SAT encoders.

ABSTRACT OF THE DISSERTATION

Verifying Correctness of Persistent Memory Programs

By

Hamed Gorjiara

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2022

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. Ensuring that these programs are crash-consistent (i.e., power failures) is a major challenge. Stores to persistent memory are not immediately made persistent — they initially reside in processor cache and are only written to PM when a flush occurs due to space constraints or explicit flush instructions. It is more challenging to test crash consistency for PM than for disks given the PM's byte-addressability that leads to significantly more states. Most of the existing state-of-the-art testing tools require heavy user annotations, report violations that may not correspond to actual bugs, do not test the recovery procedure, and rely on a test suite to cover all test scenarios.

This dissertation describes three different testing tools to verify the crash consistency of persistent memory programs:

1. Jaaru: a fully-automated and ultra-efficient model checker for PM programs. Key to Jaaru's efficiency is a new technique based on constraint refinement that can reduce the number of executions that must be explored by many orders of magnitude. This

exploration technique effectively leverages commit stores, a common coding pattern, to reduce the model checking complexity from exponential in the length of program executions to quadratic.

2. PSan: a tool introducing robustness as a sufficient correctness condition to ensure that program executions are free from bugs resulting from missing flushes. PSan implements an algorithm for checking robustness. This tool 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.

3. Yashme: a tool that can detect a novel class of crash consistency bugs for persistent memory programs, which we call persistency races. Persistency races can cause non-atomic stores to be made partially persistent. Persistency races arise due to the interaction of standard compiler optimizations with persistent memory semantics. A major challenge is that in order to detect persistency races, the execution must crash in a very narrow window between a store with a persistency race and its corresponding cache flush operation, making it challenging for naive techniques to be effective. Yashme overcomes this challenge with a novel technique for detecting races in executions that are prefixes of the pre-crash execution. This technique enables Yashme to effectively find persistency races even if the injected crashes do not fall into that window.

These testing frameworks were capable of finding many bugs in well-tested applications ranging from persistent data structures to real-world frameworks. These bugs are reported to the developers of these frameworks and most of them are confirmed and the corresponding fixes are available on their Github repositories.

Chapter 1

Introduction

Persistent memory (PM) technologies, such as phase change memory (PCM) [97, 154, 165], resistive random-access memory (RRAM) [153], Spin-Transfer Torque memory (STT-MRAM) [90], or 3D XPoint [32], promise to combine the performance and flexibility of DRAM with the persistency of flash storage. Persistent memory revolutionizes the storage-memory hierarchy [124, 133, 75]. This technology became commercially available with Intel's release of Optane DC Persistent Memory [74]. In terms of pricing, such PM technologies are even cheaper than DRAM per GB of capacity [4]. 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. Persistent memory can potentially change the way programs manipulate data structures to achieve greater performance; with PM, programs can use a single copy of a data structure both as an in-memory working data structure and as a persistent store of the data, eliminating the overhead of serialization and deserialization of data in the program executions. The low latency and durability of PM have spurred the development and redesign of file systems [40, 91, 92, 110, 150, 156, 158, 159, 28, 83], databases [7, 93, 35, 113, 127],

log-based systems [103, 27, 81, 102, 50, 69], key-value stores [26, 155, 157, 84, 166, 170, 64], and concurrent DRAM indexes [25, 164, 98, 129, 149, 172, 19] for persistent memory.

1.1 Primary Challenge of Using Persistent Memory

While persistent memory offers great performance, it is very challenging to write frameworks for persistent memory that are both correct and efficient [126, 30, 36, 108, 107, 85, 22, 31, 66, 112, 114]. The main challenge is raised by the fact that stores are not immediately written to persistent memory after getting issued by CPU. These stores are initially written to volatile cache. The persistent memory is only eventually updated when the cache line is written back. The cache system might decide to write back the modified cache lines to persistent memory in any arbitrary order due to space constraints. Writing correct PM-based applications is especially challenging because the cache and CPU registers are volatile and their contents vanish after failures, *e.g.*, due to a system crash or a power failure. Consequently, a poorly-timed crash can yield data inconsistency in the persistent data structure.

To elaborate on this problem, let's consider a program example in Figure 1.1-a. For this program, we assume variables `x` and `y` are located on different cache lines. When this program runs, the stores `x = 1` and `y = 2` update the cache lines corresponding to the variables `x` and `y`, but they are not written to persistent memory yet. If the program crashes in this state, the data on the modified cache lines are lost since the cache is volatile. Consequently, none of these stores can become persistent and be observed by the post-crash execution. Another scenario is that after the cache is updated with the store `y = 2`, the cache becomes full. Depending on the cache eviction policy, the cache line containing the variable `y` may be written back to persistent memory before the cache line containing the variable `x`. In this scenario, the persistency order becomes different from the volatile memory order defined in the program in Figure 1.1-a. This can be problematic if the system failure happens before

the cache line containing the variable `x` becomes persistent. In this scenario, if the post-crash execution reads from variable `x` and `y`, it observes the value `2` for `y` without observing the store `x = 1`. This can cause data inconsistency bugs in the program.

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

(a) Incorrect program.

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

(b) Correct program.

Figure 1.1: (a) An example of normal program without proper flush operations (b) with proper flush operation. In this example we assumed `x` and `y` are located in different cache lines.

Processor manufacturers have introduced new instructions such as `CLWB` and `SFENCE` on x86 [73], and `DC CVAP` on ARM [6], to force cache lines to be written back to persistent memory. Thus, developers can use these instructions to enforce the persistency order to be as same as the program order. Developers of PM programs need to carefully use these instructions since a missing flush instruction can make a program vulnerable to crash consistency bugs. In addition, these instructions have performance overhead, and excessively using them yields performance degradation. Therefore, using these instructions correctly is very challenging; it requires both subtle reasoning about the ordering of memory operations and attention to detail to not miss persisting any of the many stores a program may perform. Moreover, testing the correctness of persistent storage code *w.r.t.* failures is challenging. Exposing a bug requires that the machine fails at a specific instruction and depends on the state of the cache before the failure.

These challenges prompted us to work on techniques and approaches to improve the reliability of applications and frameworks that use persistent memory. In the rest of this chapter, we describe three key challenges in developing applications for persistent memory. Then, for each of them, we describe our solution and the primary idea behind it to address the challenges and to identify various crash consistency bugs in persistent memory programs.

1.2 Model Checker

The problem of PM consistency has received much attention. There is a line of recent work on testing/dynamically checking a PM program to find consistency-related bugs. XFDetector [107] uses a finite state machine to track the consistency and persistency of persistent data by implementing a shadow PM, and with the help of user-provided annotations to identify commit variables. XFDetector only shows violations of programming patterns for consistency and does not generate an execution that shows how the violation can actually lead to a bug. Moreover, it only supports scenarios in which a single failure occurs and ignores the possibility of the occurrence of failures in the post-failure execution. Different from XFDetector [107], PMTest [108] computes the persistency status of writes and ordering constraints between writes. Developers must annotate the code with checking rules to ensure that the code establishes the correct persistency and ordering properties. PMTest only executes the pre-failure portion of the program and thus does not test failure recovery, which may also contain bugs.

Pmemcheck [85] is a binary rewriting tool that checks how many stores were not made persistent and detects memory overwrites, redundant flushes, and unnecessary flushes [85]. Similar to PMTest, Pmemcheck also requires user annotations and only executes the pre-failure execution.

These testing-based bug-finding tools suffer from two major drawbacks: (1) They need users to add extra annotations for various cache line flushing properties, which not only incur burdens on users but also are error-prone themselves. Consequently, if the developer misses an annotation or adds an incorrect annotation, the tool will have false negatives and miss real bugs or have false positives and report bugs that are not real. (2) Violations they report are with respect to design principles and may or may not correspond to actual bugs, *e.g.*, certain tools report data has not been flushed. However, in some cases, the data may never

be accessed in future executions. Thus, the absence of a flush is a false positive that does not represent a real bug. (3) Most of these testing tools do not test the recovery procedure and dismiss the possibility of occurrence of system crashes in the recovery code. (4) Some of these tools are testing-based and only test the running example and they require a test suite that covers all testing scenarios to fully test the program. These drawbacks call for techniques such as model checking that can exhaustively explore states without needing manual effort and provide strong witnesses (*e.g.*, executions) for bugs exposed.

Model checking has been used extensively in the systems community (*e.g.*, EXPLODE [162], FiSC [163], or SAMC [99]) to find bugs in file/storage systems. However, there are several fundamental differences between the file system bug problem and persistent memory crash consistency problem that preclude direct application of existing model checkers in the PM setting: (1) disks have a fundamentally different programming interface than PM ; updates to a disk block are only made upon making an explicit write request, (2) disks have a larger block size and therefore there are fewer possible states to enumerate, and (3) operating systems receive explicit notifications of when disk blocks are written. All of these factors combined indicate that the state space to be explored for model checking disks is significantly smaller than that for PM programs.

In fact, a recent technique Yat [95] attempts to use an eager model checking approach to enumerate all possible post-failure memory states for a PM program before it is aware of what parts of the state the post-failure execution will read from. Since the number of memory states that must be explored grows exponentially with the number of stores that have not been flushed to memory, Yat cannot scale. For example, consider the common scenario of code that allocates a cache line aligned array of n 64-bit integers, initializes the data, and crashes right before flush operations for that array. This array spans $n/8$ cache lines and the persistent memory copy of each cache line has 9 possible states (*i.e.*, the initial value and the state after each of the 8 writes). Therefore, persistent memory has $9^{n/8}$ possible states that

Yat must explore.

1.2.1 Jaaru: Model checking Persistent Memory Programs

We develop Jaaru [62], a *fully-automated* and *ultra-efficient* model checker for PM programs that achieves many *orders-of-magnitude* reductions in the number of states that must be explored, compared to eager techniques such as Yat. It does *not* require any user annotation; as a model checker, Jaaru exhaustively explores all possible states and can potentially find more bugs than testing-based techniques.

Key to Jaaru's efficiency is a *constraint-refinement based technique* that effectively leverages *commit stores* ; a common programming practice in data structure implementations to drastically reduce the space of executions. We elaborate on this insight below.

As stated above, a major challenge in model checking PM programs is the enormous post-failure state space the model checker must explore ; a store writes a value into the cache, and the value is not persisted until the cache line is flushed. However, when a failure occurs, it is unclear whether a cache line has been flushed yet, leading to a large number of possibilities that the model checker must explicitly enumerate.

To solve this problem, our *major insight* is that we can exhaustively explore all executions by enumerating only a subset of post-failure states using *constraints on the time at which a cache line was previously flushed*. A `clflush` or `clflushopt` instruction flushes a cache line, imposing a constraint on the possible values that a persistent variable can have after the failure. Jaaru *builds* such constraints during a pre-failure execution and *refines* them during a post-failure execution (see §3.1.1). Leveraging these constraints in partial order reduction [46, 168] enables Jaaru to explore exactly one post-failure state for each *equivalence class* of post-failure executions, defined by which pre-failure stores are read by post-failure

loads.

To effectively leverage this insight, we made an observation that there are often many stores that have not been flushed out to persistent memory, PM programs often record in some fashion, using a commit store, whether data is in a consistent state (see § 3.1.2). For example, when adding a subtree to a node, the store of the node pointer to the subtree is a commit store. Post-failure PM programs then read from this commit store to determine whether data is consistent. This is a common practice in data structure implementations (1) because the information about consistency also provides a reference to where the data is stored (*e.g.*, if the pointer from the node to subtree is null, the subtree is not persisted; otherwise, it can be found by following the pointer) and (2) for efficiency purposes. Such checks explicitly prevent the post-failure execution from accessing many unflushed stores (*e.g.*, if the pointer is null, the program cannot access any data protected by the pointer).

This pattern offers an opportunity for us to *not* explicitly enumerate all possible states at a failure ; *lazily* enumerating the stores *read by the actual loads* in the post-failure execution, as opposed to *eagerly* enumerating all of them, reduces the number of executions to be explored from *exponential* in the length of the program execution to *linear* (see §3.1.2). This observation leads to the *lazy exploration approach* used in Jaaru, which does not enumerate stores until loads are executed in the recovery code.

Note that leveraging such a programming pattern leads to efficiency, but has nothing to do with the thoroughness of the state search ; Jaaru always exhaustively explores all the non-determinism that arises from the persistency of cache lines. As a result, **Jaaru does not generate any false positives or negatives** ; it reports *all* bugs *w.r.t.* an input and any bug it reports must be a real bug. For programs that do not obey such a programming idiom (*e.g.*, the recovery code directly reads the data without checking consistency), Jaaru would not miss any bug, but it would certainly spend more time on state exploration. In practice, however, Jaaru is often still efficient because PM programs are extremely unlikely

to read from many non-flushed cache lines.

Usage scenarios. Despite the aforementioned advantages, model checking is not a silver bullet for bug finding in PM programs. For example, even though Jaaru is orders of magnitude more efficient than existing model checkers such as Yat, Jaaru still needs to execute a program many times (*e.g.*, between 24 and 891 in our experiments) to fully explore the state space, taking a large amount of time for checking. Compared to testing tools such as PMTest and XFDetector, Jaaru is able to find more bugs, in a completely automated fashion. However, it has difficulty checking programs such as Redis that interact with the outside world and whose non-determinism from the network would require deterministic replay for a model checker to work. As such, the best use case for Jaaru is to exhaustively check widely-used libraries such as PMDK, finding as many potential bugs as possible before their release, while non-exhaustive tools such as PMTest and XFDetector can scalably check large programs and find bugs only when they are triggered in tests.

Chapter 3 elaborates on the algorithm behind Jaaru and its implementation. This chapter also provides a rigorous evaluation of Jaaru on popular persistent memory benchmarks.

1.3 Robustness

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 [30, 18, 161, 51, 53, 104, 33, 89, 151, 14, 167, 137, 52, 115], locks [9, 20, 68, 77, 105], or synchronization-free regions [59] 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 [95, 85, 108, 107, 126, 62, 76, 106, 67, 39, 125, 49, 63] 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 [125] 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.

1.3.1 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 [131] is such a persistency model that ensures that the "persistency memory order is identical to volatile memory order". Most hardware persistent memory specifications do not provide strict persistency. However, Intel has developed an optional new feature called enhanced *Asynchronous DRAM Refresh* (eADR) that relies on stored power to flush the contents of the cache to persistent memory during a power failure. Consequently, eADR-enabled persistent memory provides strict persistency. However, eADR functionality cannot be relied upon because it requires the system vendor to provide additional stored energy hardware such as a battery. Due to these specialized requirements on system vendors, it is expected that many Intel PM systems will not provide strict persistency for the foreseeable future according to our email discussions with Intel engineers.

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

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

Example. To illustrate, consider the example from Figure 1.2 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

```
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 1.2: 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.

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.

1.3.2 PSan: Checking Robustness with Constraints

We develop PSan [61], 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 PSan. 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 1.3: A weakly-persistent execution that reads `r1 = 1` and `r2 = 2` is not robust.

To illustrate, consider the executions in Figure 1.3, 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 1.3, PSan determines a flush instruction must be inserted after `x = 2` to fix the robustness violation.

Chapter 4 defines robustness as a sufficient correctness condition and elaborates on algorithm PSan uses to identify robustness violations in the programs. This chapter also describe how PSan localizes a bug and suggests the exact fixes for the stores that are missing flush and fence instructions. At the end of Chapter 4, we describe our evaluation of PSan on 3 real-world applications and a collection of data structures for persistent memory.

1.4 Persistency Race

This section of the dissertation presents a new class of persistent memory bugs, referred to as *persistency races*. Persistency races stem from the fact that most programming language specifications provide compilers with the freedom to assume other threads will not observe a non-atomic store until a synchronization operation. Compilers can for example implement a non-atomic store with multiple store instructions. This is often referred to as *store tearing*. For example, given an architecture having 16-bit store instructions with immediate fields, the compiler might use two 16-bit store-immediate instructions to implement a 32-bit store. Although it is rare that compilers introduce these optimizations, it is enough of a concern that **both PMDK developers and the Linux Kernel developers take care to avoid**

it. Indeed, the Linux kernel mailing list provides several examples [37] of modern compilers tearing non-atomic stores even when they are aligned, word-length stores.

Compilers can also introduce store tearing via other optimizations. Mainstream compilers commonly rewrite code that copies or initializes several contiguous fields into calls to the libc functions `memcpy`, `memmove`, or `memset`. These functions do not guarantee 64-bit atomicity and can hence result in store tearing. Store tearing creates the possibility that a poorly timed crash can cause non-atomic stores to be made partially persistent. A post-crash execution can then potentially read values that mix bytes from multiple different store operations. This could for example cause a post-crash execution to read an invalid array index, leading to further corruption.

Store tearing is not the only potential danger of persistency races. Compilers can also stash temporary values in the memory location safely assuming that data race freedom means that other threads will not see these values. Crashes can result in such temporary values being made persistent.

```
1 pmobj->val = 0x1234567812345678;
2 //crash here
3 flush(&pmobj->val);
```

(a) Pre-Crash Code

```
1 if (pmobj->val != 0) {
2   printf("0x%" PRIx64 "\n",
3     pmobj->val);
4 }
```

(b) Post-Crash Code

Figure 1.4: A persistency race example. We assume `pmobj->val` is initially 0 and both executions are single threaded. `gcc` optimization level `O1` and above generate ARM64 code for this example that can print `0x12345678`. PRIx64 is a macro for printf that prints a 64-bit integer as hex.

Figure 1.4 presents an example of persistent memory code with a persistency race. Figure 1.4a and Figure 1.4b show the code snippets executed before and after the crash, respectively. Suppose that the machine experiences a power failure immediately after line 1 in Figure 1.4a. Since the store to the `val` field is non-atomic, the compiler is free to implement this store with multiple store instructions. Thus, it is possible for only some of the bytes of this store

to be made persistent. When the `val` field is read in line 3 of Figure 1.4b, the post-crash execution can read a value that is some combination of the bytes from the previous value and the newly stored value. This concern is not theoretical—the ARM64 backend of gcc generates code for this program that could print `0x12345678`.

A program has a persistency race if there exist a pre-crash execution E_{pre} and post-crash execution E_{post} such that (1) a load l in E_{post} reads from a non-atomic store s in E_{pre} and (2) the store s is not *persistency ordered* before the load of it in E_{post}. To provide an example of persistency ordering, if the pre-crash execution explicitly flushes a store s before it crashes, then the store s is persistency ordered before any loads that might read from it in the post-crash execution.

Persistency vs. Data Race. Persistency races are similar in spirit to data races because both persistency races and data races violate assumptions made by compilers and thus can break the abstraction of a language-level store writing the specified value to memory. Several tools have been designed to detect data races in code that uses standard lock-based concurrency control [42, 43, 44, 70, 111, 171, 141]. These tools generally take one of two approaches: (1) they verify that all accesses to shared data are protected by a locking discipline or (2) they compute a happens-before relation to detect concurrent conflicting accesses.

While persistency races are similar to data races, there are important fundamental differences between the two as each persistency race involves three distinct events: (1) the racing store in the pre-crash execution, (2) the crash event against which the store races, and (3) a race-observing load in the post-crash execution that observes the effects of the race. This differs from data races that consist of two memory operations that race against each other. Persistency races exist even in single-threaded programs. Intuitively, in a persistency race, a pre-crash execution thread races with the crash and a post-crash thread observes the effects of the race.

Researchers have developed techniques for detecting races in interrupt-based code [152, 29]. That body of work focuses on ensuring that interrupts are disabled when code performs a memory access that could potentially conflict with memory accesses in an interrupt handler. The focus on analyzing interrupt code that relies on disabling interrupts means that the analysis techniques are not applicable to persistent memory where a crash can occur at any point.

Most existing PM bug finding tools use techniques that fundamentally cannot detect persistency races because they just validate that stores are flushed or performed in a specific order. The one exception is that model checking tools could conceptually be adapted to find persistency races by splitting the stores into single byte stores at the cost of an exponential increase in the number of executions that must be explored. Dynamic instrumentation frameworks [85, 39, 107, 95] can observe actual store instructions in the binary. The primary challenge in detecting persistency races with these frameworks is that they cannot infer whether two stores in the binary were originally one source-level store nor can they infer which stores were atomic at the source level, thus it is not possible for these tools to directly detect a persistency race. However, if these frameworks explore the correct execution, they can potentially observe a crash caused by a persistency race, *e.g.*, a segmentation fault caused by accessing a partially persisted pointer. Moreover, these tools can give no warning for stores that could potentially be torn in the future. XFDetector [107] uses a finite state machine to track the consistency and persistency of persistent data. XFDetector finds cross-failure races, which are defined as loads that read from locations that were not persisted before a failure. Cross failure races are different from persistency races in that cross failure races model normal stores as effectively atomic and do not consider the possibility that due to compiler optimizations a store may made partially persistent. Cross failure race detection cannot detect persistency races because it does not model the effects of cache coherence or the difference between atomic and normal memory operations. XFDetector is limited to detecting cross failure races in the given execution and cannot detect cross failure races in

any other potential executions.

There is a large body of work on finding atomicity violation bugs [82, 130, 169, 109] which are fundamentally different from persistency races. Atomicity violations occur when code written by developers performs many operations that were intended to be atomic, but are not atomic because of a bug. However, persistency races violate language-level store abstractions because of a race with a crash event.

1.4.1 Yashme: Detecting Persistency Races

Detecting persistency races is extremely challenging as it requires reasoning about the cache behavior of a program, *e.g.*, where the crash occurs, where a cache line is flushed, and which pre-execution store the post-execution code reads from. On one hand, data race detectors focus on reasoning about mutual exclusions, *not* on cache behavior. As a result, none of the data race detection algorithms are directly applicable to detecting persistency races. On the other hand, existing persistent memory bug detectors reason about the timing of cache line flushes, but rely on effective test cases and appropriate crash events to find bugs. Detecting a persistency race with respect to a store, however, requires the crash to fall into a small window of execution after the store and before the (explicit or implicit) cache line flush. Such a strict requirement dictates aggressively injecting crash events in a great number of executions and exhaustively exploring thread schedules, which is impractical.

Model checkers (*e.g.*, Yat [95] and Jaaru [62]) and fuzzers (*e.g.*, PMFuzz [106]) automatically explore many possible cache states, but their effectiveness depends on the selected thread schedule—they would not detect a persistency race if the thread schedule causes the window to close; as such, they suffer from the same weakness as other bug detectors.

Clearly, a major challenge in devising an effective persistency race detector is where to inject

the crash event, *i.e.*, the nature of a persistency race is that the pre-crash event *races* with the crash event.

Key Insight. Key to the success of Yashme [63] is a shift of focus from generating race-manifesting crash events to *generalizing/expanding the executions under a small number of crash events to make races observable.* In other words, Yashme does *not* rely on perfect crash events to trigger races; instead, given a target crash event, Yashme runs the program, obtains its pre-crash execution, and *expands* it to derive many executions that can also be used to detect races.

Yashme's approach is similar to model checking in that both derive many executions from existing ones. However, Yashme expands an existing pre-crash execution using a set of *prefix constraints*, which guarantee that the derived executions are *consistent* with the original execution with respect to the post-crash execution. In particular, if the post-crash execution reads from a store in the original pre-crash execution e, it must read from the same store in any executions derived from e. Expansion enables Yashme to detect races in derived executions without actually executing them, while model checkers incur the overhead of actually executing their derived executions.

In other words, any execution that shares a common prefix (starting at the store) with e and does not later perform and persist stores that overwrite locations read by the post-crash execution is consistent and can be used to detect persistency races. Prefix-based derivation significantly expands the race-detection scope, enabling Yashme to find more bugs even than a technique that injects a crash event before every fence instruction. As a result, Yashme has found persistency bugs in **all but one of the programs we have experimented with**.

Chapter 5 formalize persistency race and describes the baseline algorithm as well as prefix algorithm that Yashme uses to detect persistency races in the program. Also, this chapter reports an evaluation of Yashme on three real-world applications and a collection of data

structure for persistent memory. In total, Yashme found 24 persistency races in these benchmarks.

1.5 Organization

The rest of this dissertation is structured as follows:

1. Chapter 2 describes background of X86 persistency model
2. Chapter 3 describes our observation regarding the commit store pattern in persistent memory and elaborates on Jaaru's constraint-based refinement technique to leverage this patter. This chapter also discusses details about the implementation of Jaaru and evaluation of Jaaru on a Recipe benchmarks and PMDK where Jaaru found 18 new bugs in them.
3. Chapter 4 presents the notion of robustness as a sufficient criterion in to assure correct usage of flush and fence instructions in persistent memory applications. This chapter formalizes robustness and provides details on algorithms used by PSan to localize persistency bugs and provide the corresponding fixes for each bug. At the end, this chapter evaluates PSan's bug finding capabilities on a collection of data structure and three popular real-world benchmarks for persistent memory.
4. Chapter 5 presents and formalizes a new class of persistency bugs that we call it persistency race. This chapter introduces an algorithm to expand the detection window to detect persistency races in executions without exploring them. At the end, this chapter evaluates Yashme on a collection of data structure and three real-world benchmarks where Yashme found 19 persistency races in these benchmarks.
5. Chapter 6 discusses existing literature and compares prior work with Jaaru, PSan, and

Yashme.

6. Chapter 7 concludes this dissertation and discuss limitations of the presented tools that can be addressed as future work.

Chapter 2

Background

This chapter briefly overviews the Intel-x86 persistency semantics following the Px86sim model in Raad *et al.* [136] which is supported by Jaaru, PSan, and Yashme.

2.1 Overview of x86 Persistent Memory Storage

In this section, we overview the Intel-x86 persistent storage system. We refer interested readers to the $Px86_{sim}$ model in Raad *et al.* [136]. For our purposes, the differences between Raad *et al.* [136] and Khyzha [86] 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 in any of the presented tools in this dissertation. Figure 2.1 presents a graphical overview of the x86-TSO storage system. Each core/thread on x86 has a store buffer that buffers stores to the cache to hide the store latency. The store buffers implement bypassing — when a core performs a load, the core checks

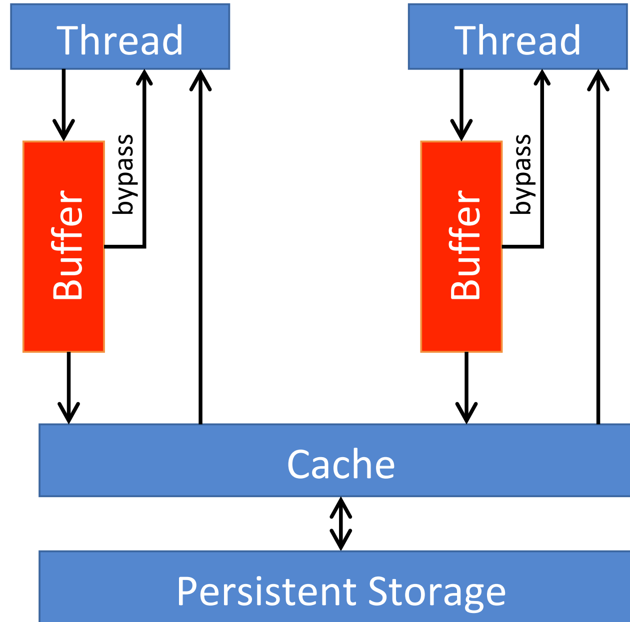


Figure 2.1: An x86-TSO storage system.

whether there is a store to the same address in its local store buffer. If so, it returns the value written by the most recent such store. Effectively, this allows the local core to observe the effect of a local store before that store becomes visible to other cores. The memory fence instruction `mfence` waits for the store buffer to be empty before future instructions can be executed. Locked `RMW` instructions also clear the store buffer before future instructions can be executed.

Stores in the store buffer are written to the cache in the order they were executed — they are written to the cache in a total order and all other threads/cores observe these stores in that same order. The cache is volatile — a power loss event will cause cached data that has not yet been written back to persistent storage to be lost. Under normal execution, 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. The three such instructions are: (1) the flush cache line instruction `clflush` that flushes a cache line, (2) the optimized flush cache line instruction `clflushopt`,

Table 2.1: Summary of reordering constraints in the $Px86_{sim}$ model. 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. These constraints are also used in Raad *et al.* [136].

Later in Program Order (columns); Earlier in Program Order (rows)

	Re	Wr	RMW	mf	sf	clflushopt	clflush
Read	✓	✓	✓	✓	✓	✓	✓
Write	✗	✓	✓	✓	✓	CL	✓
RMW	✓	✓	✓	✓	✓	✓	✓
mfence	✓	✓	✓	✓	✓	✓	✓
sfence	✗	✓	✓	✓	✓	✓	✓
clflushopt	✗	✗	✓	✓	✓	✗	CL
clflush	✗	✓	✓	✓	✓	CL	✓

and (3) the cache line write back instruction `clwb`. Each of these instructions takes as input the address of the cache line and flushes that line.

A key difference between these instructions is how they can be reordered across other instructions. Table 2.1 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 flushed to 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 the `clflushopt` instruction [136], and thus we treat them identically in our discussions.

Chapter 3

Model Checker

Recall from Chapter 1, testing persistent memory programs are difficult since the program has to crash at the specific instruction and it depends on the state of the cache before the crash event. This chapter elaborates on Jaaru implementation, an efficient model checker for persistent memory. In fact, this chapter makes the following contributions:

- **Automated and Efficient Model Checking:** It elaborates on the developed model checker based on a novel partial order reduction algorithm that uses *constraint refinement* and leverages *commit stores*.
- **Support for Simulating Multiple Failures:** Failures during recovery have the potential to corrupt data structures. Testing for bugs in recovery procedures requires the ability to generate multiple failure events — *e.g.*, one failure event to cause the initial recovery and the second failure event during the recovery procedure. This chapter presents the first tool that supports simulating the effect of an arbitrary number of failure events to find bugs in recovery code.
- **Full TSO Support:** Jaaru incorporates a full simulation of the underlying TSO memory model including support for store buffering, buffering flush operations, and

buffering `sfence` operations.

- **Implementation:** It describes the implementation of Jaaru model checker. Jaaru has an LLVM compiler frontend to instrument programs and Jaaru is implemented as a runtime library. Jaaru simulates the x86-TSO memory model and provides full support for multi-threaded PM programs.

- **Evaluation:** It provides a discussion on evaluation of Jaaru with PMDK [33] and RECIPE [98]: Jaaru is effective at finding persistency bugs in our benchmark set. Jaaru finds 18 new correctness bugs in extensively studied PM programs, while PMTest and XFDetector finds only 1 and 4 correctness bugs, respectively.

3.1 Basic Ideas

Recall that prior work (*e.g.*, Yat [95]) on model checking persistent memory programs eagerly enumerates all possible post-failure states of persistent memory. As the number of states grows exponentially with the amount of data that has not been flushed, this approach can easily have scalability problems. Such eager approaches will explore many post-failure states that yield identical post-failure executions in which the loads read from the same stores. Dynamic partial order reduction (DPOR) [1, 46, 168] is a popular technique that can determine that these states produce the same execution, and instead explore the equivalent post-failure executions once.

3.1.1 Constraint-Refinement

Traditional DPOR techniques do not consider the effect of cache line flushes and volatile memory. Naïve adaptation of these techniques in our setting would lead to the exploration

of many states that are *not* possible due to the use of instructions such as `clflush` that explicitly flush cache lines.

To reduce search space, our first idea is to use `clflush` instructions to infer constraints on the *last time each cache line was written back to persistent memory* in a pre-failure execution and refine these constraints in a post-failure execution to narrow down when a cache line became persistent. For example, when a `clflush` instruction leaves the store buffer, it forces the cache line to be written back to persistent memory. That same cache line can later be written back to persistent memory due to space constraints in the cache. Hence, the `clflush` instruction essentially sets a constraint that the last time the corresponding cache line is written back to memory must be *after* the `clflush` instruction exits the store buffer.

Figure 3.1 illustrates the application of this idea on an execution prior to a failure. The program executes the instruction sequence on the left-hand side prior to the failure. The blue line shows the order that stores were written to the cache. Both variables x and y are located in the same cache line. After the program executes the stores $\texttt{y} = \texttt{1}$ and $\texttt{x} = \texttt{2}$, it performs a `clflush` instruction. This instruction flushes the cache line that holds `x` and `y` to persistent memory. At this point, Jaaru computes that the cache line for `x` and `y` was most recently flushed during the interval $[\texttt{clflush}, \infty)$ as represented by the red line in Figure 3.1. After the `clflush`, the program performs the stores $\texttt{y} = \texttt{3}$, $\texttt{x} = \texttt{4}$, $\texttt{y} = \texttt{5}$, and $\texttt{x} = \texttt{6}$. Finally, power is lost and the program fails. The red interval indicates that when the machine is powered back up, the persistent storage may have the values 2, 4, and 6 for the variable `x`.

Note that there are constraints between the values for variable `x` and those for `y` since they share a cache line. For example, it is not possible for the post-failure state of the persistent memory to have $\texttt{y} = \texttt{1}$ and $\texttt{x} = \texttt{6}$, because the store $\texttt{y} = \texttt{5}$ is ordered between $\texttt{y} = \texttt{1}$ and $\texttt{x} = \texttt{6}$. To ensure that variables that share a cache line have consistent values, Jaaru *refines* these intervals using the values observed by loads during the recovery execution. Figure 3.2 shows a post-failure execution. This execution reads the value 4 from the variable `x`. This

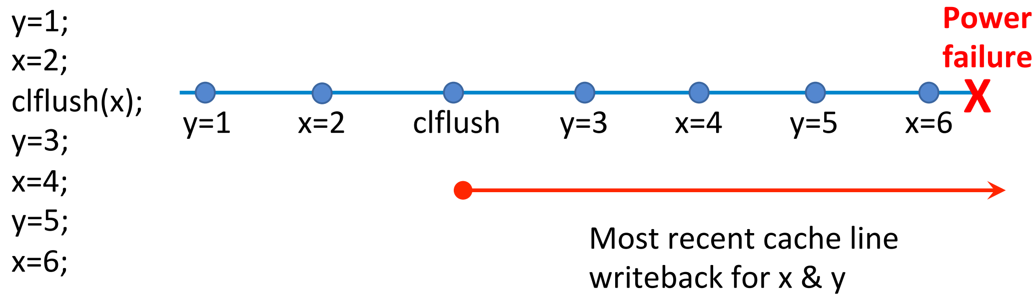


Figure 3.1: Pre-failure execution of a simple PM program. We assume that `x` and `y` reside on the same cache line. The blue line represents the total order in which stores are written to the cache. The red line shows the interval for the last time the cache line containing `x` and `y` may be written back to persistent memory.

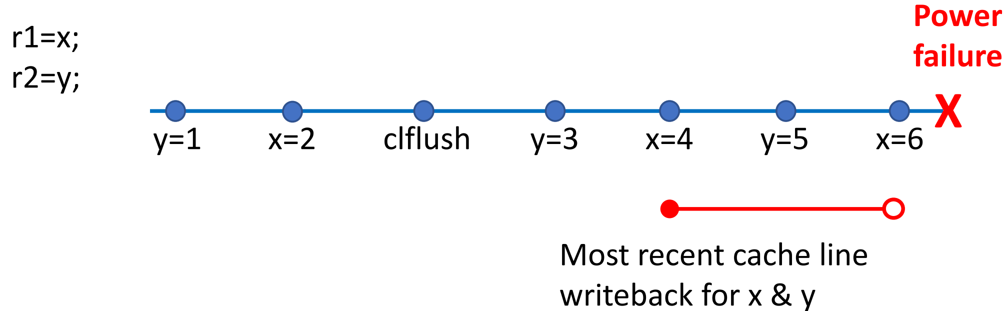


Figure 3.2: Post-failure (recovery) execution of the program that reads the value 4 from `x`. This *refines* the interval for the most recent writeback of the cache line to be between the store $\texttt{x} = \texttt{4}$ and the store $\texttt{x} = \texttt{6}$.

tells us that the cache line must have been flushed some time after the store $\texttt{x} = \texttt{4}$ and before the store $\texttt{x} = \texttt{6}$. Thus, we can refine the interval for the most recent flush to be $[\texttt{x} = \texttt{4}, \texttt{x} = \texttt{6})$, which imposes a much tighter bound.

Since both variables `x` and `y` share the same cache line, reading the value 4 for `x` constrains the set of values that we can read from `y`. In particular, since the last flush occurred some time during the interval from $\texttt{x} = \texttt{4}$ to $\texttt{x} = \texttt{6}$, we know that the cache line was flushed some time after the assignment $y = 3$ and potentially after the assignment $\texttt{y} = \texttt{5}$. Therefore, if the post-failure execution reads from `y`, it could only read the value 3 or 5. It could not read the

value $y = 1$, because the fact that the read from `x` returned 4 tells us that the cache line was flushed after `y = 1` was overwritten.

Jaaru uses this refinement-based approach to simulate cache line flushes and lazily construct the state of persistent memory after the failure, eliminating the need to eagerly explore all (equivalence classes of) states.

3.1.2 Leveraging Commit Stores for Additional Efficiency

Our constraint-refinement approach works well for PM programs because it effectively leverages commit stores to achieve efficiency. Commit stores are a rather common programming practice; in fact, all programs in our evaluation have such commit stores. To effectively leverage such stores, Jaaru does *not* eagerly enumerate all pre-failure stores; instead, Jaaru lazily enumerates a small subset of them that are *actually read* by a post-failure execution.

```
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 3.3: An example program with a commit store.

To illustrate, Figure 3.3 presents a simple program that uses a commit store. There are two methods here ; method `addChild` that adds a child to store data and method `readChild` that returns a pointer to the data stored in the child. We first discuss the `addChild` method. The store at Line 3 writes a reference to the `data` field in the newly created child node. Next, the `clflush` instruction at Line 4 forces this write to persistent memory. Finally, the commit

store at Line 5 makes the child node reachable from the data structure and the `clflush` at Line 6 makes the commit store persistent.

We next discuss the `readChild` method. The load at Line 10 checks whether the `child` field is non-null. If it is, then we know that (1) the `clflush` instruction at Line 6 completed and (2) the `child` node has been persisted and is safe to read in Line 11.

To illustrate how Jaaru leverages this pattern for efficient state exploration, let us consider a client program that executes method `addChild`, fails, and then calls the `readChild` method during recovery. Jaaru injects failures in the execution of method `addChild` at three points: (1) immediately before the `clflush` instruction at Line 4, (2) immediately before the `clflush` instruction at Line 6, and (3) at the end of the execution of method `addChild`. Injecting failures at these three points is sufficient to explore all distinct program behaviors (see § 3.2). While Jaaru supports failure scenarios that involve crashes in the recovery routine, in this example we focus on a single failure for simplicity.

To inject a failure, Jaaru stops the execution at the failure point, resets volatile memory, and starts a new execution with the same persistent memory region. In the new execution, loads from persistent memory check the stores from the pre-failure execution to determine which values the program will read from.

Let us first consider the failure immediately before Line 4. Since the `clflush` instruction has not executed, the write to the `data` field may not have been persisted. When the `readChild` method executes, it first reads the `child` field. Since the `child` field is null, it does not access the `data` field. Jaaru explores exactly one post-failure execution for this failure point.

Next, consider the failure immediately before Line 6. The `data` field has been persisted by the first `clflush` instruction, but the write to the child field has not. Thus, when the post-failure execution reads from the `child` field, Jaaru observes that the interval for the most recent flush of the `child` field is $[0, \infty)$. Jaaru then explores two executions. In the first execution,

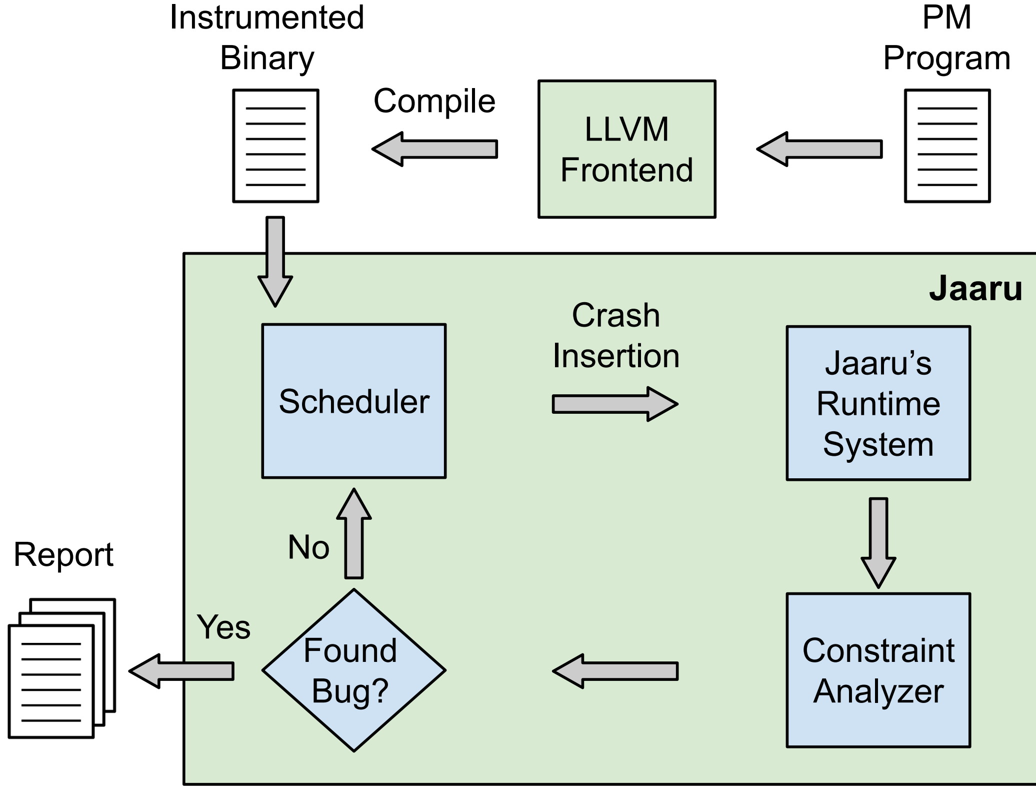


Figure 3.4: Jaaru system overview.

the `child` field is null, and this execution has the same behavior as the previously explored execution. In the second execution, the `child` field is non-null and thus it reads the `data` field. Since the interval $[\texttt{clflush}_4, \infty)$ for the `data` field's cache line starts after the last write to the `data` field, the method returns the `data` field ($\texttt{clflush}_4$ denotes the `clflush` instruction at Line 4.).

Finally, consider the failure at the end of the execution of method `addChild`. At this point, both `clflush` instructions have executed. When the post-failure execution reads from the `child` field, Jaaru observes that the interval for the most recent flush of the child field is $[\texttt{clflush}_6, \infty)$. Therefore the load must see the value written to the `child` field and thus it reads the `data` field. Since the interval $[\texttt{clflush}_6, \infty)$ for the cache line of the `data` field starts after the last write to `data`, the method returns `data`.

To illustrate why such stores are useful, consider the following scenario. Suppose that method `readChild` accesses the `data` field of the child node without first checking the commit store in Line 5. If the `addChild` method crashes before the first `clflush` instruction, there would be two different potential post-failure states for the `data` field. If the child node has n different cache lines that were accessed in a similar manner, then the number of post-failure states would grow to be $O(2^n)$. If the post-failure code accesses all of the child's states, the model checker would have to explore $O(2^n)$ executions. The commit store limits the number of unflushed stores that the post-failure program execution reads from, and thus the executions Jaaru must explore.

The complexity of model checking programs that use commit stores like this example is $O(m^2)$ where m is the length of the execution. We obtain this complexity because the number of failure injection points is $O(m)$, the post-failure execution involves $O(m)$ steps, and with commit stores, we explore two executions at each failure point ; a first execution that reads from the commit store and a second execution that reads the value of the memory location before the commit store.

Note that prior techniques that eagerly explore all pre-failure stores cannot take advantage of such commit stores. The key difference between prior model checkers such as Yat and Jaaru is that Yat enumerates all possible states at the failure point *before* executing the post-failure code (thus with a complexity of $O(2^n)$) while Jaaru executes the post-failure code and lazily explores pre-failure stores that are actually read by the post-failure code.

3.1.3 System Overview

Jaaru uses an LLVM compiler pass to instrument both atomic and normal memory accesses along with fences and cache flush operations. The instrumented binary is then dynamically linked with the Jaaru library. Figure 3.4 presents an overview of Jaaru. A failure scenario

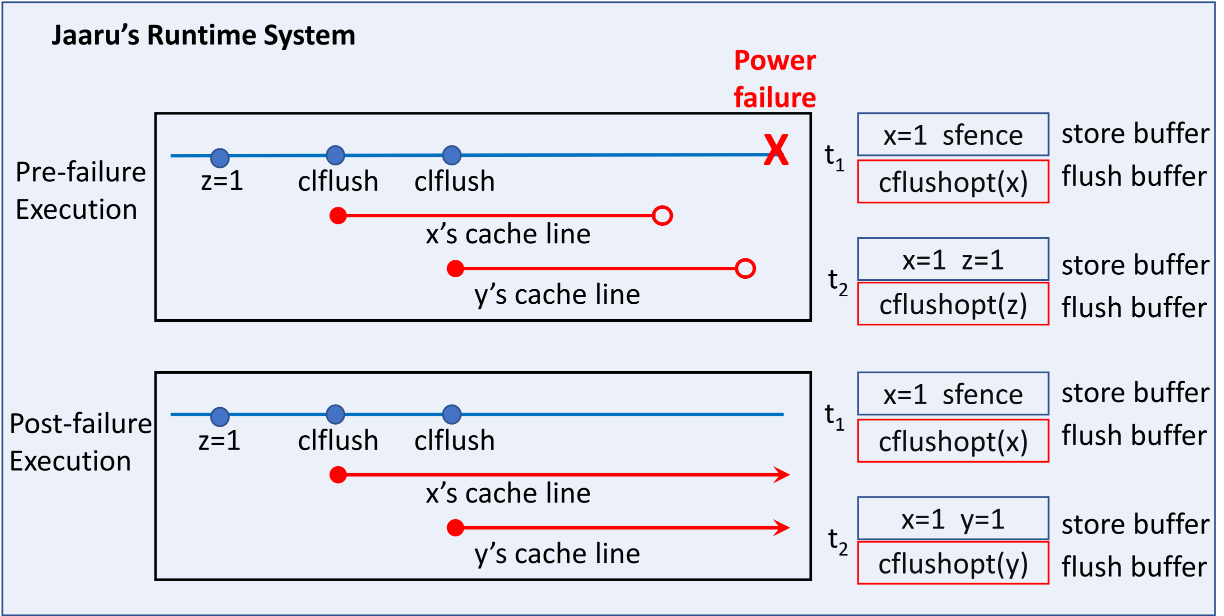


Figure 3.5: An example of Jaaru's runtime system.

involves multiple executions ; the simplest failure scenario (a single failure) involves a pre-failure execution and a post-failure execution. To simulate a failure scenario, Jaaru keeps the information about each of the executions in the sequence that comprises the failure scenario. Figure 3.5 shows the exploration of a failure sequence composed of a pre-failure execution and the current post-failure execution.

Jaaru uses a fork-based approach to roll back executions to simulate failures and start new executions. In each execution, Jaaru records all of the stores that have been written to the cache and the `clflush` instructions that have taken effect (shown with the blue lines). Jaaru also records a set of intervals for every flushed cache line to identify the time ranges of the most recent writes of each cache line into persistent memory (shown in the red lines). As shown earlier in the example, these intervals are used by the model checker to make decisions about the values of variables in the post-failure execution. The right side of each execution shows the thread-specific state Jaaru maintains ; each thread has a local *store buffer* that simulates the processor's store buffer and a *flush buffer* that implements the reordering of

`clflushopt` instructions (based on the constraints in Table 2.1).

3.2 Model Checking Algorithm

This section presents the model checking algorithm. We begin by presenting the following notations that we will use throughout the chapter:

- We refer to an execution as e.
- A given failure scenario may involve a sequence of multiple executions ending in failures. We record this sequence of executions that have been executed on the persistent store using a stack, referred to as *exec*.
- Function `top`(*exec*) denotes the most recent execution (the current one) on the stack *exec*.
- Function `prev`(e) returns the execution that immediately precedes e in *exec*.
- A global sequence number counter σ_{curr} is used to assign increasing sequence numbers to stores, `clflush`, `sfence` instructions.
- Each store, `clflush`, and `sfence` instruction i is assigned a sequence number σ_i. These numbers record the total order in which these instructions take effect in the cache.
- Each execution e has a map e.`getcacheline`() that maps an address to an interval in which the cache line was most recently flushed to persistent memory in the exection e.
- Each execution e has a map e.`queue`() that maps each address *addr* to a sequence of tuples $\langle val, \sigma \rangle$ that record the values stored at the address and the sequence number σ generated at the moment that value was stored.

- We denote a thread using $\tau \in \mathcal{T}$.
- Each thread τ has a store buffer S_τ that keeps a queue of store, `clflush`, and `sfence` operations that have not yet taken effect in the cache.
- Each thread τ has a cache line flush buffer F_τ that stores the set of `clflushopt` operations that have not yet flushed the cache line to persistent storage.
- We refer to the timestamp as t.

The Jaaru LLVM frontend instruments only memory operations and cache operations as those are the operations relevant to persistent storage. Jaaru implements a software simulation of those instructions with full support for the persistency semantics from the Px86_{sim} model [136]. The majority of PM-based tools have been developed for x86 since it provides the most advanced and mature architectural support for accessing persistent memory. By fully supporting x86 semantics, Jaaru satisfies the fast-growing need for a scalable and fast model checker to validate and test these programs. Although the current version of Jaaru is developed for x86, the primary idea behind it is not limited to x86 and could potentially be adapted to support other architectures such as ARM.

The TSO memory model separates the executions of stores, cache flush operations, and `sfence` operations into two phases: (1) the initial phase that often inserts an operation into a buffer and (2) the second phase that removes the instruction from the buffer and updates the state of the cache or persistent storage. We present our algorithm for each of the stages.

Executing instructions. Figure 3.6 presents our algorithm for the first phase of instruction execution, which inserts an instruction into each thread's local store buffer S_τ. The `mfence` instruction waits until S_τ is empty and then clears the thread's flush buffer F_τ.

Updating storage. The second phase occurs when the instruction leaves the store buffer. This phase updates the storage system. Figure 3.7 presents our algorithm for this phase.

```
1: function EXEC_STORE(addr, val, τ)
2:     Enqueue ⟨store, addr, val⟩ into S_τ.
3: function EXEC_CLFLUSH(addr)
4:     Enqueue ⟨clflush, addr⟩ into S_τ.
5: function EXEC_CLFLUSHOPT(addr)
6:     Enqueue ⟨clflushopt, addr, σ_curr⟩ into S_τ.
7: function EXEC_SFENCE
8:     Enqueue ⟨sfence⟩ into S_τ.
9: function EXEC_MFENCE
10:    Evict all entries in S_τ.
11:    Flush F_τ.
```

Figure 3.6: Algorithm for executing instructions.

```
1: function EVICT_SB(⟨store, addr, val⟩)
2:     σ_curr := σ_curr + 1
3:     Enqueue ⟨val, σ_curr⟩ into top(exec).queue(addr).
4:     t_{τ,CacheID(addr)} = σ_curr
5: function EVICT_SB(⟨clflush, addr⟩)
6:     σ_curr := σ_curr + 1
7:     cl := top(exec).getcacheline(addr)
8:     cl.begin = σ_curr
9:     t_{τ,CacheID(addr)} = σ_curr
10: function EVICT_SB(⟨clflushopt, addr, σ⟩)
11:    Add ⟨addr, max(σ, t_{τ,CacheID(addr)}, t_τ)⟩ to F_τ.
12: function EVICT_SB(⟨sfence⟩)
13:    σ_curr := σ_curr + 1
14:    Flush F_τ.
15:    t_τ = σ_curr
16: function EVICT_FB(⟨addr, σ⟩)
17:    cl := top(exec).getcacheline(addr)
18:    cl.begin = max(cl.begin, σ)
```

Figure 3.7: Algorithm for evicting store and flush buffers.

We have four different implementations of the EVICT_SB function for different types of instructions.

The EVICT_SB($\langle$store, $addr$, $val\rangle$) function handles store instructions. This function assigns a sequence number to each store. These sequence numbers enforce a total order over all writes to the cache. The function then moves the store to the queue of stores that records possible cache line values based upon the address it writes to. Finally, the function updates the timestamp ($t_{\tau,\texttt{CacheID}(addr)}$) for the most recent write to the cache line or `clflush` from this thread to be the store's sequence number.

The EVICT_SB($\langle$`clflush`$, addr\rangle$) function handles the cache line flush instruction `clflush`. The function first assigns a unique sequence number to the instruction. It then updates the lower bound of when the cache line was most recently flushed to be the sequence number for this particular flush operation. Finally, the function updates the timestamp for the most recent write to the cache line or `clflush` from this thread to be the store's sequence number.

The EVICT_SB($\langle$`clflushopt`$, addr, \sigma\rangle$) function handles the optimized cache line flush instruction `clflushopt`. The `clflushopt` instruction can be reordered with other `clflushopt` instructions, previous stores to other cache lines, `clflush` instructions to different cache lines, and later stores to any cache line. Support for reordering with previous operations is implemented by computing the *maximum sequence number* of the most recent instruction that the `clflushopt` cannot be reordered with. Support for reordering with later instructions is implemented by a flush buffer that is emptied when an instruction, which cannot be reordered with previous `clflushopt` instructions (*i.e.*,`sfence`, `mfence`, or `RMW` instructions), executes.

The EVICT_SB($\langle$sfence$\rangle$) function handles the store fence instruction `sfence`. This `sfence` instruction is ordered relative to all previous `clflushopt` instructions and thus it flushes the thread's flush buffer when it exits the thread's store buffer.

Finally, the EVICT_FB($\langle addr, \sigma\rangle$) function handles `clflushopt` instructions when they are evicted from the flush buffer by an `sfence`, `mfence`, or `RMW` instruction. This function updates the lower bound of when the cache line was most recently flushed to be the sequence number σ from the tuple in the flush buffer. Recall that this sequence number is the maximum of the following four values: (1) the current sequence number when the `clflushopt` instruction was first executed, (2) the sequence number of the most recent `sfence` instruction executed by the thread, (3) the sequence number of the most recent store to the same cache line executed by the same thread, or (4) the sequence number of the most recent `clflush` to the same cache line executed by the same thread.

```
1: function BuildMayReadFrom(addr)
2:     if ∃val.S_τ = b_1.⟨addr, val⟩.b_2 ∧ ∀val'.⟨addr, val'⟩ ∉ b_2 then
3:         return {⟨top(exec), _, val⟩}
4:     if ∃val, σ.top(exec).queue(addr) = m_1.⟨val, σ⟩ then
5:         return {⟨top(exec), _, val⟩}
6:     return ReadPreFailure(prev(top(exec)), addr)
7: function ReadPreFailure(e, addr)
8:     cl := e.getcacheline(addr)
9:     set := {⟨e, σ, val⟩ | σ < cl.end ∧ e.queue(addr) = m_1.⟨val.σ⟩.m_2 ∧ (σ ≤ cl.begin ⇒ ∀val'.∀σ' ≤
   cl.begin.⟨val', σ'⟩ ∉ m_2)}
10:    if ∃⟨val, σ⟩ ∈ set.σ ≤ cl.begin then
11:        return set
12:    else
13:        return set ∪ ReadPreFailure(prev(e), addr)
```

Figure 3.8: Algorithm for BuildMayReadFrom.

Load operations. Figures 3.8 and 3.9 present our algorithm for loads. We split handling of loads into two functions: (1) the BuildMayReadFrom function that computes and returns a set of stores that a load may read from and (2) the DoRead function that refines the cache line flush intervals once Jaaru has selected a specific store for the load to read from. Splitting the load handling into two components makes it straightforward to integrate loads into Jaaru's exploration.

We first discuss the BuildMayReadFrom function in Figure 3.8. This function returns a set of tuples for each possible store that the load may read from. Each tuple contains the execution e that performed the store, the sequence number σ of the store, and the value val stored. We use _ when the store is from the current execution and thus does not have a sequence number that can be used to constrain when a cache line was last flushed in the previous execution.

Lines 2–3 check whether there is a store to read from in the store buffer, and if so, returns the tuple for the newest such store. More precisely, the syntax $b_1.\langle addr, val\rangle.b_2$ represents the state of store buffer with b_1 being the oldest operations and b_2 being the newest operations. A load can read from a store $\langle addr, val\rangle$ in the store buffer if there are no newer stores to the same address.

```
1: function DoRead(addr, ⟨e, σ, val⟩)
2:     if e ≠ top(exec) then
3:         UpdateRanges(prev(top(exec)), addr, ⟨e, σ, val⟩)
4: function UpdateRanges(e_c, addr, ⟨e, σ, val⟩)
5:     if e ≠ e_c then
6:         cl := e_c.getcacheline(addr)
7:         ⟨val′, σ′⟩ := first(e_c.queue(addr))
8:         cl.end = min(cl.end, σ′)
9:         UpdateRanges(prev(e_c), addr, ⟨e, σ, val⟩)
10:    else
11:        cl := e_c.getcacheline(addr)
12:        cl.begin := max(cl.begin, σ)
13:        Let σ′ be the sequence number for the next tuple after ⟨val, σ⟩ in e_c.queue(addr) or ∞ if there is
   no such tuple.
14:        cl.end := min(cl.end, σ′)
```

Figure 3.9: Algorithm for DoRead.

Lines 4–5 check whether there is a store in the current execution that has updated the cache. If so, they return the tuple for that store. More precisely, the syntax $m_1.\langle val, \sigma\rangle$ represents the sequence of stores written to the cache with m_1 being the older operations. A load can read from a store $\langle val, \sigma\rangle$ in the cache queue for an address if there are no newer stores to the same address. Otherwise, Line 6 invokes the ReadPreFailure function to compute potential stores from the executions before the most recent failure.

We next discuss the ReadPreFailure function in Figure 3.8. This function computes the set of stores from previous executions that a load may read from. Lines 8–9 compute the set of stores that would have been present on the cache line for the time range specified by the cache line's last flush interval. Line 10 checks whether there was a store performed before the earliest possible time for the cache line flush. If there is no such store, it is possible that the load has read from an earlier execution. In this case, the algorithm recursively calls ReadPreFailure on earlier executions and combines the set of stores from the current execution with those returned by the recursive call.

After the model checking algorithm has selected a store for the load to read from, it invokes the DoRead function in Figure 3.9 to refine the most recent cache line flush intervals for previous executions. Line 2 checks whether the store is from the current execution. If so,

there is no refinement to be performed and the function returns. Otherwise, it calls the function UPDATERANGES to refine the interval in which the last cache flush was performed.

We next discuss the UPDATERANGE function. Line 5 checks whether the store is from the execution e_c. If not, Lines 6–9 refine the upper bound of the most recent flush interval to occur before the first store because the load reads from a store from a prior execution e and thus we know that the current execution e_C did not flush the cache line after performing a store. It then recursively calls the UPDATERANGE function on previous executions.

If the store is from the execution e_c, then Lines 11–14 refine the interval for the most recent cache line flush. The key insight is that the cache line must have been flushed after the store that the load reads from and before any subsequent stores.

Exploration algorithm. Finally, we present the core model checking algorithm. Figure 3.10 presents the EXPLORE function that implements Jaaru's exploration. The EXPLORE function takes in an execution s and a stack of executions *exec*. Lines 2–3 inject failures and start new executions. Lines 4–8 decide whether to evict an entry from a thread's store buffer. Function `next`(τ, h) calls the appropriate EVICT function from Figure 3.7. Line 9 selects the next thread to execute. Line 10 checks whether the thread's next operation is a load. If so, Lines 11–14 handle the load ; we first call BUILDMAYREADFROM to compute the set of potential stores that the load may read from. The `foreach` loop then explores executions for each possible store that the load may read from.

If the thread's next operation is *not* a load, then Line 16 explores the next step. Function `next`(τ) computes the next step by calling the appropriate EXEC function from Figure 3.6.

Injecting failures. The natural points to inject failures are those immediately before operations that flush cache lines. The reason is that writes to the cache *increase* the set of possible post-failure executions while flushes *decrease* the set of possible post-failure executions. Thus, injecting failures at these points is sufficient to explore all program behaviors. Jaaru,

```
1: function Explore(s, exec)
2:     if choose to fail then
3:         Explore(s_0, exec.push(fresh_execution()))
4:     if choose to evict then
5:         Select τ from nonemptystorebuffer(s)
6:         Pop head h off of S_τ
7:         Explore(s.next(τ, h), exec)
8:     else
9:         Select τ from enabled(s)
10:        if next action a for thread τ is a load then
11:            rfset := BuildMayReadFrom(a.addr)
12:            for each ⟨e, σ, val⟩ ∈ rfset do
13:                Explore(s.DoRead(⟨e, σ, val⟩), exec)
14:            end for
15:        else
16:            Explore(s.next(τ), exec)
```

Figure 3.10: The main model checking Algorithm.

therefore, injects failures at those points.

For long runs or scenarios in which multiple failures are injected, injecting failures before every flush can result in exploring many executions. Jaaru contains an optimization that skips injecting a failure if there have been no writes since the last injected failure. Jaaru can also support injecting failures into a post-failure execution (with a command line option). This option controls the maximum depth of the *exec* stack.

Locked `RMW` instructions. Locked (atomic) read-modify-write instructions include compare-and-swap (CAS), atomic exchange, and many atomic arithmetic instructions. On x86 these instructions also have fence-like semantics. They are equivalent to the atomic execution of the following sequence of instructions: `mfence`, load, store, and `mfence`. Jaaru implements them by atomically executing this sequence.

Mixed size accesses. C and C++ programs may access fields using stores and loads with different widths. For example, a 32-bit integer field in a union may be initialized to 0 with a 64-bit integer store and then read with a 32-bit integer load. We implement accesses that are larger than a byte as a sequence of byte accesses that are performed atomically. Thus, a 32-bit load is implemented as four 8-bit loads.

Checksum-based recovery. One approach to ensure data persistency is to write a checksum along with data ; recovery code reads the checksum to verify that the data was persisted. Checksum-based recovery differs from other approaches in that the recovery code may read from a larger number of non-persisted stores. Jaaru provides special support that can exhaustively check programs that use checksum-based recovery without explicit flushes.

Debugging support. Jaaru can identify different types of bugs including missing fences, misordered flushes, missing flushes, and misordered stores that cause atomicity violations. The most common bug type we found was due to missing cache line flush instructions. Looking at the entire trace to understand a bug is not easy. Therefore, we extend Jaaru with additional functionality to help developers quickly determine why a program crashed.

Our observation is that a missing flush instruction effectively increases the number of pre-failure stores that a post-failure load may read from. Jaaru, therefore, contains optional support for flagging loads that can read from more than one store. To facilitate debugging, Jaaru prints out the load that can read from multiple stores, the source location of the load, each of the stores, their locations in the trace, and their source locations. Our experience shows that this information is very useful for quickly understanding missing flush instructions that cause the program to crash or loop.

3.2.1 Discussion

Existing DPOR algorithms [21, 46, 96, 140, 143, 144, 147] are not directly applicable in the setting of persistent memory. None of the traditional DPOR algorithms consider the effect of volatile memory such as crashes and cache flushes. For example, a crash makes pre-failure stores that were executed but not written to persistent memory completely disappear.

Jaaru can be viewed as implementing a form of dynamic partial order reduction that avoids

exploring equivalent executions. Pre-failure executions that differ in when cache lines are flushed and thus generate different post-failure states can still yield the same post-failure executions if the post-failure executions never read from the memory locations that contain different values. Such cache line flushes can be viewed as commuting with the crash. Other cache line flushes make stores visible to post-failure loads and thus do not commute with the crash. Jaaru's constraint refinement algorithm lazily identifies non-commuting cache line operations during the post-failure execution and effectively explores reordering such cache line flushes.

Many PM programs are multi-threaded, creating the opportunity for concurrency bugs. Jaaru does not exhaustively explore all concurrent schedules and thus does not provide any guarantees that it will find concurrency bugs. However, since Jaaru controls the concurrent schedule and fully simulates the TSO memory model, as future work, it can be used to fuzz for concurrency bugs.

3.3 Evaluation

In this section, we evaluate Jaaru's bug-finding capabilities and performance with a set of benchmarks. Our system configuration is reported in Table 3.1.

Table 3.1: System configuration.

CPU	6-core 3.7 GHz Intel i7-8700K processor
Volatile Memory	32GB DDR4, 2666MT/s
Non-volatile Memory	Full $Px86_{sim}$ semantics simulated (see §3.2)
OS	Ubuntu Linux 18.04
Compiler	gcc version 7.5.0 opt level O3 clang version 11.0.0 opt level O3

Our benchmarks. We have evaluated Jaaru on PMDK [33] and RECIPE [98]. PMDK is a library used extensively in prior work to evaluate bug-finding techniques [107, 108]. Both PMTest and XFDetector would require extra annotations to cover different behaviors of PMDK (*e.g.*, PMTest requires the persistency order of every single variable to be defined by annotations). These annotations are on top of the normal assertions used to sanity check the program. However, Jaaru, as a model checker, can exhaustively explore the state space without the need to write any extra assertions other than the basic sanity checks that programs often have. For RECIPE, we were not able to run the P-HOT program because it did not compile with LLVM. All programs in the PMDK library have been used.

Memcached and Redis have both been ported to use PMDK and evaluated in prior work [107, 108]. Unfortunately, Memcached and Redis can only be executed as servers that interact with clients via sockets. Model checking a program that interacts with other programs requires support for *deterministically replaying those socket interactions* that the current version of Jaaru does not support. Jaaru could potentially be integrated with existing record-and-replay debugging frameworks to lift this limitation.

3.3.1 Bug Detection

We ran Jaaru over PMDK and RECIPE automatically to find bugs. The inputs are examples that come with these benchmark suites. We have not developed any new inputs ourselves. Jaaru has found a total of 25 bugs, of which 18 are new bugs that have not been reported before. Bugs that Jaaru can identify must have some visible manifestation ; either a crash, *e.g.*, segmentation fault, or an assertion failure in the program. Missing sanity checks in the program can result in silent data corruption where the program appears to recover successfully but has incorrect data.

We first discuss our experience with PMDK. Figure 3.11 reports the bugs we have found.

For each bug, we list the program in which the bug was found. *Note that the majority of these bugs are in the core libpmemobj library in PMDK and the examples merely have served as test cases for the library.* For each bug, Figure 3.11 reports the symptoms of the bug, *e.g.*, an assertion failure or illegal memory access. For many of these bugs, we have found that multiple failure injection points have led to the same symptom. These bugs may or may not be the same and to be conservative we report each such group of bugs as one bug. We have found 6 new bugs in the PMDK library ; only bug #2 was previously found by XFDetector [107]. Some of these bugs are not missing-flush bugs since the stores are followed by appropriate flush instructions, but atomicity violations in which partially completing updates leaves the data structures in inconsistent states. None of these 6 bugs was reported before (in either PMTest [108] or XFDetector [107]).

#	Benchmark	Symptom
1	Btree*	Illegal memory access at btree_map.c:89
2	Btree	Failed to open pool error
3	Hashmap_atomic*	Assertion failure at heap.c:533
4	CTree*	Assertion failure at obj.c:1523
5	Hashmap_atomic*	Assertion failure at pmalloc.c:270
6	Hashmap_tx*	Illegal memory access at obj.c:1528
7	RBTree*	Illegal memory access at rbtree_map.c:137

Figure 3.11: Bugs found in PMDK. *Bugs with a * are new bugs.* Only the second bug was reported before in XFDetector [107].

We next discuss results for the RECIPE benchmarks. We have found 12 new bugs in the RECIPE programs. Many programs contain multiple bugs. When Jaaru has found an execution that causes the program to crash (or loop) we have examined Jaaru's outputted trace and debugging information to understand the bug. Since these benchmarks are easier to understand than PMDK benchmarks, we have fixed the bug and used Jaaru to look for additional bugs. We continued this until the program executed correctly.

Figure 3.12 presents the bugs we have found. We confirmed that each bug caused the program to crash. Jaaru found bugs in *every program*. These bugs are primarily missing flush instructions in object constructors. All of the bugs can potentially corrupt a persistent

#	Benchmark	Type of Bug
1	CCEH*	Missing flush in CCEH constructor
2	CCEH*	Missing flush in CCEH constructor
3	CCEH*	Missing flush in CCEH constructor
4	FAST_FAIR	Missing flush in header constructor
5	FAST_FAIR	Missing flush in entry constructor
6	FAST_FAIR*	Missing flush in btree constructor
7	P-ART*	Use of non-persistent data structure in Epoch
8	P-ART*	Missing flush in Tree constructor
9	P-ART*	Use of non-persistent data structure for recovery
10	P-BwTree*	GC crash leaves data structure in inconsistent state
11	P-BwTree*	Missing flush of GC metadata pointer
12	P-BwTree*	Missing flush of GC metadata
13	P-BwTree*	Missing flush in AllocationMeta constructor
14	P-BwTree*	Missing flush in BwTree constructor
15	P-CLHT	Missing flush in clht constructor
16	P-CLHT	Missing flush for hashtable object
17	P-CLHT	Missing flush for hashtable array
18	P-MassTree	Flushed referenced object instead of pointer

Figure 3.12: Bugs were found by Jaaru in every program of RECIPE. *Bugs with a * are new bugs.*

data structure leading to data loss.

Many bugs are simple cases of forgetting to flush stores or mistakenly flushing the wrong memory location. However, we have found other kinds of bugs. In P-ART, the developer has used a vector data structure from tbb to track locks that must be unlocked in the recovery procedure. The bug is that tbb data structures do not persist across failures. In P-BwTree, Jaaru has found a logical error in the garbage collection (GC) algorithm in which failures during the GC can corrupt the GC data structures. This bug is an atomicity violation and not a case of missing flushes.

Comparing these results with the bugs found by PMTest [108] and XFDetector [107], Jaaru appears to have a stronger bug-finding ability than PMTest and XFDetector. For example, PMTest reported three new bugs and XFDetector reported four; several of these bugs were performance bugs. On the contrary, Jaaru found serious functional bugs that can corrupt data structures and lead to a crash or an assertion failure in the program. This is not surprising because Jaaru explores many more states than PMTest and XFDetector, which focus on a

single execution.

Among the several bugs reported before, three were not found by Jaaru. We inspected those bugs and found it was because (1) two were performance bugs that are *not* our focus and (2) one was in the Redis code which we did not test. Jaaru could be extended to find performance bugs such as redundant cache flushes and fences.

3.3.2 Jaaru Bug Reporting

We presented Jaaru and the bugs found by our tool to the authors of RECIPE and we received overall positive feedback. At the time of writing, 6 out of 18 bugs found by Jaaru were fixed by the developers of RECIPE. There were 6 bugs that were related to memory allocators and garbage collectors. The RECIPE developers did not fix the persistency bugs related to memory allocators because they believe these bugs need to be addressed by the memory allocators, which is not their focus. The remainder of the bugs were already fixed before our bug report.

3.3.3 Performance

Figure 3.13 presents the performance results for Jaaru on RECIPE benchmarks. Providing performance results for a model checker requires first fixing the bugs we have found so that Jaaru can run to completion and fully explore the state space of the program; otherwise, it would not make sense to report running time. We have spent much time fixing all the bugs we have found in RECIPE so that the model checker can fully explore these benchmarks. The bugs in the PMDK framework are more complicated and would take more time to fix, so we did not include our performance results for PMDK. Note that Jaaru is able to model check each RECIPE program in less than 15 seconds. We next discuss our evaluation of the state

Benchmark	#JExec.	JTime	#FPoints	#Yat Execs.
CCEH	891	14.51s	528	2.17×10^{182}
FAST_FAIR	170	1.48s	41	5.43×10^{15}
P-ART	174	1.86s	22	1.21×10^{34}
P-BwTree	71	0.79s	36	1.50×10^{16}
P-CLHT	25	1.59s	12	1.93×10^{605}
P-Masstree	24	0.17s	16	1.67×10^{15}

Figure 3.13: Jaaru's state space reduction. Reported are the number of times Jaaru executes a program (**JExec.**), time Jaaru takes to finish exploration (**JTime**), number of failure injection points (**FPoints**), and the number of program executions Yat needs to eagerly explore pre-failure stores (**Yat Execs.**).

space reduction that Jaaru achieves on these programs, relative to an eager model checking approach such as that implemented in Yat [95]. Since Yat is not publicly available, we have calculated the number of legal post-failure states that Yat would have to explore. Figure 3.13 presents these results. Given the very large number of executions Yat would have to explore, it is unlikely to be feasible to exhaustively model check these realistic programs with Yat.

To better understand Jaaru's effectiveness, we compare the total number of executions with the number of failure injection points in the original execution. As shown in Figure 3.13, Jaaru only explores a few executions per failure injection point. The number of executions per failure injection point ranges from 1.5 to slightly less than 8.

It does not make much sense to compare performance directly between Jaaru and non-exhaustive approaches such as PMTest and XFDetector, which detect bugs on single executions. However, as a reference, Jaaru incurs an overall slowdown of 736× per execution, which is on par with the overhead of XFDetector (*i.e.*, from dozens of times to almost a thousand times as reported in the paper [107]). PMTest and Pmemcheck have much lower overhead (1.69× and 22.3×, respectively). This is because Jaaru fully simulates the x86 TSO persistency semantics while the other tools ignore the effects of store buffers.

3.3.4 Key Takeaway

Our results highlight the strengths and weaknesses of model checking: Jaaru finds more bugs without any user involvement, but cannot easily handle programs with complex interactions with the outside world. Jaaru is a good fit for checking library code that is usually small in size but has a large impact. Non-exhaustive tools such as PMTest and XFDetector should be used to check large programs such as Redis whose non-determinism from the network can give a model checker much trouble. It is also clear that the constraint refinement approach enables Jaaru to efficiently check these programs; without refinement, it would not be possible for a model checker to scale even to library code.

Chapter 4

Robustness

Recal from Chapter 1, majority of testing frameworks detect persistency bugs that have visible symptoms such as assertion failure or segmentation fault. The issue is that not all bugs can have visible manifestation such as silent data corruption bugs. This chapter introduces a notion of robustness and elaborates on PSan's implementation. To be more specific, 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.

4.1 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 [132, 131, 41, 80, 60, 47]. Among them, *strict persistency* is the most conservative and intuitive model which integrates memory persistency into memory consistency [131]. 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 describes the strict persistency model and robustness condition. The formal definition of them can be found in the supplemental material of PSAN paper [61]. First, we will introduce some notations. 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. Each execution *Exec* includes n crash events and $n+1$ sub-execution. 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 $st\langle x\rangle$.

4.1.1 Strict Persistency

We describe 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 $st\langle y\rangle$.

4.1.2 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 1.3, 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 with n crash events.

Definition 4.1.1. An execution with n crash events under weak persistency model is robust if:

1. the last sub-execution of the program under weak persistency model is identical to the last sub-execution of program under strict persistency model ;

2. in execution under weak persistency model, if $st\langle x\rangle$ in any sub-execution is read from by a load in later sub-executions, then $st\langle x\rangle$ must exist in the execution under strict persistency model .

3. the execution under strict persistency model preserves the sequenced-before and reads-from relations, the happens-before relation over stores, and the TSO order in the execution under weak persistency model. .

The *happens-before relation over stores* is defined in Section 4.1.4. Note that the execution under weak persistency model and execution under weak persistency model may not in general have the same crash point. This means threads in each sub-execution under weak persistency model can have different crash point compared to threads in each each sub-execution under strict persistency model.

Definition 4.1.1 states that at any point of the execution under weak persistency model, the most recent sub-execution has the same behavior as that of some strictly persistent execution. For store operations that are not included by any of sub-executions under strict persistency, their effects are either not written to the persistent memory or not read from by loads in later sub-executions. However, if the stores in any sub-execution is read from by loads in later sub-executions, then it must be included in sub-execution under strict persistency.

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

Definition 4.1.2. A program P is robust to a weak persistency model if every execution *Exec* of program P is robust.

4.1.3 Persistent Lock-Free Data Structures

As prior studies note [149, 9, 78, 34], *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.

4.1.4 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 [45] 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.

States:

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

[LOAD]

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

[STORE ISSUE]

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

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

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

4.2 Basic Ideas

PSAN builds on the open-source Jaaru infrastructure [62] for simulating the x86 persistent memory model. Jaaru's frontend takes as input the PM program source and generates an instrumented binary. The instrumented binary is executed by Jaaru, and Jaaru generates an execution trace. Jaaru assumes as input a set of test cases that explore a program's PM data structures. These can potentially be generated by existing test data generation tools [106, 5, 16, 17, 12, 8, 100, 148, 145, 57, 58, 135]. Jaaru generates executions of PM programs, and then PSAN checks these executions for robustness violations using Jaaru's plugin interface.

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

4.2.1 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 4.1.4 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 4.2: An example of non-robust program with missing flush and drain operations. `x` and `y` are initialized to 0.

Figure 4.2 presents an example that we will use to present our basic approach. The left column in Figure 4.2 shows the code of the pre-crash execution and the right column shows the code of the post-crash execution. Section 4.4.1 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 4.2-a.

Consider an execution in which `r1 = 2` and `r2 = 5`. Figure 4.3 shows such an example and

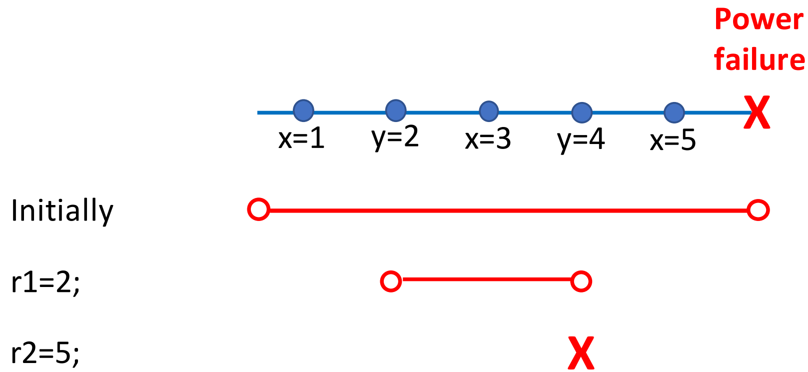


Figure 4.3: Constraints for execution of code in Figure 4.2 where r1 = 2 and r2 = 5.

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`.

4.2.2 Supporting Threads

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

```
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 4.4: `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 $\texttt{x} = \texttt{1}$ is TSO ordered before $\texttt{y} = \texttt{1}$. Can the execution read $\texttt{r1} = \texttt{0}$ and $\texttt{r2} = \texttt{1}$?

```
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 4.5: 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 $\texttt{r1} = \texttt{1}$, $\texttt{r2} = \texttt{0}$, and $\texttt{r3} = \texttt{1}$?

Per-Thread Crash Intervals

Naïvely applying potential crash interval constraints to a multi-threaded execution trace using TSO order is overly restrictive. Figure 4.4 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 4.2.1 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 4.5 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 4.5-(a) and 4.5-(b) present the pre-crash execution code for thread τ_1 and thread τ_2. Figure 4.5-(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 $\mathbb{C}(\tau_1) = [0, \texttt{getcl}(\texttt{x = 1}))$. 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 $\mathbb{C}(\tau_2) = [\texttt{getcl}(\texttt{y = r1}), \infty)$. 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 4.4 is robust, but `x = 1` is not persistent even though it is TSO ordered before `y = 1`, and `y = 1` is made persistent.

4.2.3 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:

$$st\langle x,\tau_1\rangle \xrightarrow{rf} ld\langle x\rangle \Rightarrow \mathbb{C}(\tau_1) := [\texttt{getcl}(st\langle x,\tau_1\rangle),\infty) \wedge \mathbb{C}(\tau_1). \quad (4.2.1)$$

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:

$$st\langle x,\tau_1\rangle \xrightarrow{rf} ld\langle x\rangle \wedge st\langle x,\tau_1\rangle \xrightarrow{tso} st\langle x,\tau_2\rangle \Rightarrow \mathbb{C}(\tau_2) := [0,\texttt{getcl}(st\langle x,\tau_2\rangle)) \wedge \mathbb{C}(\tau_2). \quad (4.2.2)$$

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

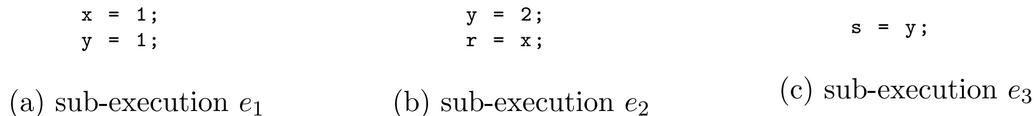
been executed before $st\langle x, \tau_1\rangle$:

$$st\langle x, \tau_1\rangle \xrightarrow{rf} ld\langle x\rangle \wedge st\langle y, \tau_3\rangle \xrightarrow{hb} st\langle x, \tau_1\rangle$$
$$\Rightarrow \mathbb{C}(\tau_3) := [\texttt{getcl}(st\langle y, \tau_3\rangle), \infty) \wedge \mathbb{C}(\tau_3). \quad (4.2.3)$$

4.2.4 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 4.3.1.



(a) sub-execution e_1 (b) sub-execution e_2 (c) sub-execution e_3

Figure 4.6: 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 4.6 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 $\mathcal{C}(e_1) = [0, \texttt{getcl}(\texttt{x = 1}))$, 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 $[0, \texttt{getcl}(\texttt{x = 1})) \wedge [\texttt{getcl}(\texttt{y = 1}), \infty)$, because the first sub-execution of the equivalent execution must crash after `y = 1` commits to the cache. Also, $\mathcal{C}(e_2)$ becomes $[0, \texttt{getcl}(\texttt{y = 2}))$, because the second sub-execution of the equivalent execution must crash before `y = 2` commits to the cache. Otherwise, the older store `y = 1` would be overwritten. However, the constraints in $\mathcal{C}(e_1)$ are not satisfiable, and such equivalent execution does not exist.

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

4.3 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 4.7: A simple concurrent programming language.

We present our algorithm for detecting robustness violations with respect to the simple concurrent language used by Px86_{sim} [136], as described in Figure 4.7. We assume that `Reg` is a finite set of registers (local variables), `Val` is a finite set of values, and $\texttt{TId} \subseteq \mathbb{N}$ 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.

4.3.1 Operational Semantics

Figure 4.8 presents our algorithm in operational semantics as an extension to the Px86_{sim} operational model. Before performing the analysis in Figure 4.8, the algorithm in Figure 4.1

for computing clock vectors and sequence numbers is applied to the corresponding operations. After the analysis in Figure 4.8, we extend the transitions for the Px86_{sim} operational model [136].

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]

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

Figure 4.8: 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 4.2.1 and 4.2.3. Then the last line in Figure 4.8 iteratively applies the implication 4.2.2 for each store in the set $\texttt{next}(st\langle x, \tau \rangle, e_c)$.

Generally speaking, we can prove the correctness of our algorithm by contradiction. Suppose the algorithm reports a robustness violation for a program that is robust. This means that a constraint inferred by later load conflicts with the existing constraint. These constraints are either of the form $[0, \alpha)$ or of the form $[\beta, \infty)$ but they cannot have the same form. If the new constraint is in the form of $[0, \alpha)$, there is a later store to the same location, x, with the clock of α that enforces this constraint that can be in the same executions or later executions. In this case, the existing constraint is in the form of $[\beta, \infty)$. This means that prior loads read from a store to a different variable, y, that happens after the later store to x. The store to y has to happen after the later store to x since $\beta > \alpha$. Due to the condition 2 in Definition 4.1.1, it can be inferred the later store to x has to be in the consistent prefix under the strict persistency model which means the cache line is flushed after the later store to x. Since we assumed the program is robust the load from x needed to read from the later store to x which contradicts our initial assumption where the load reads from the first store to x. Similar reasoning for the case where the new constraint is in the form of $[\beta, \infty)$. *We present a correctness proof for the algorithm in the supplemental material of our* PSan *paper [61].* Refer to this source for more detailed technical information on the proof of the algorithm.

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

Reading from Too Old of Store. Figure 4.9 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

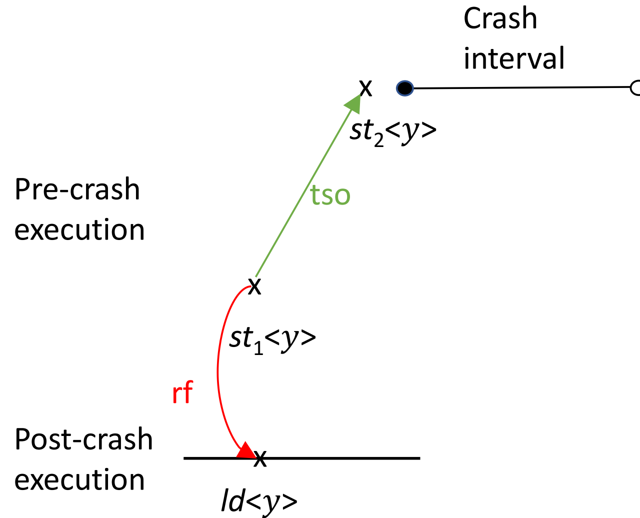


Figure 4.9: Reading from a store that is too old.

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 4.5 in which a thread crashes between performing a store and flushing and draining the store, but a second thread observes the presence of that store and then persist stores of their own. In this case, the primary fix interval would be empty, and PSan would produce an alternate interval for that second thread.

Alternatively, to fix this bug by colocating fields on the same cache line, PSan would compute the store that sets the beginning of the crash interval shown in Figure 4.9. 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 4.10 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

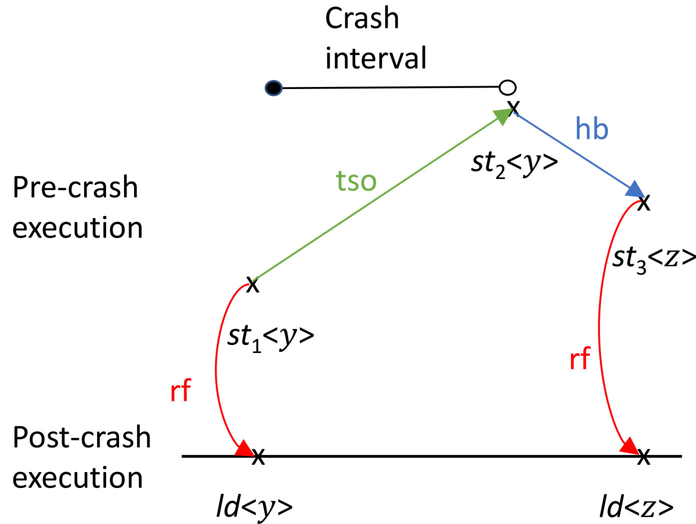


Figure 4.10: Reading from a store that is too new.

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.

4.4 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 [62]. 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.

4.4.1 Methodology

We first tested PSan on the RECIPE [98] collection of PM indexes based on B+-trees, tries, radix trees, and hash tables [123, 72, 98]. CCEH [123] is an efficient hash table for persistent

memory. FAST_FAIR [72] 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 [33], Memcached [35], and Redis [93]. 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.

4.4.2 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 4.1 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 4.1 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 4.1, in FAST_FAIR benchmark. In this benchmark, the `header` class (Shown in Figure 4.11) is used at the beginning of the `page` class [72]. 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.

```
1   class header{
2       page* leftmost_ptr;          // 8 bytes
3       page* sibling_ptr;           // 8 bytes
4       uint32_t level;              // 4 bytes
5       uint32_t switch_counter;     // 4 bytes
6       std::mutex *mtx;             // 8 bytes
7       union Key highest;           // 8 bytes
8       uint8_t is_deleted;          // 2 bytes
9       int16_t last_index;          // 2 bytes
10      uint8_t dummy[5];            // 5 bytes
11  }
```

Figure 4.11: Source code for bug #9 which leads to unaligned accesses by the program.

Memory Management Bugs. In addition to robustness violations in Table 4.1, PSan found 9 more robustness violations in P-ART and 4 more in P-BwTree. PSan found these violations in memory management code such as garbage collection and the memory allocation implementation. As mentioned in the paper [98], 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.

4.4.3 Performance

We next ran 100 random executions with both PSan and Jaaru, the underlying model checker, to report the overhead of PSan. Table 4.2 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.

4.4.4 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* [36], *pointer tagging* [101], 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 4.1, 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 [62] and Witcher [49], 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.

Table 4.1: 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*

Table 4.2: 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

Chapter 5

Persistency Race

Chapter 3 and Chapter 4 discusses tools and techniques to detect persistent memory bugs that are originated by developers' error in inserting flush and fence instructions. This chapter discusses a new class of persistency bugs, that we call it *Persistency Race*, which arises because of the interaction of compilers' optimizations with persistent memory. This chapter makes the following contributions:

- **Persistency Races:** It recognizes the issue of persistency races and formalize the notion of persistency races.
- **Persistency Race Detection:** It presents a baseline race detection approach that can detect persistency races in persistent memory software.
- **Expanding Detection Window:** It recognizes the problem that a crash must occur in a very narrow window of an execution to expose a given persistent race. It then presents an optimized persistency race detection algorithm that expands the window for detecting persistency races.

- Evaluation: It discusses how we have implemented Yashme with a full simulation of $Px86_{sim}$ semantics and applied it on widely-used persistent memory programs including RECIPE persistent memory indexes; the PMDK library; Memcached, a high-performance cache server; and Redis, a persistent memory data store. Yashme found a total of 24 persistency bugs in every single program with 10 benign races.

5.1 Motivation

The correctness of crash consistent data structures rests on careful analysis of the ordering of operations to reason about the potential intermediate states that a crash can leave a data structure in. Applying this type of reasoning to non-atomic memory operations is problematic due to compiler optimizations. Compilers perform optimizations assuming that programs are race-free—other threads (or post-crash executions) will *not* observe updates to the states of non-atomic shared variables until a release operation, *e.g.*, an unlock, is performed. For example, a compiler may implement a non-atomic store using multiple store instructions (*i.e.*, store tearing) or even generate new store instructions (*i.e.*, store inventing) to temporarily stash intermediate results, *e.g.*, if the compiler runs out of registers to store temporary values.

5.1.1 Example Persistency Race

To provide a concrete example, we examined the source code of the Cacheline-Conscious Extendible Hashing (CCEH) hashtable [123], which is distributed with the RECIPE suite of persistent memory indexes [98]. Figure 5.1 presents the `Insert` procedure. It uses a CAS on the `key` field to lock a slot in the hashtable. When the slot is locked, it first writes the value field and then the `key` field. This design relies on the fact that both `value` and `key` fields reside on the same cache line to ensure that the store to the `value` field persists before the

```
int Segment::Insert(Key_t& key, Value_t value,
  size_t loc, size_t key_hash) {
  ...
  if (CAS(&_[slot].key, &LOCK, SENTINEL))
  {
    _[slot].value = value;
    mfence();
    _[slot].key = key;
    ret = 0;
    break;
  }
  ...
}
```

Figure 5.1: The `Segment::Insert` method from the `CCEH` hashtable. The store to the `key` field commits an insertion into the table. This store is non-atomic and thus a poorly timed crash could cause the key to be partially written.

store to the `key` field. Once the key field is written, the key-value value insertion has been committed to the table. The caller of this procedure later flushes both stores to persistent memory.

The problem with this implementation is that since the store to the `key` field is non-atomic, the compiler is free to implement this store with multiple store instructions. While we might imagine this would only occur in cases where the `key` field is not aligned or does not match the native word size of the machine, there are examples of modern compilers implementing such aligned, word-size stores using multiple store instructions [37]. Hence, a crash could potentially cause an incorrect key to be inserted into the table. To fix this bug, the developer should implement the store of the `key` field using an atomic store operation. On an x86 processor, this fix would incur minimal overhead—the atomic store can still be compiled into a normal store instruction as long as the compiler is prevented from performing problematic optimizations (such as store tearing).

5.1.2 Severity of Persistency Races

Ubiquity. The conventional wisdom in concurrent programming is for developers to (1) use locks to protect critical sections and (2) only use atomic operations when strictly necessary for

Table 5.1: Ubiquity of persistency races.

(a) Summary of popular compilers and observed store optimizations that can lead to persistency races.

Compiler	**Arch**	**Store Optimizations**
gcc	ARM64	Use a non-atomic pair of stores for a 64-bit store
gcc & LLVM-clang	ARM64	Replace a seq. of stores of zero with a `memset`
gcc & LLVM-clang	ARM64	Replace a seq. of assignments with a `memmove` or `memcpy`
LLVM-clang	x86-64	Replace a seq. of stores of zero with a `memset`
LLVM-clang	x86-64	Replace a seq. of assignments with a `memcpy`
gcc	x86-64	Replace a seq. of assignments with a `memmove`

(b) Number of memory operations (*i.e.*,`memset`, `memcpy`, `memmove`) used in source code of Fast_Fair, CCEH, and RECIPE benchmarks compared to number in the assembly code generated by `clang` version 11.0 with the `-O3` option.

Prog	**#src-op**	**#asm-op**
CCEH	6	33
Fast_Fair	1	4
P-ART	17	8
P-BwTree	6	15
P-CLHT	0	0
P-Masstree	3	14

performance or progress guarantees. Applying such practices to persistent memory programs inevitably leads to implementations with persistency races. Developing race-free persistent data structures requires extensive use of atomic operations or other techniques like *checksums*.

In our experiments with the RECIPE [98] persistent benchmark suite, we found a total of 19 persistency races in the persistent memory indexes that we were able to execute. [1]

Compilers Performing Store Optimizations. Since persistency races hinge upon certain store optimizations performed by a compiler, we have conducted a study of recent versions of `gcc` (version 10.3) and `LLVM-clang` (version 11.0), two widely-used compilers for native code, with a goal to understand how common these optimizations are on different architectures. As reported in Table 5.1a, store optimizations are widely used by both `gcc`

[1] *i.e.*, CCEH, Fast&Fair, and Recipe benchmarks except P-HOT. We were not able to execute the P-HOT because it did not compile with LLVM.

and `clang` on both ARM and x86 architectures. Whether such optimizations are applied to a particular program depends on the implementations of compilers and libraries (such as `memmove` and `memcpy`).

5.1.3 Empirical Validation

To understand the importance of this issue, we carried out a study on a collection of data structures [72, 123, 98]. We compiled each data structure with `clang` version 11.0 with the `-O3` optimization level. Table 5.1b compares the number of different memory operations (*i.e.*, `memset`, `memcpy`, and `memmove`) that appear in the source code with the number of them that appear in the assembly code. For all programs except P-ART and P-CLHT, the assembly code contains more memory operations than the source code, showing that the compiler has replaced normal stores with memory operations to optimize the code. We carefully audited both P-ART and P-CLHT programs to understand why they do not report more memory operations in their assembly code. For P-ART, it turns out the source code uses 14 `memset` operations inefficiently in the constructors. The compiler optimized them into 3 `memset` and replaced different normal write operations in the source code with 2 `memcpy`. For P-CLHT, we observed that this program uses a lock-free design and critical store operations are defined as volatile and the compiler did not optimize them with memory operations.

The transformation of normal stores into function calls to `memcpy` and `memset` is very common and disabling this optimization in the compiler would likely have significant performance penalties. Compiler optimizations are prone to persistency races not only because of store tearing but also due to store inventing [79]. Compilers can legally invent stores to memory locations that code is guaranteed to write to, *e.g.*, the compiler could generate new store instructions to temporarily stash intermediate results if the compiler runs out of registers to store temporary values. Thus, fixing persistency races requires restricting legal optimizations

and adding constraints to memory operation calls. While this might seem an easy solution, in practice it can be extremely challenging since all of these optimizations are currently standards compliant and thus ensuring safety would *require revising the C/C++ language standard*. This process would likely take many years and there is no guarantee that the standards committee would not simply decide that developers should simply use atomics to avoid persistency races. In addition, there would be a wait for any changes to get rolled out to compilers and libraries.

Although persistency races may not manifest under a particular compiler/architecture, they can lead to bugs that are extremely difficult to detect. For example, a library or compiler update may expose a latent persistency race in recovery code, triggering cascading bugs and even system-wide failures. Such failures can lead to disasters in mission-critical systems—*e.g.*, upgrading the compiler can expose a latent bug in a storage system, causing complete loss of data. As such, there is a pressing need to find and fix such bugs *early on* before they manifest. In fact, the (Intel) developers of PMDK have confirmed [65] that "we do try to ensure that those issues are not present in PMDK and we do extensive validation on compiled binaries to that end; but something can always slip through the cracks; and we definitely don't want to depend on compiler-defined behavior if we can avoid that."

5.2 Yashme Overview

This section presents our basic idea and an overview of how Yashme finds persistency races. We focus our presentation of persistency races on the x86-TSO persistency model. However, persistency races are more general than x86-TSO and will be applicable on other hardware and software persistency models.

Yashme is focused on finding whether an application or library has code for which the

language standard permits the compiler to generate code that exhibits a persistency race. Thus, Yashme instruments the LLVM Intermediate Representation (IR) to call into the Yashme library that simulates the x86-TSO persistency model and monitors for persistency races.

Yashme has two modes of operation: (1) model checking and (2) random execution. In the model checking mode, Yashme explores all executions to find persistency races. This mode is suitable for programs that are relatively small. For large programs for which it is time-consuming to explore all possible executions, Yashme can operate in random mode to detect persistency races.

Yashme's basic approach is to simulate the execution of a PM program, inject a crash, and then simulate the execution of the post-crash recovery program. During the post-crash execution, we compute *which stores may have persisted incorrect values due to the crash*. There are two ways that PM program executions can ensure that stores are fully persisted: (1) the execution explicitly flushes the cache line *after* the store writes to the cache line and *before* the crash to ensure that the stored value was persisted or (2) the post-crash execution reads from a later atomic store to the same cache line and relies on *cache coherence* to ensure the persistency of the store.

Section 5.2.1 presents our approach to handling (1) explicit flushes and (2) cache coherency to determine whether stores are fully persisted. Section 5.2.2 then presents how Yashme leverages the idea of *execution prefixes* to maximize the persistency races that can be detected at a given injected crash.

5.2.1 Basics

Flush Operations. Flush operations can be used to force a cache line to be persisted after a store is fully completed. Without an appropriate flush operation, a non-atomic store can be *partially persisted*. Thus accurately modeling the effects of cache line flush operations is critical for detecting persistency races. We first describe how Yashme models `clflush`.

Figure 5.2 presents an example of using the `clflush` instruction to flush a cache line. In this example, the pre-crash execution stores 1 to the variable x, and then persists this store by executing a `clflush` instruction. To ensure that a `clflush` instruction persists a store, it is critical that the store *happens before* the clflush instruction. Although store s is non-atomic, this execution does not expose a persistency race because s has been flushed before the crash (although other executions can expose a persistency race).

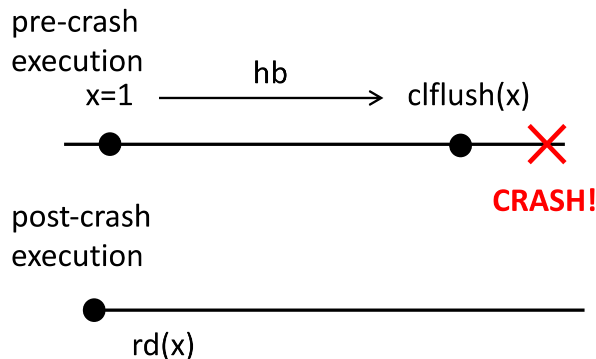


Figure 5.2: Example of using `clflush` to flush the store to x.

We next discuss how Yashme tracks whether stores have been persisted using the `clflush` instruction. Yashme assigns each operation an increasing clock σ that uniquely identifies the operation. Yashme tracks which stores have been flushed by building a map `flushmap`, which maps the clock of each store s to a pair of the form $\langle \tau, \sigma \rangle$, where τ is the identifier of the thread that performs a `clflush` that happens after s, and σ represents the clock that labels that `clflush` such that there is no other clflush that is ordered between the store s and this clflush by happens before. To build this map, when a `clflush` instruction takes effect on

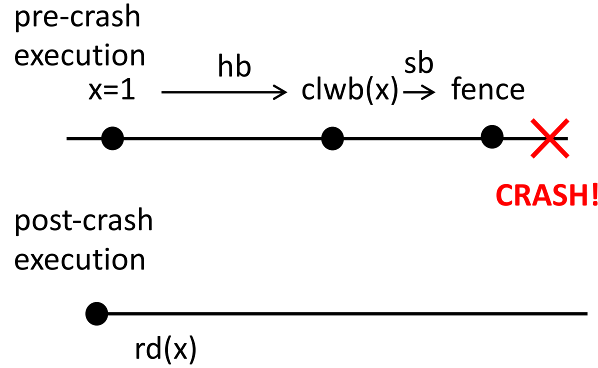


Figure 5.3: Example of using `clwb` to flush the store to x.

the cache, Yashme updates `flushmap` for the latest store to each memory location to include the thread that executed the `clflush` and the clock of the `clflush`. When the post-crash execution reads from `x`, Yashme determines that since `flushmap`($\sigma_{\texttt{x=1}}$) (*i.e.*, `flushmap` applied to the timestamp of the store `x=1`) is not empty, there is no persistency race.

The x86 architecture provides a second, more efficient cache line flush mechanism: the cache line write back instruction `clwb`. Figure 5.3 presents an example in which the pre-crash execution stores 1 to the variable x, and then persists this store by executing a `clwb` instruction and a fence instruction. To persist the store, it is critical that (1) the store happens before the corresponding clwb instruction and (2) the thread that executes clwb also executes a fence instruction later. In our example, although the store `x=1` is non-atomic, this execution does not have a persistency race because `x=1` has been flushed before the crash.

We next extend our approach to track whether stores have been persisted using the `clwb` instruction. Yashme maintains a per-thread set F_τ of `clwb` instructions that have *not* been followed by a fence. When a thread τ executes a fence instruction, Yashme processes each of the `clwb` instructions in F_τ for the thread. When a `clwb` instruction takes effect on the cache, Yashme updates the `flushmap` for the latest store to each memory location (if the store happens before the `clwb` instruction) to include the thread that executed the fence instruction and the sequence number of the fence. Similarly, when the post-crash execution

reads from `x`, Yashme determines that `flushmap`($\sigma_{\texttt{x=1}}$) is not empty and hence there is no persistency race.

Cache Coherence. Cache coherence protocols ensure a total order in the persistence of stores to the same cache line. Figure 5.4 provides an example of an execution that uses cache coherence to avoid a persistency race. Assume that the variables `x` and `y` reside on the same cache line. We use the notation $\texttt{y}_{\texttt{rel}}\texttt{=1}$ to indicate that the store of 1 to `y` is an atomic release store. In the pre-crash execution, the store to `x` happens before the store to `y`. Since `x` and `y` are on the same cache line, cache coherence protocols guarantee `x=1` is completely written to the cache line before $\texttt{y}_{\texttt{rel}}\texttt{=1}$ even if store to `x` is torn into multiple store operations. Since the post-crash execution observes the store to `y`, the cache line is flushed sometime after persisting $\texttt{y}_{\texttt{rel}}\texttt{=1}$ and before the crash event. Consequently, the post-crash execution must also observe the fully completed store to x due to cache coherence. Thus, there is no persistency race in this execution.

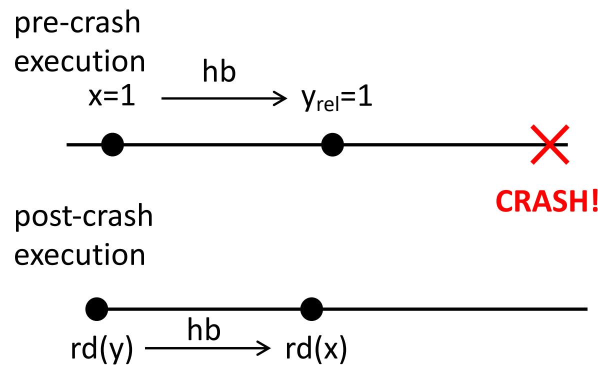


Figure 5.4: Example of coherence preventing persistency races. Assume that the variables `x` and `y` reside on the same cache line and that the store to `y` is an atomic release store.

To model the effect of cache coherence, Yashme maintains a map `lastflush` that maps each cache line to a clock vector that represents the earliest point in the pre-crash execution where the cache line could have been written back to persistent memory—any stores that happen before this point must have been fully persisted. When the post-crash execution reads from an atomic store $\texttt{y}_{\texttt{rel}}\texttt{=1}$, Yashme updates the `lastflush` map to indicate that the

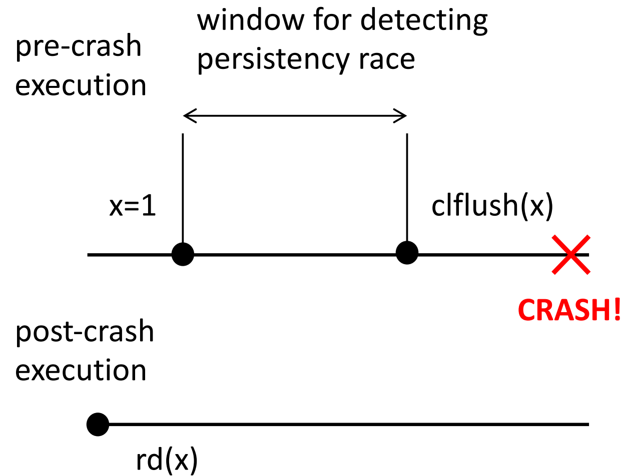


Figure 5.5: Crash misses window for detecting persistency race using core algorithm.

cache line must have been written back some time after the atomic store to `y`. When the post-crash execution reads from the non-atomic store to `x`, Yashme uses the `lastflush` map to determine there is no persistency race because the cache line must have been persisted after the store to `x` was completed. More technical details about the basic algorithm are described in §5.4.

5.2.2 Key Idea: Expanding the Detection Window

Our core approach is to randomly inject crash events and use the aforementioned maps to determine the existence of a persistency race. In particular, we can use the `flushmap` map to detect whether a store was fully persisted via a flush operation and the `lastflush` map to determine whether a store must have been fully persisted because the post-crash execution has read from a later store to the same cache line. However, this approach can only detect persistency races involving stores in *a small window of the pre-crash execution*. In practice, persistent memory programs often flush stores in a timely manner. Hence, detecting a persistency race from a given store requires the crash point to fall into the window between the store and the explicit flushing of its corresponding cache line.

Here we present how we optimize the core approach to improve its ability to detect persistency races. Figure 5.5 shows an example crash scenario to illustrate this problem. In this example, the pre-crash execution writes to the variable x, flushes the write, and then crashes. The post-crash execution then reads from x. Since the crash occurs after the write is flushed, the approach misses detecting the persistency race in this program. To detect this persistency race, the program must crash in the small window of time between the store to x and the corresponding flush. This implies that detecting races using the approach would require injecting crashes in a large number of executions, which can be prohibitively expensive for large programs.

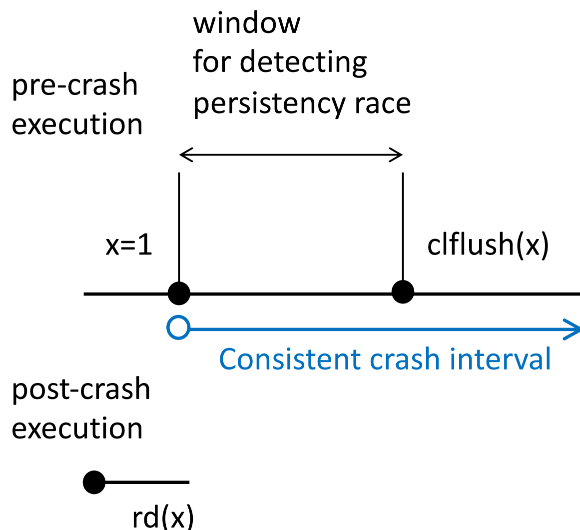


Figure 5.6: Prefixes of pre-crash execution that are consistent with the post-crash execution.

Our key insight for effectively detecting persistency races is that we can check whether the post-crash execution E' has a persistency race with *any prefix* E^+ *of the pre-crash execution* E *that is consistent with* E'. Figure 5.6 illustrates this insight. While the pre-crash execution has flushed the store to x, the post-crash execution has not read from any store that happens after the cache line flush. Thus the post-crash execution at this point is *consistent with any prefix* of the pre-crash execution starting at the store that writes 1 to x. The blue arrow shows the range of consistent prefixes of the pre-crash execution. In Figure 5.7, as

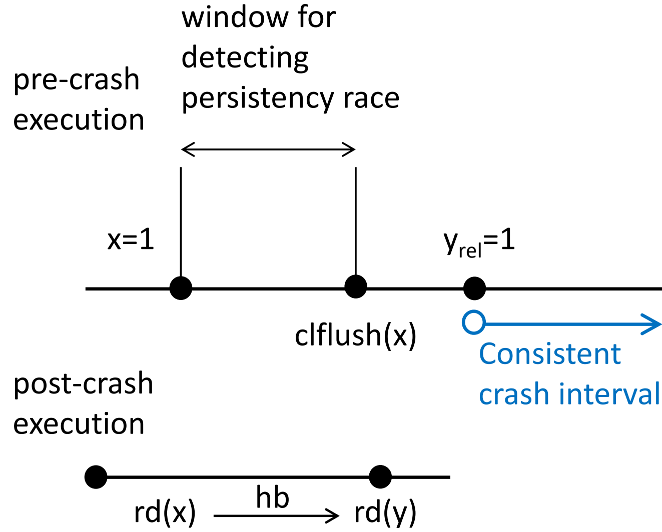


Figure 5.7: Prefixes of pre-crash execution that are consistent with the post-crash execution after reading from `y` residing on the same cache line as `x`.

the post-crash execution reads from the atomic variable y, Yashme updates the constraint of pre-crash execution to be *consistent* with the post-crash execution and to include the clflush(x) instruction.

Consistent Prefixes. Intuitively, a consistent prefix of the pre-crash execution must contain any statement which happens before a pre-crash store that the post-crash execution reads from. Yashme tracks every pre-crash store that the post-crash executions reads from and computes the shortest consistent prefix by using clock-vector-based techniques [44] that are commonly used by race detectors. Yashme uses the consistent prefix to determine whether there is a prefix of the pre-crash execution that did not execute a given `clflush`, `clwb`, or fence instruction. If so, Yashme ignores the instruction when checking for races, because there is a pre-crash execution that does not execute the instruction and yields the same post-crash execution. For example, in Figure 5.6, there is a prefix of the pre-crash execution which does not execute `clflush (x)`. Thus, Yashme can ignore this instruction. However, after reading `y` in Figure 5.7, `clflush(x)` must be executed in all prefixes of the pre-crash execution and cannot be ignored anymore. The constraint prefix helps Yashme find persistency races even

when the crash event is inserted outside of the detection window in the model-checking mode or random mode.

Multi-threaded Programs. *Yashme fully supports multi-threaded programs.* For multi-threaded executions, we use a per-thread prefix of the execution. Note that in the multithreaded case the prefix-based approach can detect persistency races in executions that cannot be generated by inserting a crash event at any point in the pre-crash execution. For example, consider a pre-crash execution in which thread 1 performs a racy store to z, flushes z, then thread 2 sets an atomic flag f to true and a post-crash execution that reads from z and if f is true then reads from z. There is no point in the pre-crash execution trace that we can insert a crash to observe the race in this code. The prefix analysis can determine that the post-crash execution has not read from any store that happens after the flush of z, and therefore we can rearrange the pre-crash execution to a race revealing execution in which thread 1 performs a racy store to z, thread 2 sets an atomic flag f to true, and then it crashes.

5.3 Algorithm Preliminaries

We begin by formalizing our notion of a persistency race. Intuitively, a persistency race occurs if an execution reads from a non-atomic store that could have been made partially persistent. Definition 5.3.1 presents our definition for whether an execution contains a persistency race.

Definition 5.3.1. A load l in a post-crash execution E' that reads from a store s in a pre-crash execution E is a persistency race if (1) s is not atomic, (2) there is no atomic release store s' to the same cache line as s in E such that $s \xrightarrow{hb} s'$ and E' reads from s' before it reads from s, (3) there is no cache-line flush `clflush` to the same cache line as s such that $s \xrightarrow{hb}$`clflush`, and (4) there is no cache-line flush `clwb` to the same cache line as s and a fence `fence` such that $s \xrightarrow{hb}$`clwb` and `clwb`$\xrightarrow{sb}$`fence`.

The first condition ensures that the compiler could legally implement the store with several store instructions or insert other store instructions to the same memory address. The second condition ensures that there were no later atomic stores to the same cache line that were read by the post-crash execution and thus the cache line must have been written back after store s completed. The third and fourth conditions ensure that there was no cache line flush that forced the CPU to flush the entire cache line and thus the processor would have been free to flush the cache line when the store was partially completed or after a compiler inserted store.

5.3.1 Persistency Races in Execution Prefixes

Our key insight for effectively detecting persistency races is that we can check whether the post-crash execution E' has a persistency race with any prefix E^+ of the pre-crash execution E that is consistent with E'.

Consistent Prefixes. For each store s in the pre-crash execution E that some load in the post-crash execution E' reads from, Yashme computes the prefix E_s of the pre-crash execution that happens-before s. Since the post-crash execution E' has observed s, it is only consistent with prefixes of the pre-crash execution that include E_s. Yashme computes E^+ as the union of the sets E_s that correspond to each pre-crash store s that the post-crash execution reads from. This union is efficiently computed using clock vector techniques that are commonly used by race detectors. The prefix E^+ is the smallest prefix of E that is consistent with the post-crash execution.

Yashme uses E^+ to compute whether a store must have been made persistent in all prefixes of E that are consistent with E'. There are two situations in which the post-crash execution must observe s as fully persisted: (1) s was flushed to persistent memory in E^+ or (2) the execution E' has already observed a later store to the same cache line.

Under x86 TSO, recall that there are two ways to flush a cache line after a store s. The program can execute the `clflush` instruction or it can execute the `clwb` instruction followed by a fence. Yashme checks whether there is a `clflush` or `clwb` instruction in E^+ to the same cache line written by the store s that happens after s. If there is a `clflush` instruction, the store s has been made persistent. If there is a `clwb` instruction, Yashme must also check whether the thread that executed the `clwb` instruction later executed an instruction in E^+ with fence semantics. If so, the store s was fully persisted.

If the cache line was not explicitly flushed, Yashme checks whether the post-crash execution has already observed a later store to the same cache line. If not, Yashme reports a persistency race and the pre-crash execution prefix E^+ combined with the post-crash execution E' as a witness.

Theorem 1 (Persistency Race in Execution Prefixes). *Given a pre-crash execution E and a post-crash execution E', there exists a race-revealing pre-crash execution E^+ that has a persistency race with E' if there is a load l in E' that reads from a store s in E and all of the following conditions hold:*

1. *s is not atomic.*
2. *$\nexists s' \in E$,* such that *s' is an atomic release store, and $s \xrightarrow{hb} s' \wedge \mathtt{samecacheline}(s, s') \wedge$ E' reads from s' before it reads from s*
3. *$\nexists \mathtt{clflush} \in E$* such that *$s' \in E, s \xrightarrow{hb} \mathtt{clflush} \wedge \mathtt{samecacheline}(s, \mathtt{clflush}) \wedge \mathtt{clflush} \xrightarrow{hb} s' \wedge$ E' reads from s'*
4. *$\nexists \mathtt{clwb}, \mathtt{fence} \in E$* such that *$s' \in E, s \xrightarrow{hb} \mathtt{clwb} \wedge \mathtt{samecacheline}(s, \mathtt{clwb}) \wedge \mathtt{clwb} \xrightarrow{sb} \mathtt{fence} \wedge$ $\mathtt{fence} \xrightarrow{hb} s' \wedge E'$ reads from s'*

Proof. Assume that we have a pre-crash execution E and post-crash execution E' where a load in E' reads from a store s in E that satisfies the conditions 1 through 4.

We define the execution E^+ to include all statements in E that happen before the stores in E that are read by E' and those stores. We next show that E^+ has a persistency race with the post-crash execution E'.

The store s in E^+ is non-atomic by the assumed condition 1 and thus satisfies condition 1 from Definition 5.3.1. The store s in E^+ satisfies condition 2 by assumption and thus satisfies condition 2 from Definition 5.3.1. By assumption, condition 3 is true for the store s in E. Since E^+ by construction only contains events from E that happen before some store s' that E' reads from, there cannot be a `clflush` in E^+ to the same cache line as s that is ordered after s. Therefore, s in E^+ satisfies condition 3 of Definition 5.3.1. By the same argument, the store s in E^+ satisfies condition 4 from Definition 5.3.1. Therefore, the execution E^+ has a race with E'. □

5.4 Race Detection Algorithm

We next present our persistency race detection algorithm. We begin by presenting the following notations that we will use throughout this chapter:

- We refer to an execution as e.
- We denote a thread using $\tau \in \mathcal{T}$.
- Each thread τ has a store buffer S_τ that keeps a queue of store, `clflush`, and `sfence` operations that have not yet taken effect on the cache.
- Each thread τ has a cache line flush buffer F_τ that stores the set of `clwb` operations that have not yet flushed the cache line to persistent storage.
- A given failure scenario may involve a sequence of multiple executions ending in failures. For example, a persistency race in the recovery procedure would require two crashes:

one to get into the recovery procedure and a second to reveal a bug in the recovery procedure. We record this sequence of executions that have been executed on the persistent store using a stack, referred to as *exec*.

- Function `top`(*exec*) denotes the most recent execution (the current one) on the stack *exec*.
- Function `prev`(e) returns the execution that immediately precedes e in *exec*.
- A global sequence number counter σ_{curr} is used to assign increasing sequence numbers to stores, `clflush`, and `sfence` instructions. Each store, `clflush`, and `sfence` instruction i is assigned a sequence number σ_i. These numbers record the total order in which these instructions take effect in the cache. Using a global sequence number has no performance drawbacks since Yashme already determines the interleaving of threads and has full control over the scheduling of all memory operations.
- Each execution e has a map e.`storemap` that maps each address *addr* to the thread τ and sequence number σ generated at the moment that value was stored.
- Each e has a map e.`flushmap` that maps a store's sequence number σ to a set of pairs $\langle \tau, \sigma \rangle$ of a τ and the sequence number σ of the first flush it performed after the store.
- Each e has a map e.`lastflush` that maps a cache line identifier to a clock vector that is a lower bound for when the cache line was written back.
- Each e has a clock vector $e.CV\text{pre}$ that records how much of the execution e that later executions have observed.

The TSO memory model separates the executions of stores, cache flush operations, and `sfence` operations into two phases: (1) the initial phase that often inserts an operation into a buffer and (2) the second phase that removes the instruction from the buffer and updates the state of the cache or persistent storage. We present our algorithm for each of the stages.

Executing instructions. Figure 5.8 presents our algorithm for the first phase of instruction execution, which inserts an instruction into each thread's local store buffer S_τ. The `mfence` instruction waits until S_τ is empty and then clears the thread's flush buffer F_τ. RMW instructions also have `mfence` like semantics and are handled in the same fashion.

```
 1: function EXEC_STORE(addr, val, τ)
 2:     Enqueue ⟨store, addr, val⟩ into S_τ.
 3: function EXEC_CLFLUSH(addr)
 4:     Enqueue ⟨clflush, addr⟩ into S_τ.
 5: function EXEC_CLWB(addr)
 6:     Enqueue ⟨clwb, addr⟩ into S_τ.
 7: function EXEC_SFENCE
 8:     Enqueue ⟨sfence⟩ into S_τ.
 9: function EXEC_MFENCE
10:     Evict all entries in S_τ.
11:     Flush F_τ.
```

Figure 5.8: Algorithm for executing instructions.

Evicting Operations. Figure 5.9 presents our algorithm for the second phase of instruction execution when the instructions exit the core's store buffer and take effect on the memory system. The Evict_SB procedure for stores assigns a store a clock when it is evicted from the store buffer. It then updates the `storemap` for the execution to record that the address *addr* was written to by the current thread τ and store. The Evict_SB procedure for `clflush` instructions assigns the `clflush` instruction a clock.

Next it updates the `storemap` for each most recent store to an address on the same cache line to include the thread and clock vector for this `clflush` instruction if (1) the store happens before the `clflush` instruction and (2) there is not already a flush instruction in `storemap` that happens before the `clflush`. The Evict_SB procedure for `clwb` instructions adds the `clwb` instruction to the thread's flushbuffer F_τ.

The Evict_SB procedure for `sfence` instruction evicts the `clwb` instructions in the thread's flushbuffer F_τ. The Evict_FB procedure handles evicting a `clwb` instruction from the flushbuffer. It takes as parameters (1) the address, clock vector, and thread identifier for

1: **function** EVICT_SB($\langle$store, $addr$, $val\rangle, \tau$)
2: $\sigma_{\text{curr}} := \sigma_{\text{curr}} + 1$
3: $\texttt{top}(exec)\texttt{.storemap} := \texttt{top}(exec)\texttt{.storemap}[addr \to \langle \tau, \sigma_{\text{curr}} \rangle]$.
4: **function** EVICT_SB($\langle$clflush, $addr$, CVclflush$\rangle, \tau$)
5: $\sigma_{\text{curr}} := \sigma_{\text{curr}} + 1$
6: **for all** $addr_s$ such that $\texttt{CacheID}(addr_s) = \texttt{CacheID}(addr)$ **do**
7: $\langle \tau_s, \sigma_s \rangle = \texttt{top}(exec)\texttt{.storemap}(addr_s)$
8: **if** $\sigma_s < CV\text{clflush}(\tau_s) \wedge$
9: $\nexists \langle \tau', \sigma_{\tau'} \rangle \in \texttt{top}(exec)\texttt{.flushmap}(\sigma_s), \sigma_{\tau'} < CV\text{clflush}(\tau')$ **then**
10: $\texttt{top}(exec)\texttt{.flushmap}(\sigma_s) := \texttt{top}(exec)\texttt{.flushmap}(\sigma_s) \cup \{\langle \sigma_\tau, CV\text{clflush}(\tau_{\text{fence}}) \rangle\}$
11: **function** EVICT_SB($\langle$clwb, $addr$, $CV\rangle, \tau$)
12: Add $\langle addr, CV, \tau \rangle$ to F_τ.
13: **function** EVICT_SB($\langle$sfence$\rangle, \tau$)
14: $\sigma_{\text{curr}} := \sigma_{\text{curr}} + 1$
15: Flush F_τ.
16: **function** EVICT_FB($\langle addr, CV\text{flush} \rangle, \tau_{\text{flush}}, \langle CV\text{fence}, \tau_{\text{fence}} \rangle$)
17: **for all** $addr_s$ such that $\texttt{CacheID}(addr_s) = \texttt{CacheID}(addr)$ **do**
18: $\langle \tau_s, \sigma_s \rangle = \texttt{top}(exec)\texttt{.storemap}(addr)$
19: **if** $\sigma_s < CV\text{flush}(\tau_s) \wedge$
20: $\nexists \langle \tau, \sigma_\tau \rangle \in \texttt{top}(exec)\texttt{.flushmap}(\sigma_s), \sigma_\tau < CV\text{fence}(\tau)$ **then**
21: $\texttt{top}(exec)\texttt{.flushmap}(\sigma_s) := \texttt{top}(exec)\texttt{.flushmap}(\sigma_s) \cup \{\langle \tau_{\text{fence}}, CV\text{fence}(\tau_{\text{fence}}) \rangle\}$

Figure 5.9: Algorithm for evicting store and flush buffers.

the `clwb` instruction and (2) the clock vector and thread identifier for the fence instruction. This procedure updates the `storemap` for each most recent store to an address on the same cache line to include the thread and clock vector for this `clflush` instruction if (1) the store happens before the `clwb` instruction and (2) there is no flush instruction in `storemap` that happens before the fence.

Processing Loads. Figure 5.10 presents our algorithm for handling loads that read from stores performed by a prior execution e. The LOAD_ATOMIC function handles atomic loads. It updates the lower bound for the last time that the respective cache line was flushed in execution e. It then updates the clock vector CVpre that is used to compute the smallest consistent prefix of the execution e. The LOAD_NONATOMIC function handles non-atomic loads. It checks whether the store that the load reads from is ordered after the lower bound for the last time that its cache line was flush. If so, then it checks whether the cache line was flushed after the store. If not, it prints an error. It then updates the clock vector CVpre that is used to compute the smallest consistent prefix of the execution e.

```
1: function LOAD_ATOMIC(addr, ⟨e, CVs, val⟩)
2:     e.lastflush := e.lastflush[CacheID(addr) → e.lastflush(CacheID(addr)) ∪ CVs]
3:     e.CVpre := e.CVpre ∪ CVs
4: function LOAD_NONATOMIC(addr, ⟨e, CVs, τ, val⟩)
5:     if CVs(τ) > e.lastflush(CacheID(addr))(τ) ∧
6:         ∄⟨τ_f, σ_f⟩ ∈ e.flushmap(s), σ_f < e.CVpre(τ_f) then
7:         Print persistency race error
8:     e.CVpre := e.CVpre ∪ CVs
```

Figure 5.10: Algorithm for loads.

5.5 Implementation

Yashme uses the Jaaru open-source model checking infrastructure [62] to simulate program executions. This infrastructure uses an LLVM compiler frontend to automatically instrument programs to intercept reads, writes, clflush instructions, clwb instructions, and memory fences. This infrastructure implements a simulation framework for persistent memory and this framework supports injecting crashes between executions. The simulation framework enables Yashme to reason about all potential effects of cache flushes and precisely control execution while at the same time avoids requiring access to actual hardware supporting these instructions. Yashme is implemented as a plugin for the model checking infrastructure, which reports persistent memory relevant execution events to Yashme. In model checking mode, Yashme systematically injects crashes before every `clflush` or `fence` operation. While Yashme controls multithreaded scheduling to regenerate the same execution, it does not exhaustively explore the space of schedules.

We implement a new mode, *random mode*, on the infrastructure that randomly generates executions in addition to the existing model checking mode. This enables Yashme to execute programs that cannot easily be model checked. At each load, the infrastructure computes a set of candidate stores from pre-crash executions that the load could read from depending on when a cache line was made persistent. Yashme leverages this design to check all of these candidate stores for potential data races using the LOAD_NONATOMIC procedure. In random mode, Yashme randomly explores different concurrent schedules and read choices and

simulates crashes before random fence operations. Users can specify the number of random executions based on the complexity and size of the tool under test.

5.6 Evaluation

We ran our experiments on an Ubuntu Linux 18.04 machine with a 4 core Intel Xeon E3-1245 v3 CPU and 32GB RAM. We used gcc version 7.5.0 and clang version 11.0.0.

Our Benchmarks. We have evaluated Yashme on state-of-the-art persistent memory applications and libraries. Yashme was tested with the RECIPE benchmarks [98], FAST_FAIR [72], CCEH [123], PMDK [33], Memcached [35], and Redis [93]. These frameworks were used in prior works to evaluate their bug-finding tools [62, 126, 107, 108]. Prior testing tools revealed multiple bugs but none of them were capable of detecting persistency races. Yashme is the *first persistency race detector* and has found 24 persistency races in well-tested persistent memory applications.

5.6.1 Methodology

We first evaluated Yashme on a collection of data structures [98, 123, 72]. RECIPE [98] is a collection of concurrent DRAM indexes for persistent memory. We used all RECIPE benchmarks (*i.e.*, P-ART, P-BwTree, P-CLHT, and P-Masstree) except P-HOT because it did not compile with LLVM. CCEH [123] is an efficient hash table for persistent memory. FAST_FAIR [72] is a fault-tolerant B+-tree for persistent memory. We changed the compiler options for these benchmarks to disable optimizations to avoid any optimizations that might reorder memory operations and potentially cause us to miss reporting persistency races. We recompiled these programs with Yashme and used their example test application to drive our testing. These example programs manipulate each data structure through standard insertion,

deletion, and lookup operations.

We also evaluated Yashme on real-world frameworks [33, 93, 35]. PMDK [33] is a collection of libraries and tools developed by Intel for application developers to simplify accessing persistent memory devices. This is the most active open-source PM framework, which has been maintained for 7 years, and is used both by academia and industry. Similar to prior works [62, 108, 107, 126], we used example data structures provided with PMDK to find bugs in the PMDK library (*i.e.*, BTree, CTree, RBTree, Hashmap atomic and Hashmap TX). Redis [93] is a popular in-memory database and memory cache ported by Intel to use both DRAM and persistent memory. It uses PMDK's transaction APIs to store data on persistent memory. We developed our own client to modify the database server using insertion and lookup operations. Memcached [35] is a high-performance distributed memory caching system ported to use persistent memory. This in-memory key-value store uses low-level *libpmem* APIs to flush cache lines to persistent memory. We developed our own client from Memcached's test cases to evaluate Yashme. This client modifies the cache server using insertion and lookup operations. We adapted the original compilation flags for these frameworks and *only* changed the flags to cause them to link against Yashme dynamic library.

We evaluated Yashme with PM indexes with the model checking mode, and we used the random execution mode for Memcached, Redis, and PMDK which are relatively more complicated. The working mode can be specified by a command-line argument.

5.6.2 Race Detection

We run Yashme over RECIPE, CCEH, FAST_FAIR, PMDK, Memcached, and Redis to automatically detect persistency races. We manually deduplicated all race reports since one variable can participate in multiple buggy scenarios. Then, we manually inspected the race reports. Yashme has found a total of 24 races in these programs that are *all new and have*

not been discovered by prior tools. We first discuss our experience with the collection of PM data structures [98, 123, 72]. Table 5.2 reports 19 races we found in these data structures. For each bug, we list the program in which the bugs were found and the field that causes the persistency race.

Table 5.2: Races found in CCEH, FAST_FAIR, and RECIPE benchmarks.

#	Benchmark	Root Cause of Bug
1	CCEH	*value* in *Pair* struct in pair.h
2	CCEH	*key* in *Pair* struct in pair.h
3	FAST_FAIR	*last_index* in *header* class in btree.h
4	FAST_FAIR	*switch_counter* in *header* class in btree.h
5	FAST_FAIR	*key* in *entry* class in btree.h
6	FAST_FAIR	*ptr* in *entry* class in btree.h
7	FAST_FAIR	*root* in *btree* class in btree.h
8	FAST_FAIR	*sibling_ptr* in *header* class in btree.h
9	P-ART	*compactCount* in *N* class in N.h
10	P-ART	*count* in *N* class in N.h
11	P-ART	*deletitionListCount* in *DeletionList* class in Epoche.h
12	P-ART	*headDeletionList* in *DeletionList* class in Epoche.h
13	P-ART	*nodesCount* in *LabelDelete* struct in Epoche.h
14	P-ART	*added* in *DeletionList* class in Epoche.h
15	P-ART	*thresholdCounter* in *DeletionList* class in Epoche.h
16	P-BwTree	*epoch* in *BwTreeBase* class in bwtree.h
17	P-Masstree	*root_* in *masstree* class in masstree.h
18	P-Masstree	*permutation* in *leafnode* class in masstree.h
19	P-Masstree	*next* in *leafnode* class in masstree.h

Due to space constraints, we only elaborate on the persistency races for bug #1 and #2 of Table 5.2. Figure 5.1 shows the source code of pre-crash execution where the program writes to the `key` and `value` fields of a `Segment` and flushes them. Figure 5.11 shows the post crash execution where the program reads from `key` and `value` fields. These variables are non-atomic and thus a poorly timed crash could cause the program to read from from partially persisted stores in the `CCEH:Get` method in the post-crash execution.

Note that the majority of these persistency race bugs are in the core implementation of the data structures, *e.g.*, the key and value fields of the tree. Some of the persistency races were found in memory allocators. These races can lead to different symptoms in the program

```
Value_t CCEH::Get(Key_t& key) {
  ...
  if (dir_->_[slot].key == key) {
    ...
    return dir_->_[slot].value;
  }
  ...
}
```

Figure 5.11: The `CCEH::Get` method from the `CCEH` hashtable reads from the non-atomic `key` and `value` fields.

including (1) accessing an illegal memory address and crashing with a segmentation fault, (2) exiting with assertion failure, and (3) showing wrong or undefined behavior.

Next, we discuss the results for the PMDK benchmarks, Redis, and Memcached [35, 93, 33]. Redis uses PMDK's libpmemobj and transaction APIs to modify persistent memory and Memcached uses PMDK's libpmem API. Table 5.3 reports 5 new persistency races found by Yashme in PMDK, Redis, and Memcached. For each bug, we list the variable that causes the persistency race. The majority of races revealed by Memcached and PMDK testcases involve header fields for the object pool. PMDK library uses these fields for memory management, *e.g.*, defragmentation and garbage collection. Most of these races could be revealed by Redis as well. Yashme found 4 persistency races in Memcached. Similar to PMDK, the majority of these races are related to the internal representation of the object pool, *e.g.*, flags for validating the data. Persistency races in Table 5.3 can corrupt a persisted store leading to data loss. All these races are new and none have been reported before.

Table 5.3: Races found in PMDK, Redis, and Memcached.

#	Benchmark	Root Cause of Bug
1	PMDK	pointer to *ulog_entry* in ulog.c
2	memcached	*valid* variable in *pslab_pool_t* struct in pslab.c
3	memcached	*id* variable in *pslab_t* struct in pslab.c
4	memcached	*it_flags* variable in *item_chunk* struct in memcached.h
5	memcached	*cas* variable in *item* struct in memcached.h

Table 5.2 and Table 5.3 report the variables that cause persistency race in each benchmark. As mentioned in Section 5.1, store tearing is not the only way the compiler can introduce

problems. The compiler can invent stores to locations that are guaranteed to be written to [79]. Thus a persistency race on byte-size fields such as #14 in Table 5.2 and #2 - #4 in Table 5.3 are not safe. This bug report containing information for each problematic store is very beneficial to the developers to reason about different buggy scenarios. To fix these bugs, the developers need to replace racing non-atomic stores with atomic ones (in C++ change int to `atomic<int>` or in C change int to `atomic_int`. On x86 this incurs no overhead if one uses atomic stores with the `memory_order_release` memory ordering, because they are implemented with normal move instructions. But it ensures that compiler optimizations will not tear the store.

5.6.3 Optimization & Performance

Persistency races have a narrow window for detection and this can make detecting persistency races rather challenging. To understand the importance of searching for persistency races in prefixes of available executions, we injected crashes before every fence in the execution of Fast_Fair, CCEH, and the RECIPE benchmarks. We ran Yashme with this optimization (prefix) and without this optimization (baseline) to compare their bug finding capabilities. Table 5.4 reports persistency races detected by these two techniques. For each technique, we report numbers for running a single randomly generated execution. Yashme finds **5**× more persistency races. This demonstrates that the prefix-based approach can find many more persistency races because prefixes generalize executions—many of the derived executions would otherwise be hard to reach. Table 5.4 also reports the times taken to run one random execution on both Yashme and Jaaru, the underlying infrastructure. They have comparable running times because the race checks introduce minimal overheads.

Note that *persistency races were not known before and hence there does not exist any other tool with which we can compare Yashme directly.*

Table 5.4: # races detected w/ and w/o prefix-based expansion for a single execution on RECIPE, PMDK, Memcached, and Redis benchmarks, as well as execution times for both Yashme and Jaaru (the underlying checking infrastructure). Yashme incurs minimal overhead compared to Jaaru.

Benchmark	**Prefix**	**Baseline**	**Yashme Time**	**Jaaru Time**
CCEH	2	0	0.043s	0.041s
Fast_Fair	2	1	0.039s	0.039s
P-ART	0	0	0.046s	0.044s
P-BwTree	0	0	0.034s	0.033s
P-CLHT	0	0	0.159s	0.157s
P-Masstree	2	0	0.037s	0.038s
Btree	1	0	2.541s	2.095s
Ctree	1	0	2.544s	2.099s
RBtree	1	0	2.552s	2.100s
hashmap-atomic	1	0	2.298s	1.896s
hashmap-tx	1	0	2.294s	1.892s
Redis	0	0	5.623s	5.361s
Memcached	4	2	8.032s	8.035s

5.6.4 Bug Reporting and Confirmation

We contacted the authors of these programs to obtain their feedback on the bugs found by Yashme. As of the time of writing this dissertation, we have heard back from the authors of PMDK, Memcached-pmem, and the RECIPE benchmarks. For RECIPE, the authors confirmed that the bugs 9-11 and 19-21 are real bugs. For the persistency races 12-18, they are related to the code of a memory allocator that is known to be crash inconsistent—the RECIPE benchmark suite did not attempt to correctly implement a crash-consistent memory allocator. To be clear, these are all real persistency races, but the code for the memory allocator needs to be replaced anyways and hence they would not fix the bugs 12-18. Furthermore, we reported the bugs 2-5 in Memcached-pmem to its developers. They confirmed that all of these bugs are real. After our bug report, both developers of RECIPE and Memcached-pmem immediately fixed the reported persistency races. These fixes are publicly available on their github repositories. For PMDK, the developers confirmed bug 1.

5.6.5 Discussion

Analyzing Bugs. The current version of the code was compiled with gcc v7.5 and clang v11.0 for x86. We manually investigated the the assembly generated by this compiler and there were many cases of using memory operations (*i.e.*, `memset` and `memcpy`) in object initializations which can lead to store tearing and persistency bugs (*e.g.*, bug #8 in Fast_Fair). For other cases, while these particular compilers may not tear the racing stores, the fact that it can mean that compiler upgrades, architectural changes, or even unrelated changes to the code (*e.g.*, adding new fields to struct/class and breaking variable's alignments) can cause optimizations to generate problematic assembly code. Leaving a persistency race in mission-critical code (*e.g.*, storage systems, self-driving cars, airplane control systems, *etc.*) can lead to catastrophic failures and even disasters from library/compiler/hardware upgrades. *Although these bugs may or may not manifest in today's code, it can suddenly break tomorrow's executions even if user code is not changed at all.* As confirmed by PMDK developers, "making sure their code does not depend on compiler/library behaviors" is their daily routine and "they do extensive validation to that end".

Although the developer can manually inspect the assembly code on every compiler, architectural, and code change, this approach is very time-consuming and not practically possible for large source code bases. Yashme automatically flags out such bugs caused by miscompiling that can corrupt a large data store. We believe it is strictly necessary to fix such bugs in mission-critical code that cannot afford to break.

Benign Issues. Programs can access inconsistent data in the post-crash execution. However, these program accesses are not necessarily followed by sensitive operations that use this data. For examples, programs can use customized fault tolerance techniques to detect data inconsistency and ignore the inconsistent data. For example, PMDK, Redis, and Memcached use a checksum-based strategy to verify that data is consistent. This strategy

computes a checksum on the data and writes the checksum. Before using the data, the program first verifies the checksum to validate data integrity. Even if these programs read from partially-persistent data, such data are not used as they fail the checksum validation. In addition to the 24 persistency races, Yashme found 10 bugs that are benign issues due to checksums (although these are still true persistency races by definition). **Yashme did not report any other types of benign issues other than the ones from checksums.** A future implementation of Yashme could use annotations to suppress race warnings from stores that are read by the checksum validation procedure.

As with any dynamic tool, Yashme can only find bugs in the executions it explores. As such, Yashme may miss bugs in unexplored executions.

Persistency Races on eADR CPUs. Persistency races are still possible on eADR systems [73] where flushing is not required. The absence of races on a non-eADR system implies the absence of races on eADR systems, but the opposite is not true. Thus, Yashme can be used as is to guarantee the absence of races on the eADR systems. However, Yashme could be adapted to only detect races that are possible on eADR systems by adding support to handle the slightly different persistency semantics.

Chapter 6

Related Work

6.1 Programming Models for Persistent Memory

There is a great deal of work on building programming systems that allow developers to use PM in a reliable way without knowing the details of PM. For example, a line of work [18, 30, 161, 51, 53, 104] proposes to use (software or hardware) transactions to provide (failure and thread) atomicity. Another line of work [9, 10, 20, 68, 77, 105] advocates use of locks or synchronization-free regions [59]. Jaaru, PSan, and Yashme are complementary to these approaches, it can be used to check the correctness of their implementation.

6.2 Crash Consistency Detection

There exists a large body of work on testing [87, 95, 120, 160], checking [119, 139, 162, 163], and formally verifying [23, 24, 146] file system implementations to find and eliminate crash consistency bugs. Fuzzing techniques such as Janus [160] and Hydra [87] mutate disk images and file operations to explore states of file system code. Using heuristics, B3 [120]

employs a bounded testing technique to explore states in a bounded space. EXPLODE [162], FiSC [163], and SAMC [99] use model checkers to systematically explore states of a file system implementation. There is much work on torn writes in disk where only part of a multi-sector write completes. File systems use a variety of techniques to ensure consistency, such as shadow updates [11, 138], journaling [142], soft updates [116], and post-reboot checking [134, 117]. Although crash consistency bugs in file systems bear similarities with bugs in PM programs, they are fundamentally different in the access granularity as well as how writes are performed.

6.2.1 Model Checking

Model checking has been extensively studied. Stateless model checking techniques do not explicitly track which program states have been visited and instead focus on enumerating schedules [54, 55, 56, 121, 122]. To make model checking more efficient, researchers propose dynamic partial order reduction techniques [21, 46, 96, 140, 143, 144, 147] that exploit state equivalence to reduce search space.

Recent work model-checks multi-threaded programs against the TSO and PSO memory models [2, 71, 168] and the release-acquire fragment of C/C++ [3, 15, 38, 88].

Model checking is also widely used to find bugs in systems code. Model checkers such as EXPLODE [162], FiSC [163], and SAMC [99] check file system code. However, directly applying these techniques would dictate enumerating all possible PM states, which is not feasible given that PM is byte-addressable and has orders of magnitude more states than a disk.

Model checking [95, 126] has been used to find bugs in persistent memory programs.

Yat [95] 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 [126] finds bugs in persistent memory programs by using symbolic execution. It tracks the state of persistent memory objects and their corresponding cache lines in the program, *i.e.*, whether the cache line is modified. Agamotto updates constraints on these states as the program runs and uses them to identify different types of persistency bugs including correctness, performance, and custom user-defined bugs. It uses a priority-based static analysis to steer program execution to program states that frequently modify PM. This approach can miss bugs because it only reasons about whether stores are made persistent and does not reason about the order that stores are made persistent. Persistency races result from the interaction between the pre-crash and post-crash executions. Agamotto only explores the pre-crash execution and thus cannot be easily extended to find persistency races.

6.2.2 Persistent Memory Testing Tools

There is a recent line of work on checking/testing PM programs to find bugs. Pmemcheck [85] checks how many stores were not made persistent and detects memory overwrites using binary rewriting. PMTest [108] lets developers annotate a program with checking rules to infer the persistency status of writes and ordering constraints between writes. XFDetector [107] uses a finite state machine to track the consistency and persistency of persistent data by implementing a shadow PM, and with the help of user-provided annotations to identify commit variables. Although these tools are able to find many bugs, none of these tools can systematically explore the state space. In particular, they simply check whether data is persisted appropriately. However, buggy data structures can have windows of vulnerability when crashes can cause failures even if all data is persisted and ordered. This motivates us to develop Jaaru, a model checker that can thoroughly explore states to find bugs. Agamotto [126] finds bugs in persistent memory programs by using symbolic execution. It tracks the state of persistent memory objects and their corresponding cache lines in the program, *i.e.*, whether the cache line is modified. Jaaru [62] takes a constraint-based approach

to enumerating executions that can drastically reduce the number of post-failure executions. PMDebugger [39] is a debugger developed on top of Valgrind that tracks operations to find persistency bugs. PMDebugger relies on programmers to explicitly annotate ordering constraints. Although these tools are able to find many bugs, they all face limitations such as requiring user annotations, not catching bugs with invisible symptoms, or leaving developers with long traces that they must manually analyze. On the contrary, PSan can detect and localize bugs without user annotations. PMFuzz [106] is an automatic test case generator for persistent memory program. It fuzzes different inputs to incrementally generates test cases that produce different memory image. PSan can be used along with PMFuzz to check the executions explored by generated test cases. Hippocrates [125] is a tool for automatically fixing persistency bugs by analyzing crash information.

Yashme is built on Jaaru's constraint-based execution engine that enables Yashme to observe many possible stores that a load can read from (depending on when the cache line was evicted) and checks for persistency races in all of them. Although all existing tools are able to find many bugs, they all effectively treat writes to persistent memory as atomic or cannot distinguish if a source-level store operation is torn at the binary level. Thus, the current implementations of all previous tools are unable to detect persistency races.

6.2.3 Relation of Robustness to Prior Work

Robustness to weak persistency models builds on a rich literature of defining the correctness of concurrent code by relating concurrent executions to other executions. In the context of weak memory models, a program is robust [13, 128, 94, 118] against a weak memory model if all of the program's executions under the weak memory model are permitted under the sequential consistency model.

Robustness provides a rigorous foundation that subsumes prior work that relied on heuristics

Table 6.1: Comparison with other tools; robustness subsumes ordering heuristics/conditions used in existing tools.

Tool	**Persistent Order**
PSan	Robustness
Witcher [49]	Dependence heuristic
PMDebugger [39]	User annotations
PMTest [108]	User annotations
XFDetector [107]	Commit store annotations
Jaaru [62]	Crash/assertion failure
Yat [95]	Crash/assertion failure
Agamotto [126]	Does not check order
Pmemcheck [85]	Does not check order
PMFuzz [106]	Just fuzzes input, uses Pmemcheck or XFDetector for checking
Hippocrates [125]	Does not repair ordering bugs

or annotations to check whether stores are persisted in the correct order. Note that in the comparison with prior work and throughout this dissertation, 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 6.1 summarizes the approaches other tools take to checking the order of PM stores. All these conditions are essentially instances of robustness violations. Witcher [49] 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 [108] and PMDebugger [39] 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 [107] requires that ordering constraints are specified implicitly by annotating a set of commit variables, otherwise it can report false positive. Jaaru [62] and Yat [95] only detect ordering bugs when the program crashes or asserts, and developers must manually localize the bug. Pmemcheck [85] and Agamotto [126] 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.

6.3 Constructive Approaches

In addition to these general-purpose debugging tools, there is a rich literature on systematic transforms for lock-free data structures to use persistent memory [149, 9, 78, 34, 48, 36]. Most of these constructive approaches leverage different techniques to deduce flush and fence instructions for lock-free programs. More recently, Mirror [48] 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. [78] introduce the notion of *durable linearazibility* to data-race-free programs to become crash consistent. *Durable linearazibility* is implemented as a set of transformation rules, which preserve the original happens-before ordering for persistent memory. While these constructive approaches suffice to ensure robustness, they may inject unnecessary fence and flush instructions. PSan can determine that weaker transformations also preserve robustness in several cases: (1) To avoid reasoning about the challenging corners of weak memory models, many implementations might follow the standard advice to use only release/acquire or SC atomics even when weaker atomics would suffice. But with stronger atomics, prior work [78, 34] would generate unnecessary fence instructions. (2) These transformations do not consider that consecutive writes to the same cache line eliminate the need for flush/fence operations between the writes. (3) If there are multiple relaxed stores to the same cache line, some of these techniques will cause a flush to be generated after each store. (4) If some persistent memory locations are used as temporary storage and are never read from after a crash, prior approaches would force stores to those locations to have flushes. (5) An existing RMW operation may suffice to serve as the needed fence instruction. In Section 4.1, we describe the notion of robustness as a sufficient requirement to guarantee crash consistency in lock-free frameworks.

6.4 Fix Suggestion

PSan can save significant manual effort compared to repairing bugs without a tool. While Hippocrates [125] 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 4.5). 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.*

Chapter 7

Conclusion

7.1 Summary

Assuring the correctness of persistent memory programs with respect to failure is very challenging. This is because exposing a bug requires the machine to fail at a specific instruction and also it depends on the state of the cache. Prior tools and techniques had shortcomings including requiring annotations by the users, not testing the recovery code, or requiring a complete test suite to fully check the program. This dissertation presents Jaaru as the first efficient model checker for persistent memory programs. Jaaru uses a constraint refinement-based approach that drastically reduces the number of executions that must be explored. Jaaru is the first tool to fully model the TSO persistent memory model. Our evaluation shows that Jaaru effectively finds bugs in our benchmark applications and that Jaaru reduces the number of executions that must be explored by several orders of magnitude.

Jaaru and prior testing tools require developers to manually inspect long execution traces to fix persistency bugs. These tools only can report bugs that have visible symptoms such as assertion failure or segmentation fault. The issue is that not all bugs have visible

manifestations such as silent data corruption bugs. This dissertation 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 for the stores that are missing flush or fence instructions. PSan found 48 bugs (including 17 new bugs) in 13 popular PM benchmarks ranging from DRAM indexes to real-world applications. This shows the effectiveness and usefulness of PSan in detecting and fixing persistency bugs in these programs.

All prior testing tools focused on detecting bugs that originated from developers' errors in inserting correct flush and fence instructions. This dissertation formally defines a new class of bugs in PM programs, persistency races, and presents Yashme, the first tool to detect persistency races. This dissertation describes basic approaches in detecting persistency races and the challenges in using them in practice. To resolve these challenges, Yashme builds on a novel technique that improves persistency race detection by checking prefixes of the pre-crash execution. This technique enables Yashme to detect persistency races in the executions without exploring them. Our evaluation on a collection of persistent memory data structures and three real-world applications and libraries shows that Yashme has found 24 real persistency races in our applications.

7.2 Limitations and Future Directions

Jaaru has its limitations and can be extended in several directions. First, the existing implementation only supports X86 architecture. Persistent memory can be beneficial in programs developed for ARM64 architecture. Jaaru can be extended to support ARM persistency semantics and model check these programs. Second, Jaaru can only detect bugs that can cause the program to crash. However, Jaaru controls concurrent schedule and fully simulates TSO memory model, as future work, it can be extended to support different types

of bugs such as performance bugs, concurrency bugs with respect to persistent memory, and *etc.*. Third, The existing implementation of Jaaru systematically simulates crashes and explores executions. However, for some big-data programs using only the constraint refinement technique may not be sufficient. As an extension, Jaaru could benefit from different fuzzing techniques to improve the search space exploration algorithm.

PSan presents robustness as a sufficient correctness condition. Some persistent memory programs are tolerant to reading stale values and use different techniques such as checksum computation. The existing implementation of PSan reports false positives for such techniques. PSan requires developers to annotate the usage of checksum to not report false positives which incurs a burden on users. PSan can benefit from static analysis to identify the code locations that the program can read from unflushed stores and discard them in order to not report false positives to the user. In addition, robustness provides a great foundation to assure the correct usage of flush and fence instructions in the program. The existing version of PSan only suggests which stores are missing the flush and fence instructions in the program. PSan can be extended to apply fixes automatically to the source code instead of suggesting the fixes to the users. The ideal case is to run a non-persistent memory program with PSan and PSan automatically insert flush and fence operations in the program. This would yield to automatically transforming any generic program to a correctly persisted version that can run on persistent memory without having any crash consistency bugs.

Programs can access inconsistent data in the postcrash execution. However, these program accesses are not necessarily followed by sensitive operations that use this data. For example, programs can use customized fault tolerance techniques to detect data inconsistency and ignore the inconsistent data. The existing version of Yashme reports false positives for such cases. As future work, Yashme can benefit from annotations or static analysis to identify code sections that use these customized fault tolerance techniques. The existing version of Yashme only reports variables that are causing persistency races to the users. However, the

user is responsible to make that variable atomic and propagate this change throughout the entire code base. Yashme can be extended to automatically apply these fixes to the code base without any manual efforts by the user.

This section provided the limitations and directions on how the presented tools can be extended. We only provided the high-level ideas and the design, implementation, and evaluation of them remain as future work.

Bibliography

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 2014 Symposium on Principles of Programming Languages*, POPL '14, pages 373–384, 2014.

[2] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 353–367, 2015.

[3] P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo. Optimal stateless model checking under the release-acquire semantics. *Proceedings of the ACM on Programming Languages*, October 2018.

[4] P. Alcorn. Intel optane dimm pricing: $695 for 128gb, $2595 for 256gb, $7816 for 512gb. `https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html`, 2019.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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, 2016.

[11] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. Technical report, Sun Microsystems Solaris, 2003.

[12] 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.

[13] 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.

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

[15] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 12–21, 2007.

[16] 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.

[17] 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.

[18] 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, 2018.

[19] 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.

[20] 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 and Applications*, OOPSLA '14, pages 433–452, 2014.

[21] K. Chatterjee, A. Pavlogiannis, and V. Toman. Value-centric dynamic partial order reduction. *Proceeding of the ACM on Programming Languages*, 3(OOPSLA), October 2019.

[22] H. Chauhan, I. Calciu, V. Chidambaram, E. Schkufza, O. Mutlu, and P. Subrahmanyam. NVMOVE: Helping programmers move to Byte-Based persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*, Savannah, GA, Nov. 2016. USENIX Association.

[23] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 270–286, 2017.

[24] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, 2015.

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

[26] 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.

[27] 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.

[28] 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.

[29] N. Chopra, R. Pai, and D. D'Souza. Data races and static analysis for interrupt-driven kernels. In L. Caires, editor, *Programming Languages and Systems*, pages 697–723, Cham, 2019. Springer International Publishing.

[30] 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 *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 105–118, 2011.

[31] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[32] I. Corporation. 3D XPoint^{TM}: A breakthrough in non-volatile memory technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html`, 2018.

[33] I. Corporation. Persistent memory development kit. `https://pmem.io/pmdk/`, 2020.

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

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

[36] 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.

[37] W. Deacon. Re: [patch 1/1] fix: trace sched switch start/stop racy updates. `https://lore.kernel.org/lkml/20190821103200.kpufwtviqhpbuv2n@willie-the-truck/`, August 2019.

[38] B. Demsky and P. Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 20–36, October 2015.

[39] 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.

[40] 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.

[41] 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.

[42] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–255, New York, NY, USA, 2007. Association for Computing Machinery.

[43] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252, New York, NY, USA, 2003. Association for Computing Machinery.

[44] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language*

Design and Implementation, pages 121–133, New York, NY, USA, 2009. Association for Computing Machinery.

[45] 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.

[46] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, 2005.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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, 2020.

[52] 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.

[53] 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 '17, pages 70–81, 2017.

[54] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Berlin, Heidelberg, 1996.

[55] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, 1997.

[56] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

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

[58] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. `https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/`, November 2008.

[59] 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 '18, pages 46–61, 2018.

[60] 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.

[61] H. Gorjiara, W. Luo, A. Lee, G. H. Xu, and B. Demsky. *Checking Robustness to Weak Persistency Models*, pages 1218–1232. Association for Computing Machinery, New York, NY, USA, 2022.

[62] 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.

[63] 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.

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

[65] P. B. Hamed Gorjiara, Brian Demsky. Email exchanges with PMDK developers, 2021.

[66] S. Haria, M. D. Hill, and M. M. Swift. *MOD: Minimally Ordered Durable Datastructures for Persistent Memory*, page 775–788. Association for Computing Machinery, New York, NY, USA, 2020.

[67] 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.

[68] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys '17, pages 468–482, 2017.

[69] 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.

[70] J. Huang, P. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*, pages 337–348, New York, NY, USA, June 2014. Association for Computing Machinery.

[71] S. Huang and J. Huang. Maximal causality reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '16, pages 447–461, 2016.

[72] 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.

[73] 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.

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

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

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

[77] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, 2016.

[78] 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.

[79] Jade Alglave, Will Deacon, Boqun Feng, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Piggin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. Who's afraid of a big bad optimizing compiler?, 2019.

[80] 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.

[81] 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.

[82] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 389–400, New York, NY, USA, 2011. Association for Computing Machinery.

[83] 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.

[84] 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.

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

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

[87] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 147–161, 2019.

[88] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proceedings of the ACM on Programming Languages*, 2(POPL), December 2017.

[89] 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.

[90] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '13, pages 256–267, 2013.

[91] 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.

[92] 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.

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

[94] 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.

[95] 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.

[96] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *International Conference on Fundamental Approaches to Software Engineering*, pages 308–322. Springer, 2010.

[97] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, 2009.

[98] 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.

[99] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 399–414, USA, 2014. USENIX Association.

[100] 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.

[101] 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.

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

[103] 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.

[104] 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 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, 2017.

[105] 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 '18, pages 258–270, 2018.

[106] 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.

[107] 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 25th 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.

[108] 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 24th 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.

[109] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 37–48, New York, NY, USA, 2006. Association for Computing Machinery.

[110] 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.

[111] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 210–221, New York, NY, USA, 2010. Association for Computing Machinery.

[112] P. Mahapatra, M. D. Hill, and M. M. Swift. Don't persist all: Efficient persistent data structures. *arXiv preprint arXiv:1905.13011*, 2019.

[113] 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.

[114] 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.

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

[116] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*, Monterey, CA, June 1999. USENIX Association.

[117] M. K. Mckusick and T. J. Kowalski. Fsck - the unix file system check program, 1994.

[118] 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.

[119] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 361–377, 2015.

[120] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 33–50, 2018.

[121] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A systematic testing tool for concurrent software. *Logic-Based Program Synthesis and Transformation*, page 16, November 2007.

[122] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 267–280, 2008.

[123] 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.

[124] 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.

[125] 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.

[126] I. Neal, B. Reeves, B. Stoler, A. Quinn, Y. Kwon, S. Peter, and B. Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI '20, 2020.

[127] 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.

[128] 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.

[129] 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.

[130] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 25–36, New York, NY, USA, 2009. Association for Computing Machinery.

[131] 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.

[132] 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.

[133] 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.

[134] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 206–220, New York, NY, USA, 2005. Association for Computing Machinery.

[135] 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.

[136] 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.

[137] 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.

[138] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, feb 1992.

[139] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 270–280, 2009.

[140] O. Saarikivi, K. Kähkönen, and K. Heljanko. Improving dynamic partial order reductions for concolic testing. In *2012 12th International Conference on Application of Concurrency to System Design*, pages 132–141. IEEE, 2012.

[141] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15:391–411, November 1997.

[142] S. T. Sct and S. C. Tweedie. Journaling the linux ext2fs filesystem, 1998.

[143] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, FASE'06, pages 339–356. Springer, 2006.

[144] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proceedings of the 2nd International Haifa Verification Conference on Hardware and Software, Verification and Testing*, HVC'06, pages 166–182. Springer, 2006.

[145] 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.

[146] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 1–16, 2016.

[147] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 219–234. Springer, 2012.

[148] 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.

[149] 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.

[150] 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.

[151] 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.

[152] Y. Wang, L. Wang, T. Yu, J. Zhao, and X. Li. Automatic detection and validation of race conditions in interrupt-driven embedded software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 113–124, New York, NY, USA, 2017. Association for Computing Machinery.

[153] H. S. P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai. Metal-oxide RRAM. *Proceedings of the IEEE*, 100(6):1951–1970, June 2012.

[154] H. S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, December 2010.

[155] 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.

[156] 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.

[157] 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.

[158] 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.

[159] 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.

[160] W. Xu, H. Moon, S. Kashyap, P. Tseng, and T. Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 818–834, 2019.

[161] 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.

[162] J. Yang, C. Sar, and D. Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIXChapter 1 Introduction Persistent memory (PM) technologies, such as phase change memory (PCM) [94, 149, 160], resistive random-access memory (RRAM) [148], Spin-Transfer Torque memory (STT- MRAM) [87], or 3D XPoint [30], promise to combine the performance and flexibility of DRAM with the persistency of flash storage. Persistent memory revolutionizes the storage-memory hierarchy [119, 128, 72]. This technology became commercially available with Intel's release of Optane DC Persistent Memory [71]. Such PM technologies are cheaper than DRAM per GB of capacity [4]. 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. Persistent memory can potentially change the way programs manipulate data structures to achieve greater performance; with PM, programs can use a single copy of a data structure both as an in-memory working data structure and as a persistent store of the data, eliminating the overhead of serialization and deserialization of data in the program executions. The low latency and durability of PM have spurred the development and redesign of file systems [38, 88, 89, 107, 145, 151, 153, 154, 27, 80], databases [7, 90, 33, 109, 122], 1 Symposium on Operating Systems Design and Implementation, OSDI '06, page 10, USA, 2006. USENIX Association.

[163] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, November 2006.

[164] 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.

[165] H. Yoon, J. Meza, N. Muralimanohar, N. P. Jouppi, and O. Mutlu. Efficient data mapping and buffering techniques for multilevel cell phase-change memories. *ACM Transactions on Architecture and Code Optimization*, 11(4):40:1–40:25, December 2014.

[166] 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.

[167] 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.

[168] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 250–259, 2015.

[169] W. Zhang, C. Sun, and S. Lu. Conmem: Detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 179–192, New York, NY, USA, 2010. Association for Computing Machinery.

[170] 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.

[171] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 121–132, Scottsdale, AZ, USA, 2007. Institute of Electrical and Electronics Engineers.

[172] 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.

Appendix A

Jaaru Artifact Evaluation

A.1 Abstract

This artifact contains a vagrant repository that downloads and compiles the source code for Jaaru, its companion compiler pass, and benchmarks. The artifact enables users to reproduce the bugs that are found by Jaaru in PMDK (i.e., Figure 3.11 of the dissertation) and RECIPE (i.e., Figure 3.12) as well as the performance results to compare Jaaru with Yat (i.e., Figure 3.13).

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

- **Algorithm:** Lazy exhaustive model-checking
- **Program:** Jaaru
- **Compilation:** GCC 7.5.0 and Clang
- **Binary:** Instrumentation LLVM pass

- **Data set:** RECIPE and PMDK benchmarks
- **Run-time environment:** Any system that can run Vagrant
- **Hardware:** One 6 core 3.7 GHz Intel i7 machine with 32 GB DDR4 memory
- **Run-time state:** Managed by our x86 simulator
- **Execution:** Automated by our tooling system
- **Metrics:** Crashing the program under test
- **Output:** Program crash for bugs. Logging performance measurement for executions.
- **Experiments:** Regenerating all bugs found by Jaaru. Reproducing performance results and comparing them with Yat (fully automated by our custom tooling)
- **How much disk space required (approximately)?:** 80G
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** About 20 mins
- **Publicly available?:** Yes. Open-source on GitHub
- **Code licenses (if publicly available)?:** GNU GENERAL PUBLIC LICENSE Version 2
- **Data licenses (if publicly available)?:** BSD-3-Clause and Apache License 2.0.
- **Workflow framework used?:** Vagrant.
- **Archived (provide DOI)?:** https://doi.org/10.6084/m9.figshare.13392338

A.3 Description

Our workflow has four primary parts: (1) creating a virtual machine and installing dependencies needed to reproduce our results, (2) downloading the source code of Jaaru and the benchmarks and building them, (3) providing the parameters corresponding to each bug to reproduce the bugs, and (4) running the benchmarks to compare Jaaru with the naive exhaustive approach (*i.e.*, Yat). After the experiment, the corresponding output files are generated for each bug and each performance measurement.

How to Access

All source code is open-source and available on GitHub. Our packaging requires cloning the vagrant system repository from https://github.com/uci-plrg/jaaru-vagrant. As described in the *README.md* file of the repository, you will need to install a VirtualBox VM and Vagrant on your machine. Then, the vagrant setup will install the required dependencies and download the source code of the tools from our git repository. Next, it builds each tool on the virtual machine.

Hardware Dependencies

Our tooling system and Jaaru have no special hardware dependencies and it can be running on any x86 machine with at least 32GB RAM and 4 cores.

Software Dependencies

To run our system, the following should be installed on the local machine:

- Linux (we tested on Ubuntu)
- Vagrant
- VirtualBox
- Vagrant-disksize plugin

Data Sets

To evaluate Jaaru, our tooling system downloads the source code of RECIPE and PMDK from our git repository. We forked a branch from the original source code of these benchmarks that don't contain our bug fixes. The tooling system automatically sets up and builds these benchmarks and runs them under Jaaru to identify bugs in them.

A.4 Installation

Please see the *README.md* file of the https://github.com/uci-plrg/jaaru-vagrant repository, which contains a detailed step-by-step guide to setup Jaaru on a virtual machine. Then, our scripts automatically do the following:

1. Install all the dependencies needed to install and evaluate Jaaru on different benchmarks.
2. Check out the source code for LLVM, Jaaru, Jaaru's LLVM pass, RECIPE, and PMDK.
3. Include Jaaru's LLVM pass to LLVM and building it
4. Set up and build Jaaru with two different configurations (One for RECIPE that uses *libvmemmalloc*, and one for PMDK that uses *libpmem* APIs).

5. Set up and building RECIPE (including CCEH, FAST_FAIR, P-ART, P-BwTree, P-CLHT, and P-Masstree benchmarks) and PMDK benchmarks.
6. Generate three scripts in the *home (or ∼/)* directory of the virtual machine to generate the results.

Once the scripts are finished setting up the virtual machine and benchmarks, the user can use Jaaru on the virtual machine to further evaluate different benchmarks or regenerate our evaluation results.

A.5 Experiment Workflow

After setting up the virtual machine, the user can use 'vagrant ssh' to connect to the VM and use Jaaru. The detailed instructions to run the suggested workflow is included in the *README.md* file of https://github.com/uci-plrg/jaaru-vagrant repository. There are three scripts in the *home* directory of the virtual machine that user can run:

recipe-perf.sh : It runs the RECIPE benchmarks using Jaaru and gathers measurements to compare Jaaru against Yat. For each benchmark, the corresponding log file is generated in *∼/results/recipe-performance.*

recipe-bugs.sh : It runs the RECIPE benchmarks using Jaaru and sets the corresponding parameters to reproduce each bug. For each bug, the corresponding log file is generated in *∼/results/recipe-bugs.*

pmdk-bugs.sh : It runs PMDK benchmarks by using Jaaru and set the corresponding parameters to reproduce each bug. For each bug, the corresponding log file is generated in *∼/results/pmdk-bugs.*

In our tooling system, the *timeout* is used in both *recipe-bugs.sh* and *pmdk-bugs.sh* scripts to recover from segmentation fault. The timeout needs to be adjusted if the user uses a slower machine.

A.6 Evaluation and Expected Result

After successfully running the experiment using our scripts, the *results* directory is generated in the *home* directory. This directory contains the following results:

RECIPE

Performance Results: For each RECIPE benchmark, there is a *-Performance* file in the *∼/results/recipe-performance* directory (for a total of 6 files). These files contain the performance information corresponding to Figure 3.13.

Bugs: There are 18 files in *∼/results/recipe-bugs* directory. Each file contains the corresponding logs for the bug that Jaaru found. Figure A.1 contains information about how Jaaru identified each bug correspond to Figure 3.12.

PMDK

There are 7 files in *∼/results/pmdk-bugs* directory. Each file contains the corresponding logs for the bug that Jaaru found. Figure A.2 contains information about how Jaaru identified each bug correspond to Figure 3.11.

#	Bug ID	Cause of Bug
1	CCEH-1	Getting stuck in an infinite loop
2	CCEH-2	Segmentation fault in the program
3	CCEH-3	Segmentation fault in the program
4	FAST_FAIR-1	Segmentation fault in the program
5	FAST_FAIR-2	Segmentation fault in the program
6	FAST_FAIR-3	Segmentation fault in the program
7	P-ART-1	Segmentation fault in the program
8	P-ART-2	Illegal memory access in the program
9	P-ART-3	Getting stuck in an infinite loop
10	P-BwTree-1	Segmentation fault in the program
11	P-BwTree-2	Segmentation fault in the program
12	P-BwTree-3	Segmentation fault in the program
13	P-BwTree-4	Segmentation fault in the program
14	P-BwTree-5	Segmentation fault in the program
15	P-CLHT-1	Illegal memory access in the program
16	P-CLHT-2	Illegal memory access in the program
17	P-CLHT-3	Getting stuck in an infinite loop
18	P-MassTree-1	Illegal memory access in the program

Figure A.1: More information about the bugs that are found by Jaaru in RECIPE benchmarks.

#	Benchmark Found	Symptom
1	Btree*	Illegal memory access at btree_map.c:89
2	Btree	Failed to open pool error
3	Hashmap_atomic*	Assertion failure at heap.c:533
4	CTree*	Assertion failure at obj.c:1523
5	Hashmap_atomic*	Assertion failure at pmalloc.c:270
6	Hashmap_tx*	Illegal memory access at obj.c:1528
7	RBTree*	Assertion failure at tx.c:1678

Figure A.2: More information about the bugs that are found by Jaaru in PMDK benchmarks.

A.7 Experiment Customization

The experiment workflow can be customized to install and run everything on the local machine instead of the virtual machine. To set up everything locally, download *data/setup.sh* script from the https://github.com/uci-plrg/jaaru-vagrant repository in the *home* directory of your local machine and run the script after installing the dependencies.

A.8 Notes

Note that the performance results generated for RECIPE can be different from the numbers that are reported in the dissertation since there is non-determinism in scheduling threads; when stores, flushes, and fences leave the store buffer; and memory alignment in the *malloc* procedure. This non-determinism can possibly impact on the type of bugs reported in Figure A.1 and Figure A.2 for RECIPE and PMDK benchmarks. Also, for some bugs, the segmentation fault (or assertion failure) occurs in Jaaru code. This is caused by illegal memory access by the program under test.

Appendix B

PSan Artifact Evaluation

B.1 Abstract

This artifact contains a vagrant repository that downloads and compiles the source code for PSan (a plugin for Jaaru), its companion compiler pass, and benchmarks. The artifact enables users to reproduce the bugs that are found by PSan in PMDK and RECIPE (Table 4.1 in Chapter 4) as well as comparing bug-finding capabilities and performance of PSan with Jaaru, a persistent memory model checker (Table 4.2 in Chapter 4).

B.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Robustness violations detector
- **Program:** PSan
- **Compilation:** GCC 7.5.0 and Clang
- **Binary:** Instrumentation LLVM pass

- **Data set:** CCEH, FAST_FAIR, RECIPE, Redis, Memcached, and PMDK benchmarks
- **Run-time environment:** Any system that can run Vagrant
- **Hardware:** One 6 core 3.7 GHz Intel i7 machine with 32 GB DDR4 memory
- **Run-time state:** Managed by Jaaru's x86 simulator
- **Execution:** Automated by Jaaru's tooling system
- **Metrics:** Reporting robustness violations in program under test
- **Output:** Suggestions for robustness violations bugs. Logging performance measurement for executions.
- **Experiments:** Regenerating all bugs found by PSan. Reproducing performance results and comparing them with Jaaru (underlying open-source model checker)
- **How much disk space required (approximately)?:** 80G (For using our VM 200G)
- **How much time is needed to prepare workflow (approximately)?:** 4 hours
- **How much time is needed to complete experiments (approximately)?:** About 90 minutes
- **Publicly available?:** Yes. Open-source on GitHub
- **Code licenses (if publicly available)?:** GNU GENERAL PUBLIC LICENSE Version 2
- **Data licenses (if publicly available)?:** Lenovo, BSD-3-Clause, and Apache License 2.0.
- **Workflow framework used?:** Vagrant.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.6326792

B.3 Description

Our workflow has four primary parts: (1) creating a virtual machine and installing dependencies needed to reproduce our results, (2) downloading the source code of PSan and the benchmarks and building them, (3) providing the parameters corresponding to each bug to reproduce the bugs, and (4) Comparing bug-finding capabilities PSan with the Jaaru (The underlying model checker) on how automatically PSan suggests fixes found by Jaaru. After the experiment, the corresponding output files are generated for each bug.

To simplify the evaluation process, we created an instance of VM that includes all the source code and corresponding binary files. This VM is fully set up and it is available on Zenodo repository. This document also provides a guideline on how to setup the VM and use it to reproduce PSan's evaluation results.

How to Access

All source code is open-source and available on GitHub. Our packaging requires cloning the vagrant system repository from https://github.com/uci-plrg/psan-vagrant. As described in the *README.md* file of the repository, you will need to install a VirtualBox VM and Vagrant on your machine. Then, the vagrant setup will install the required dependencies and download the source code of the tools from our git repository. Next, it builds each tool on the virtual machine.

we created an instance of VM that includes all the source code and corresponding binary files. This VM is fully set up and it is available on Zenodo repository.

Hardware Dependencies

Our tooling system and PSan have no special hardware dependencies and it can be running on any x86 machine with at least 32GB RAM and 4 cores.

Using our VM: To properly import the pre-built VM instance, please verify you have enough storage on your disk ($\sim 200G$) and you used the most recent version of Vagrant ($>= 2.2.19$) to avoid facing any errors.

Software Dependencies

To run our system, the following should be installed on the local machine:

- Linux (we tested on Ubuntu)
- Vagrant
- VirtualBox
- Vagrant-disksize plugin

Also, In order for Vagrant to run, we should first make sure that the VT-d option for virtualization is enabled in BIOS.

Data Sets

To evaluate PSan, our tooling system downloads the source code of RECIPE, Redis, Memcached, and PMDK from our git repository. We forked a branch from the original source code of these benchmarks that don't contain our bug fixes. The tooling system automatically sets up and builds these benchmarks and runs them under PSan to identify bugs in them.

B.4 Installation

Please see the *README.md* file of the https://github.com/uci-plrg/psan-vagrant repository, which contains a detailed step-by-step guide to setup PSan on a virtual machine. Then, our scripts automatically do the following:

1. Install all the dependencies needed to install and evaluate PSan on different benchmarks.
2. Check out the source code for LLVM, PSan, PSan's LLVM pass, RECIPE, Redis, Memcached, and PMDK.
3. Include PSan's LLVM pass to LLVM and building it
4. Set up and build PSan with two different configurations (One for RECIPE that uses *libvmemmalloc*, and one for PMDK that uses *libpmem* APIs).
5. Set up and building RECIPE (including CCEH, FAST_FAIR, P-ART, P-BwTree, P-CLHT, and P-Masstree benchmarks), Redis, Memcached, and PMDK benchmarks.
6. Generate ten scripts in the *home (or ∼/)* directory of the virtual machine to generate the results.

Once the scripts are finished setting up the virtual machine and benchmarks, the user can use PSan on the virtual machine to further evaluate different benchmarks or regenerate our evaluation results.

If you are using our VM, you need to download it from Zenodo repository. Refer to *Step 3* of the Readme.md file for further details.

B.5 Experiment Workflow

After setting up the virtual machine, the user can use 'vagrant ssh' to connect to the VM and use PSan. The detailed instructions to run the suggested workflow is included in the *README.md* file of https://github.com/uci-plrg/psan-vagrant repository. There are seven scripts in the *home* directory of the virtual machine that user can run:

perf.sh : It runs PMDK, Redis, Memcached, and RECIPE benchmarks using PSan and gathers measurements to compare PSan against Jaaru. For each benchmark, the corresponding log file is generated in *~/results/performance*.

recipe-bugs.sh : It runs the RECIPE benchmarks using PSan and sets the corresponding parameters to reproduce each bug. For each benchmark, the corresponding log file is generated in *~/results/recipe-bugs*.

compare-jaaru.sh : It runs the RECIPE benchmarks using PSan and sets the corresponding parameters to reproduce each bug that Jaaru found. For each benchmark, the corresponding log file is generated in *~/results/recipe-jaaru-bugs*.

pmdk-bugs.sh : It runs PMDK benchmarks by using PSan and set the corresponding parameters to reproduce each bug. For each test case, the corresponding log file is generated in *~/results/pmdk-bugs*.

B.6 Evaluation and Expected Result

After successfully running the experiment using our scripts, the *results* directory is generated in the *home* directory. We created an online document that elaborate details of each bug and describe the fix suggestions by PSan. You can check out this file for investigating each bug

found by PSan and each bug fix. Regardless, the *results* directory contains the following results:

RECIPE Jaaru's Bugs

There are 14 files in ∼*/results/recipe-jaaru-bugs* directory with the pattern of *[BENCHMARK_NAME]-bug.log*. Each file contains the robustness violations found by PSan in each benchmark and corresponding suggestions for each of them. Figure B.1 contains information about how PSan reports bugs with * correspond to Table 4.1.

#	Bug ID	PSan Fix	Cause of Bug
1	CCEH-bug-1	src/CCEH_LSB.cpp:175:23	Missing flush for *dir._[i]*
2	CCEH-bug-2	src/CCEH_LSB.cpp:174:14	Missing flush for pointer dir._[i]
3	CCEH-bug-3	src/CCEH_LSB.cpp:172:1	Missing flush for CCEH object
4	FAST_FAIR-bug-1	btree.h:192:17	Missing flush for root of btree
5	FAST_FAIR-bug-2	btree.h:1860:10	Missing flush for btree object
6	P-ART-1	memset	Missing flush for deletionList
7	P-ART-2	Tree.cpp:23:61	Missing flush for Tree object
8	P-BwTree-Bug-2	bwtree.h:467:19	Missing flush for gc_metadata_p
8	P-BwTree-Bug-2	bwtree.h:467:19	Missing flush for gc_metadata_p
9	P-BwTree-Bug-3	src/bwtree.h:346:7	Missing flush for gc_metadata_p
10	P-BwTree-Bug-4	src/bwtree.h:2014:7	Missing flush for AllocationMeta object
11	P-BwTree-Bug-5	src/bwtree.h:3114:13	Missing flush for BwTree object
12	P-CLHT-Bug-1	src/clht_lf_res.c:220:21	Missing flush for clht_t object
13	P-CLHT-Bug-2	src/clht_lf_res.c:220:21	Missing flush for Hashtable object
14	P-CLHT-Bug-3	memset	Missing flush for table in hashtable object

Figure B.1: More information about the bugs found by PSan and Jaaru in CCEH, Fast_Fair, and RECIPE benchmarks. Number after ':' represents the line number.

RECIPE New Bugs

There are 18 files in ∼*/results/recipe-bugs* directory with the pattern of *[BENCHMARK_NAME]-races.log*. Each file contains the persistency races found by PSan in each benchmark. Figure B.2 contains information about how PSan reports bugs without * correspond to Table 4.1.

#	Bug ID	PSan Fix	Cause of Bug
1	CCEH-bug-1	src/CCEH_LSB.cpp:19:11	Missing flush for sema
2	CCEH-bug-3	src/CCEH_LSB.cpp:37:19	Missing flush for key
3	CCEH-bug-5	src/CCEH_LSB.cpp:51:11	Missing flush for sema
4	FAST_FAIR-bug-1	./btree.h:583:40	Missing flush for switch_counter
5	FAST_FAIR-bug-2	./btree.h:654:36	Missing flush for last_index
6	FAST_FAIR-bug-3	./btree.h:593:36	Header not fitting within the cacheline
7	FAST_FAIR-bug-4	./btree.h:603:49	Missing flush for ptr
8	P-ART-1	N.cpp:97:43	Missing flush for typeVersionLockObsolete
9	P-ART-2	N.cpp:108:37	Missing flush for typeVersionLockObsolete
10	P-ART-3	N.cpp:119:33	Missing flush for typeVersionLockObsolete
11	P-ART-4	N4.cpp:21:28	Missing flush for keys
12	P-ART-5	N4.cpp:25:32	Missing flush for children
13	P-ART-6	N16.cpp:12:28	Missing flush for keys
14	P-ART-Mem-Bugs	6 different locations	Epoch not properly persisted
15	P-ART-8	N16.cpp:19:14	Missing flush for count
16	P-ART-9	Epoche.cpp:66:22	Missing flush for nodeCount
17	P-BwTree-Bug-1	src/bwtree.h:2089:23	Missing flush for next
18	P-BwTree-Mem-Bugs	3 different locations	Missing flushes in memory management code

Figure B.2: More information about the bugs found by PSan in CCEH, Fast_Fair, and RECIPE benchmarks. Number after ':' represents the line number.

PMDK

There are 4 files in ~*/results/pmdk-bugs* directory. Figure B.3 corresponds to bugs in Table 4.1 in PMDK benchmark.

#	Bug ID	PSan Fix	Cause of Bug
1	PMDK-Bug-1	memcpy	Missing flushes after memcpy
2	PMDK-Bug-2	memcpy	Missing flushes after memcpy
3	PMDK-Bug-3	ulog.c:556:9	Missing flushes after write to dst
4	PMDK-Bug-4	memcpy	Missing flushes after memcpy

Figure B.3: More information about the bugs that are found by PSan in PMDK benchmarks. Number after ':' represents the line number.

Performance Results

Running *./perf.sh* generates *performance.out* file in *~/results/performance* directory. This file contains the performance information in Table 4.2 of the dissertation.

B.7 Experiment Customization

The experiment workflow can be customized to install and run everything on the local linux machine instead of the virtual machine. To set up everything locally, download *data/setup.sh* script from the https://github.com/uci-plrg/psan-vagrant repository in the *home* directory of your local machine and run the script after installing the dependencies.

B.8 Notes

Note that the performance results generated for the benchmarks can be different from the numbers that are reported in this dissertation since there is non-determinism in scheduling threads; when stores, flushes, and fences leave the store buffer; and memory alignment in the *malloc* procedure. This non-determinism can possibly impact on the number of bugs reported in Table 4.1 RECIPE and PMDK benchmarks.

Appendix C

Yashme Artifact Evaluation

C.1 Abstract

This artifact contains a vagrant repository that downloads and compiles the source code for Yashme, its companion compiler pass, and benchmarks. The artifact enables users to reproduce the bugs that are found by Yashme in PMDK, Memcached, and Redis (Table 5.3 of the dissertation), and RECIPE (Table 5.2) as well as the performance results to compare Yashme with Jaaru (Table 5.4).

C.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Persistency race detector
- **Program:** Yashme
- **Compilation:** GCC 7.5.0 and Clang
- **Binary:** Instrumentation LLVM pass

- **Data set:** RECIPE, Redis, Memcached, and PMDK benchmarks
- **Run-time environment:** Any system that can run Vagrant
- **Hardware:** One 6 core 3.7 GHz Intel i7 machine with 32 GB DDR4 memory
- **Run-time state:** Managed by Jaaru's x86 simulator
- **Execution:** Automated by Jaaru's tooling system
- **Metrics:** Reporting persistency race in program under test
- **Output:** Persistency race bugs. Logging performance measurement for executions.
- **Experiments:** Regenerating all bugs found by Yashme. Reproducing performance results and comparing them with Jaaru (underlying open-source model checker)
- **How much disk space required (approximately)?:** 80G
- **How much time is needed to prepare workflow (approximately)?:** 4 hours
- **How much time is needed to complete experiments (approximately)?:** About 90 minutes
- **Publicly available?:** Yes. Open-source on GitHub
- **Code licenses (if publicly available)?:** GNU GENERAL PUBLIC LICENSE Version 2
- **Data licenses (if publicly available)?:** Lenovo, BSD-3-Clause, and Apache License 2.0.
- **Workflow framework used?:** Vagrant.
- **Archived (provide DOI)?:** https://dl.acm.org/doi/10.1145/3462316

C.3 Description

Our workflow has four primary parts: (1) creating a virtual machine and installing dependencies needed to reproduce our results, (2) downloading the source code of Yashme and the benchmarks and building them, (3) providing the parameters corresponding to each bug to reproduce the bugs, and (4) running the benchmarks to compare Yashme with the Jaaru, the underlying model-checker. After the experiment, the corresponding output files are generated for each bug and each performance measurement.

How to Access

All source code is open-source and available on GitHub. Our packaging requires cloning the vagrant system repository from https://github.com/uci-plrg/pmrace-vagrant. As described in the *README.md* file of the repository, you will need to install a VirtualBox VM and Vagrant on your machine. Then, the vagrant setup will install the required dependencies and download the source code of the tools from our git repository. Next, it builds each tool on the virtual machine.

Hardware Dependencies

Our tooling system and Yashme have no special hardware dependencies and it can be running on any x86 machine with at least 32GB RAM and 4 cores.

Software Dependencies

To run our system, the following should be installed on the local machine:

- Linux (we tested on Ubuntu)
- Vagrant
- VirtualBox
- Vagrant-disksize plugin

Data Sets

To evaluate Yashme, our tooling system downloads the source code of RECIPE, Redis, Memcached, and PMDK from our git repository. We forked a branch from the original source code of these benchmarks that don't contain our bug fixes. The tooling system automatically sets up and builds these benchmarks and runs them under Yashme to identify bugs in them.

C.4 Installation

Please see the *README.md* file of the https://github.com/uci-plrg/pmrace-vagrant repository, which contains a detailed step-by-step guide to setup Yashme on a virtual machine. Then, our scripts automatically do the following:

1. Install all the dependencies needed to install and evaluate Yashme on different benchmarks.
2. Check out the source code for LLVM, Yashme, Yashme's LLVM pass, RECIPE, Redis, Memcached, and PMDK.
3. Include Yashme's LLVM pass to LLVM and building it
4. Set up and build Yashme with two different configurations (One for RECIPE that uses *libvmemmalloc*, and one for PMDK that uses *libpmem* APIs).

5. Set up and building RECIPE (including CCEH, FAST_FAIR, P-ART, P-BwTree, P-CLHT, and P-Masstree benchmarks), Redis, Memcached, and PMDK benchmarks.
6. Generate seven scripts in the *home (or ~/)* directory of the virtual machine to generate the results.

Once the scripts are finished setting up the virtual machine and benchmarks, the user can use Yashme on the virtual machine to further evaluate different benchmarks or regenerate our evaluation results.

C.5 Experiment Workflow

After setting up the virtual machine, the user can use 'vagrant ssh' to connect to the VM and use Yashme. The detailed instructions to run the suggested workflow is included in the *README.md* file of https://github.com/uci-plrg/pmrace-vagrant repository. There are seven scripts in the *home* directory of the virtual machine that user can run:

perf.sh : It runs PMDK, Redis, Memcached, and RECIPE benchmarks using Yashme and gathers measurements to compare Yashme against Jaaru. For each benchmark, the corresponding log file is generated in *~/results/performance*.

recipe-bugs.sh : It runs the RECIPE benchmarks using Yashme and sets the corresponding parameters to reproduce each bug. For each benchmark, the corresponding log file is generated in *~/results/recipe*.

pmdk-bugs.sh : It runs PMDK benchmarks by using Yashme and set the corresponding parameters to reproduce each bug. For each test case, the corresponding log file is generated in *~/results/pmdk*.

memcached-client.sh : It runs memcached client test script to execute scenarios to reproduce each bug in Memcached benchmark.

redis-client.sh : It runs Redis client test script to execute scenarios to reproduce each bug in Redis benchmark.

memcached-server.sh : It runs Memcached benchmark by using Yashme and set the corresponding parameters to reproduce each bug. The server script runs on one terminal, and the client script runs in another terminal. The persistency races are printed out on the server's terminal.

redis-server.sh : It runs Redis benchmark by using Yashme and set the corresponding parameters to reproduce each bug. The server script runs on one terminal, and the client script runs on another terminal. The persistency races are printed out on the server's terminal.

C.6 Evaluation and Expected Result

After successfully running the experiment using our scripts, the *results* directory is generated in the *home* directory. This directory contains the following results:

RECIPE

There are 7 files in ∼*/results/recipe* directory with the pattern of *[BENCHMARK_NAME]-races.log*. Each file contains the persistency races found by Yashme in each benchmark. Figure C.1 contains information about how Yashme reports each bug correspond to Table 5.2.

#	Bug ID	Cause of Bug
1	CCEH-1	Write to *value* in CCEH_LSB.cpp:29
2	CCEH-2	Write to *key* in CCEH_LSB.cpp:31
3	FAST_FAIR-1	Write to *last_index* in btree.h:643 and btree.h:831
4	FAST_FAIR-2	Write to *switch_counter* in btree.h:578 and btree.h:820
5	FAST_FAIR-3	Write to *key* in btree.h:584, btree.h:624, and btree.h:605
6	FAST_FAIR-4	Write to *ptr* in btree.h:585, btree.h:604, btree.h:625, and btree.h:828
7	FAST_FAIR-5	Write to *root* in btree.h:1857
8	FAST_FAIR-6	Write to *sibling_ptr* in btree.h:824
9	P-ART-1	Write to *compactCount* in N4.cpp:26 and N16.cpp:15
10	P-ART-2	Write to *count* in N4.cpp:27 and N16.cpp:16
11	P-ART-3	Write to *deletitionListCount* in Epoche.cpp:44
12	P-ART-4	Write to *headDeletionList* in Epoche.cpp:57
13	P-ART-5	Write to *nodesCount* in Epoche.cpp:60:22
14	P-ART-6	Write to *added* in Epoche.cpp:63
15	P-ART-7	Write to *thresholdCounter* in Epoche.cpp:78
16	P-BwTree-1	Write to *epoch* in bwtree.h:572
17	P-Masstree-1	Write to *root_* in masstree.h:1216
18	P-Masstree-2	Write to *permutation* in masstree.h:1318, masstree.h:1364, and masstree.h:1399
19	P-Masstree-3	Write to *next* in masstree.h:1162

Figure C.1: More information about the bugs found by Yashme in CCEH, Fast_Fair, and RECIPE benchmarks. Number after ':' represents the line number.

PMDK, Redis, Memcached

There are 2 files in ∼*/results/pmdk* directory. These two files contain the bug corresponding to bug #1 in Figure C.2. Bug #2 - #4 are printed in the terminal output after running *memcached-server.sh* and *memcached-client.sh* scripts. Figure C.2 corresponds to bugs in Table 5.3.

Performance Results

Running *./perf.sh* generates *performance.out* file in *~/results/performance* directory. This file contains the performance information as well as number of bugs found w/ or w/o prefix-based

#	Benchmark Found	Symptom
1	PMDK	Write to *ulog_entry* in ulog.c:561
2	Memcached	Write to *valid* in pslab.c:368
3	Memcached	Write to *id* in pslab.c:92
4	Memcached	Write to *it_flags* in slabs.c:543, items.c:519, and items.c:343
5	Memcached	Write to *cas* in memcached.c:4290 and items.c:538

Figure C.2: More information about the bugs that are found by Yashme in PMDK, Memcached and Redis benchmarks. Number after ':' represents the line number.

expansion algorithm corresponding to Table 5.4 of the dissertation. Note, since we manually deduplicate these bugs, the numbers of bugs in *performance.out* could be more than the number of bugs reported in this dissertation.

C.7 Experiment Customization

The experiment workflow can be customized to install and run everything on the local machine instead of the virtual machine. To set up everything locally, download *data/setup.sh* script from the https://github.com/uci-plrg/pmrace-vagrant repository in the *home* directory of your local machine and run the script after installing the dependencies.

C.8 Notes

Note that the performance results generated for the benchmarks can be different from the numbers that are reported in the dissertation since there is non-determinism in scheduling threads; when stores, flushes, and fences leave the store buffer; and memory alignment in the *malloc* procedure. This non-determinism can possibly impact on the number of bugs reported in Table 5.2 and Table 5.3 for RECIPE, Redis, Memcached, and PMDK benchmarks.