Automatic Testing of Persistent Memory Applications

Henrique Dominguez da Silva Nunes Fernandes

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. Miguel Matos

Examination Committee

Chairperson:
Supervisor:
Member of the Committee:

October 2022

Dedico esta tese ao meu avô com quem tive a honra de partilhar o nome. A sua confiança em mim fez-me chegar onde estou. Obrigado por tudo...

Acknowledgments

Gostaria antes de tudo mostrar a minha eterna gratidão ao Professor Miguel Matos por toda a ajuda e constante disponibilidade. A sua orientação e dedicação neste trabalho melhorou consideravelmente a experiência e a qualidade do mesmo. Agradeço também todo o apoio oferecido pelo João Gonçalves que me ajudou nos momentos mais sombrios deste trabalho.

Quero agradecer aos meus pais e ao meu irmão, que sempre estiveram lá para mim e me deram todas as condições para chegar onde cheguei.

Um especial obrigado ao Duarte, Henrique, Tomás, Manuel, Pedro, Ricardo e David por me terem suportado sempre ao longo destes anos e terem tornado o trabalho um pouco mais fácil com a sua presença e apoio.

Um grande obrigado também ao João, Duarte, Sofia, Duarte, Tiago, João e Rafael por todo o apoio ao longo do curso, pela companhia nas aulas, nos arraiais, no arco e nas casas do Manuel. Foram 5 anos excelentes graças a vocês.

Um obrigado bem fundo do coração à Mariana, que sempre acreditou em mim, me motiva a lutar pelo meu futuro e aceita as minhas ideias caras de experiências gastronómicas. Não podia pedir por uma melhor pessoa ao meu lado.

Este trabalho foi suportados por fundos nacionais pela FCT, Fundação para a Ciência e Tecnologia, através do projeto Ainur (PTDC/CCICOM/4485/2021).

Resumo

Memória persistente (PM) é uma nova tecnologia que tem performance equivalente a DRAM e oferece não-volatilidade. PM é endereçável ao byte e comunica diretamente com o processador, aumentado o desempenho quando comparado com discos.

Quando escrevemos em PM, a escrita é primeiro guardada na cache do CPU e só depois é persistida na PM. Na eventualidade de um crash ou falha de energia, qualquer escrita ainda na cache é perdida. Além disto, o hardware pode reordenar as instruções para aumentar o desempenho. Isto resulta na perda de garantias em relação ao estado da PM depois de um crash.

Foram desenvolvidas instruções de baixo nível para dar estas garantias ao programador. Uma instrução de flush ordena uma escrita para ir da cache do processador para PM. Um fence garante que qualquer escrita afetada por um flush é persistida a partir desse ponto. O uso destas instruções não é trivial, e o mau uso das mesmas leva a bugs.

Escrever aplicações de PM é difícil, logo devemos oferecer aos programadores uma ferramenta que lhes permita assegurar crash-consistency nas suas aplicações. Existem ferramentas no estado da arte, mas nenhuma oferece simultaneamente eficiência, automação e cobertura. Algumas demoram demasiado tempo a testar uma aplicação. Outras requerem que o programador recompile código e algumas testam apenas um conjunto limitado de aplicações.

Este trabalho apresenta uma ferramenta que oferece estas três características. Conseguimos isto da seguinte forma: i) criamos todos os estados possíveis numa única execução usando copy-on-write, ii) aplicamos heurísticas que reduzem a quantidade de estados a explorar e iii) paralelizamos o processo de recuperação, que permite alcançar speedup até sete vezes.

Palavras-chave: Memória persistente, crash-consistency, testagem, automação, paralelização

Abstract

Persistent memory (PM) is a new technology that performs similarly to DRAM and offers non-volatility. This byte-addressable memory communicates directly with the processor, increasing performance compared to disks.

When a store is issued to PM, it is first stored in the CPU cache and only then persisted to PM. Any stores in the cache will be lost in the event of a power failure or a crash. Moreover, the hardware can reorder the instructions to improve performance. These considerations result in the loss of guarantees about the state of PM after a crash.

Low-level instructions were designed to give these guarantees to the developer. A flush instruction orders a write to move from the CPU cache to PM. A fence guarantees all flushed stores are persisted from that point onward. Using these instructions is not trivial, and their misuse leads to bugs.

Writing PM applications is quite hard, and a tool must exist to help developers ensure crash-consistency in their applications. There are some available tools in the state of the art, but none simultaneously offer efficiency, automation, and completeness. Many take very long to test an application. Others require the user to recompile their code, and some test only specific applications.

This work presents a tool that offers these three characteristics. We achieve this by: i) generating all possible states in one execution with copy-on-write, ii) using heuristics that reduce the states to explore while maintaining completeness, and iii) parallelizing the recovery, which achieves speedups up to seven times.

Keywords: Persistent memory, crash-consistency, testing, automation, parallelization

Contents

Acknowledgments . . . v

Resumo . . . vii

Abstract . . . ix

List of Tables . . . xv

List of Figures . . . xvii

Nomenclature . . . 1

Glossary . . . 1

1 Introduction 1

2 Background 5

2.1 Persistent Memory . . . 5

2.2 Persistent Memory Development Kit . . . 8

2.3 Intel's Dynamic Binary Instrumentation Tool - PIN . . . 10

2.4 Copy-on-write (CoW) . . . 10

2.5 Types of Bugs . . . 11

2.5.1 Performance Bugs . . . 11

2.5.2 Correctness Bugs . . . 12

3 Related Work 13

3.1 Automation . . . 13

3.1.1 Yat . . . 14

3.1.2 Jaaru . . . 15

3.1.3 Witcher . . . 17

3.1.4 PMDebugger . . . 18

3.2 User Annotations . . . 20

3.2.1 PMTest . . . 21

3.2.2 XFDetector . . . 22

3.3 Bug Oracles 24
3.3.1 Agamotto 24
3.4 Discussion 26

4 Approach **31**
4.1 Design Considerations 32
4.2 Overview - how does the system work? 33
4.3 Coordinator 34
4.4 PIN tool 34
4.5 Snapshot creator 35
4.6 Shadow PM 35
4.7 Snapshot generation and heuristics 37
4.7.1 Heuristic 1 - Limit number of stores at segments 38
4.7.2 Heuristic 2 - Limit age for a store to persist 39
4.7.3 Heuristic 3 - Reordering restrictions with timestamps 40
4.8 Discussion 41

5 Implementation **43**
5.1 Components 43
5.1.1 PIN tool 44
5.1.2 Snapshot creator 46
5.1.3 Shadow PM 47
5.1.4 Recovery process 50
5.2 Configuration 52
5.2.1 Mandatory configuration 52
5.2.2 Additional features 52
5.3 Discussion 52

6 Evaluation **55**
6.1 System Configuration 55
6.2 Metrics 56
6.3 Tests and workloads 57
6.4 Ext4 with Dax versus xfs with CoW 57
6.4.1 Execution time 57
6.4.2 CPU usage 59
6.4.3 Memory usage 62

6.4.4 Discussion 62
6.5 Heuristics 63
6.5.1 Heuristic 1 - Limit number of stores between fences 63
6.5.2 Heuristic 2 - Limit age for a store to persist 67
6.5.3 Heuristic 3 - Reordering restrictions with timestamps 69
6.6 Parallelization of the recovery process 69
6.6.1 Normal test case 70
6.6.2 Taskset 71
6.6.3 Numactl 73
6.7 Recovery time estimation 74
6.8 Discussion 75

7 Conclusions **77**
7.1 Future Work 78

Bibliography **81**

List of Tables

3.1 Different tools and their characteristics. 27
3.2 Efficiency, automation, and completeness analysis of the state of the art. 28

4.1 Reorderings for a and b in Algorithm 6. 37

6.1 System configurations. 55
6.2 Average memory per snapshot in megabytes. 62
6.3 Snapshots generated with heuristic 1. 65
6.4 Snapshots generated with heuristic 2. 69
6.5 Recovery process speedup from different thread counts. 73
6.6 Estimation of recovery times. 75

List of Figures

2.1 Architecture of Persistent Memory 6

3.1 Bookkeeping Array of PMDebugger 19

4.1 Architecture of the debugging tool 33

4.2 Time diagram of the coordinator 34

5.1 Diagram of the program that creates copies. 47

5.2 Deleting all possible values but the one persisted. 48

5.3 Assigning the different snapshots to threads. 51

5.4 Example of a thread's assigned snapshots. 51

6.1 Recovery time with PMDK's skiplist 2000 operations with different threads . . . 58

6.2 CPU usage for different thread counts using PMDK's skiplist with 2000 operations. 60

6.3 CPU usage for different thread counts using PMDK's skiplist with 2000 operations for xfs with CoW (focus on user). 61

6.4 Number of written addresses at each segment. 63

6.5 Number of written addresses (> 8) at each segment. 65

6.6 Number of written addresses at each segment. 66

6.7 Age to persist up to 20 segments (programs with 1000 operations). 68

6.8 Age to persist up to 20 segments. 68

6.9 Time to recover for different thread counts - previous machine. 70

6.10 Time to recover for different thread counts using taskset and limiting xfs with CoW to 16 threads. 72

6.11 Time to recover for different thread counts using taskset 72

6.12 CPU usage for different thread counts using PMDK's skiplist with 2000 operations and taskset. 73

6.13 Time to recover for different thread counts using numactl 74

Chapter 1

Introduction

Persistent memory (PM) is an emerging technology that combines the performance and flexibility of DRAM with the non-volatility of disks and flash memory. This byte-addressable memory is available to programs through regular store and load instructions. It communicates directly with the processor via the memory bus, bypassing the kernel and increasing performance when compared to disks.

Persistent memory allows programs to be more efficient by taking advantage of this direct communication with the processor. However, stores do not persist immediately, as they are initially stored in the processor cache and only persist when the cache line is written back to PM. Modern processors have special instructions for developers to control this, such as memory flushes and fences. A memory flush flushes a write from a cache line to the PM. A memory fence guarantees that all flushed or non-temporal writes before it are persisted (*i.e.* in the write pending queue) before any writes issued after it (see Section 2 for more detail). Consequently, two or more writes to different cache lines between two memory fences are not guaranteed to be persisted in the order the programmer specified because the hardware has reordering mechanisms to improve efficiency, which may change the order the writes are persisted to memory.

Crash consistency is critical in the case of PM programs because the behaviour in the event of a crash or a power failure is not the same as that of a program using RAM. If the power fails, everything in the volatile memory is lost, and we automatically "rollback" to a correct state because the memory will be empty. With persistent memory, the writes are persisted up until a certain point of the execution, meaning that when we restart the program, the memory will have content in it. Let us consider an example where we want to write and persist a key-value pair. We persist the key first, but the value is not persisted because the program crashes due to a power outage. When the program restarts and the recovery code runs, it may attempt to check for the value associated with the key. The key has persisted, and everything seems correct, but since the value has not persisted, the key will be pointing to null or another value in that memory address. This inconsistency will cause unexpected behaviour, either leading the program to a crash due to a segmentation fault or towards an undesired execution. The origin of this bug is persisting the key before the value, which the developer must avoid.

These considerations create a need for persistent memory programs: crash consistency. This property ensures that in the event of a system failure due to a crash or a power failure, the system is restored into a consistent state, meaning no incorrect or inconsistent information is visible in memory.

Correctly using these instructions in a PM program is not always trivial, as one may forget, for instance, to flush a particular line, compromising the crash consistency of a program (e.g., in the event of a power failure, where the store would not be persisted), or unnecessarily flushing a line, decreasing the program's performance. As such, methods and tools to guarantee that we do not have bugs in PM programs are essential to safe-keep crash consistency and efficiency.

Writing correct PM programs is quite difficult and therefore several methods and tools have been proposed to detect bugs in PM programs. Ideally, we would like a tool to be fast and automatic while still finding all the bugs in a program, but this is precisely the tradeoff currently in the state of the art. Some tools search all the possible persistent memory states after a crash, finding all inconsistencies with a program for each explored state [LDK$^+$14, GXD21]. This technique is automatic but does not scale well, as it leads to state-space explosion, rendering its completion unfeasible in the proper time for large and complex applications. Some tools use annotations to overcome this[LWZ$^+$19, LSW$^+$20], allowing for the debugging to focus on a specific portion of the program or to increase the speed at which bugs are discovered through hints that the developers give in such annotations. These aspects increase efficiency, which is one desired property. Agamotto [NRSQ20] uses symbolic execution to explore different program states instead of executing the application. Agamotto requires that developers program additional custom oracles, which are then able to detect application-specific bugs. Although these techniques reduce the number of false negatives, custom oracles and annotations are error-prone, as incorrect annotations or oracles will negatively impact the efficacy of such tools while placing an extra burden on the developers.

Our solution uses the recovery process as the bug-finding oracle. We assume this recovery is correct, meaning that if a program recovers successfully from all possible states, it is bug-free. Otherwise, the application has a bug if it does not recover successfully from a particular state. We use this logic to create an automatic tool, as the developer does not need to recompile code or create any extra annotations or oracles. We demonstrate how it is impossible to replicate all possible states an application may reach due to a state-space explosion. To overcome this, we use a series of heuristics to prune certain states, allowing us to preserve bug coverage and efficiency. Our tool takes advantage of copy-on-write to create the relevant states that an application may reach in a very space-efficient manner. We only execute the application once to create all states. This allows us to parallelize the recovery, which can shorten testing time up to 7 times when compared to a linear recovery. Focusing on efficiency ensures our tool is able to finish testing an application in an acceptable timespan. This allows developers using PM to integrate our tool into their development pipelines, ensuring that their applications are bug-free.

This work presents an automatic tool that can find nearly all bugs in a program, within an acceptable time limit (we consider 12 hours), without putting extra effort into the developers' hands. Such a tool is an improvement over the state of the art, as it maximizes all three desired properties: bug coverage, efficiency, and automation.

The remainder of the document is structured as follows. §2 introduces the concepts of PM and the semantics of its instructions, as well as different types of bugs associated with PM. §3 presents a detailed look at the systems in the state of the art of PM debugging and compares them with this work. §4 explains the architecture of the tool developed in this work and its components. §4 gives an overall look at the architecture of the tool developed in this work. §5 explains in detail how each of these components was implemented. §6 provides an experimental evaluation of the tool. §7 concludes this document.

Chapter 2

Background

This chapter gives context on certain concepts and existing libraries and tools that directly relate to or impact our work's design choices. We structure this chapter as follows. Section 2.1 offers an overview of what PM is and the challenges of programming with it to create a crash-consistent PM application. Sections 2.2 and 2.3 explain the usefulness of two tools developed by Intel. The first is a series of libraries containing several functions which help developers work with PM. The second is a dynamic binary instrumentation tool we use in our solution. Section 2.4 introduces the concept of copy-on-write that we explore to increase efficiency. Section 2.5 closes this chapter with a description of the different types of bugs associated with PM.

2.1 Persistent Memory

Persistent memory (PM) or non-volatile memory (NVM) is a new hardware technology that offers speeds comparable to DRAM while also providing persistence. This kind of storage communicates directly with the memory controller, allowing software to work with in-memory data, working as both a working copy and a persistent store of such data.

PM is byte-addressable, and writes to it are not immediately made persistent. The writes are initially stored in the processor's cache, and only when a cacheline is evicted or explicitly flushed is the write persisted to PM. Persistent memory is a game-changing technology when everything runs smoothly, assuming there are no failures (*e.g.:* power failures, program crashes), but it comes with its challenges without these assumptions. Failures create two implications. First, we need to ensure data durability, meaning the writes must be persisted. Second, ordering constraints, ensuring that updates to PM become visible in the semantically correct order for the program.

Since DRAM presents no such challenges, we could apply the same assumptions to PM. We

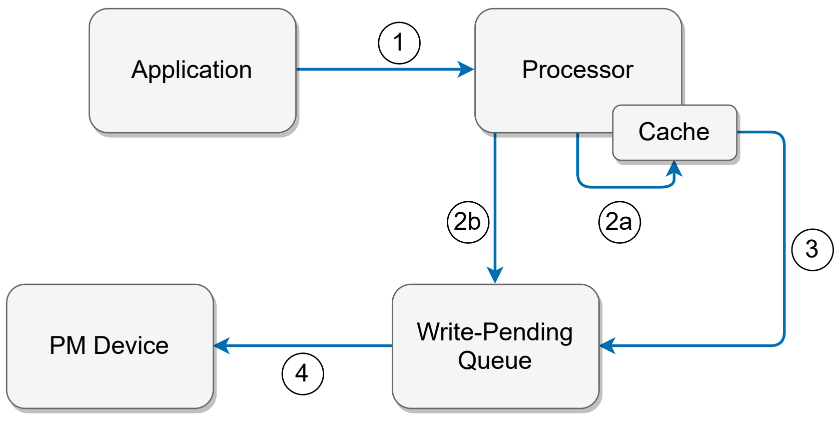


Figure 2.1: Architecture of Persistent Memory

cannot think of DRAM and PM the same way because the volatility of DRAM makes it so that in the event of a crash or a power failure, the program restarts with an empty memory. No writes in memory leave the program in a correct, even if undesired, state. With persistent memory, because writes are being persisted along with the program's execution, we have no guarantees about what exactly was persisted when the system fails, which may lead to bugs and data inconsistencies.

Creating PM programs and ensuring crash consistency is a hard task. For performance reasons, the hardware may reorder writes to memory, which increases the uncertainty one has about which writes have been persisted and in what order. There are certain instructions that help developers writing PM programs. These instructions enforce the order of persistence and the durability of stores. Let us focus on the two most important instructions: `flush` and `fence`. A `flush` instructs a cacheline to be written back to memory, adding it to the write pending queue [Rud17]. A `fence` instruction forces all writes in this queue to be persisted to memory. When we have a write, followed by `flush` and `fence`, we have the guarantee that after the `fence` instruction executes, the write has been successfully persisted to memory. Following Figure 2.1, $\textcircled{1}$ the application issues a store instruction to the processor, $\textcircled{2a}$ the processor stores the value in its cache, $\textcircled{3}$ the cacheline where the store was written to is flushed to the write-pending queue, and $\textcircled{4}$ the store is persisted to PM. A `flush` forces the action depicted in $\textcircled{3}$ and a `fence` forces $\textcircled{4}$. The flow $\textcircled{2b}$ corresponds to a non-temporal store, which is explained later in this section.

The concepts of flush and fence have various unique implementations in practice, each with its semantics. There is `mfence` and `sfence`, where the latter does not guarantee ordering of loads, only stores. Regarding flush operations, we should take the following into account: `clflush`, which flushes a line from the cache to the write-pending queue. `clflushopt`, which acts like `clflush` except that optimized flushes have no ordering guarantees between them for performance (still ordered between fences). `clwb`, which stands for cacheline write-back and acts like a `clflushopt` instruction, but does not invalidate the line from the cache and marks it as non-modified. Using `clwb` allows for a higher cache hit. If a developer decides to program with PM, they look for efficiency. Ideally, this will lead them to use optimized instructions such as `clflushopt` or `clwb` more than others. Regardless, different types of flushes will have their specific advantages depending on the semantics of the application.

On a separate note, there are also non-temporal stores (check 2b in Figure 2.1), which write directly to PM, but still require fence instructions for persistence and ordering guarantees.

To understand why programming with PM is hard and why these instructions are necessary, let us resort to Algorithm 1 to reason about the case of writing a key-value pair. We first write the key and persist it with `flush` and `fence`, and then write and persist the value using the same logic. Now let us suppose the program crashes right after line 5. In this case, the only guarantee we have is that the key has been persisted. When we recover the program, we may try to access the value through the key as in lines 7 and 8. Because the value is not guaranteed to have persisted, the behaviour from line 8 is undefined, and can lead to an attempt of accessing an invalid region of memory, crashing the program. We can therefore say that there is a data dependency between the key and the value and that to guarantee crash consistency, we need to make sure that the value is persisted before the key. This way, even if the key has not persisted, the program does not crash (assuming a check for the key before accessing its value), and we can execute the necessary instructions. Hence, the program continues its correct flow.

Algorithm 1: Example of a ordering bug with a key-value pair.

```
1 write key
2 flush key
3 fence
4 write value
5 flush value
6 fence
7 if (key):
8     a = array[value]
9 ...
```

There are already mechanisms to provide data consistency in system failures for file systems. One example is the journaling mechanism in Linux ext3 [PADAD05] as well as more recent implementations derived from it, such as the Linux ext4. Although the logic is the same when applied to PM, its nature does not allow for the application of the same techniques. To ease the burden on the developers, libraries like Intel's PMDK [Intb] offer transaction supports with atomicity guarantees [Int18, Int15]. Transactions avoid corruption due to system failures during updates to more complex structures. Developers using these libraries can abstract from the low-level instructions mentioned above and encapsulate updates between two library macros, `TX_BEGIN` and `TX_END`, and ensure the update of individual variables with `TX_ADD(var)` (see Section 2.2).

The specific case of the example mentioned above refers to atomicity, where a series of updates should collectively become visible to other processes. To better understand this, let us resort to Algorithm 2 which contains the pseudo code of a function that adds an element to a list and updates its size. Now let us suppose we have a crash anywhere inside this function before the `fence` in line 7 executes. In this case, the only guarantee we have is if `list.size` has persisted, then `list.last_element` has persisted too since the two stores are separated by a `flush` and `fence` pair. We may have the case where only `list.last_element` has persisted. In this last case, we incorrectly updated the list, as we now have an extra element but the size remains the same. This can lead to unexpetcted behavious later in the program execution and is considered an atomicity bug since both stores must be persisted as a whole.

Algorithm 2: Example of an atomicity bug.

```
1 function add_element_to_list(list, element) {
2     list.last_element = element
3     flush list.last_element
4     fence
5     list.size = list.size + 1
6     flush list.size
7     fence
8 }
```

Using persistence memory has the primary goal of increasing performance. Performance can also be affected by the misuse of PM instructions. Let us consider a fence instruction issued when all flushed stores are guaranteed to have reached the write pending queue. This particular fence adds no guarantees. It is only adding unnecessary overhead. As such, it makes sense to think of two types of bugs: correctness bugs and performance bugs. A correctness bug leads to data inconsistencies, like the case of persisting a key before the value. A performance bug is a misuse of PM instructions, like a redundant flush or fence (see §2.5).

2.2 Persistent Memory Development Kit

As mentioned earlier, Intel has developed a series of libraries to help developers work with persistent memory, called Persistent Memory Development Kit, or PMDK. These libraries contain several functions that abstract from the low-level hardware instructions introduced in the previous section. The most basic is `libpmem` which includes helper functions that tell the caller whether its CPU is compatible with certain PM instructions [Rud17].

PMDK has other libraries and one of the most relevant for developers is `libpmemobj`. As the name suggests, it manipulates objects in PM. It provides higher-level functions that guarantee the atomicity of updates to PM addresses through transactions.

Algorithm 3: Simplified example of PMDK functions.

```
1  struct coord {
2    double lat;
3    double lng;
4  }
5
6  struct coord *coordp = pmemobj_direct();
7  TX_BEGIN() {
8    TX_ADD(coordp);
9    $coordp- > lat = 38.73$;
10   $coordp- > lng = -9.14$;
11 } TX_END
```

Following Algorithm 3 we have an example with a structure with two attributes (lat and lng) that represents coordinates. To make sure we update the values in this structure atomically, we can use TX_BEGIN() and TX_END to encapsulate several changes in a transaction. The function in line 8 includes the allocated structure coordp in the transaction. In case the application crashes after executing the instruction in line 9, for instance, neither attribute in the structure updates, as the transaction is rolled back in its entirety. This is an example of how using transactions can eliminate the bug showcased in Algorithm 2.

Some of the other libraries included in PMDK are:

- The `libmemkind` library, which allows developers to use PM as volatile memory.
- The `librpmem` library, which allows remotely accessing PM.
- The `libvmalloc` library, which allows developers to migrate applications from disks to PM. This is done by converting memory allocations to persistent memory allocations.

Many of these libraries contain benchmarks that can be used to test their functionalities. In particular, we use many benchmarks from `libpmemobj` in our evaluation. These include programs that simulate a datastore with different data structures and crash-consistency mechanisms.

2.3 Intel's Dynamic Binary Instrumentation Tool - PIN

PIN [Int12] is a binary instrumentation framework that allows the manipulation of application execution and data collection regarding the same execution. PIN is dynamic because it does not require users to modify or recompile an application's source code to be able to instrument it. Instead, the instrumentation occurs during runtime while the application executes.

PIN instruments an application by injecting arbitrary code before or after any instruction. This code corresponds to the analysis routines, where gathering statistics or executing code is possible. PIN works through tools that contain instrumentation, callback, and analysis routines, which can be customized to suit the desired analysis. It is up to the developer to program each tool to do what they want, when they want, during the execution of the instrumented application.

PIN contributes to this work because it allows us to stop an application when we encounter particular instructions (*e.g.:* stores, flushes, and fences). When PIN halts the application, it can gather information about memory contents, which helps build our data structures to follow the persistent memory state. Additionally, this allows the creation of a copy of the memory mapped to an application, just like a snapshot. These snapshots simulate the state in which an application may be after a crash or power failure.

2.4 Copy-on-write (CoW)

We will introduce the concept of copy-on-write here as it is fundamental in this work and is later used to better explain the architecture in Chapter 4 and the implementation in Chapter 5.

Copy-on-write, or CoW, is a technique to share resources efficiently [BC03]. When a copy is created, it is not actually created until it is modified. In the context of this work, using copy-on-write means that when we create a copy with this technique, the space occupied in memory is only the difference between the original and the copy, not the copy fully. In the end, the result is having each copy occupying memory equivalent to the delta in memory since the last snapshot.

To better understand how copy-on-write is efficient, let us consider a large file of a hundred megabytes. If we create a regular copy of this file, we now have two files, each with a hundred megabytes. However, if we create a copy of this file using copy-on-write, we will only occupy a hundred megabytes in memory instead of two hundred with the previous approach. If we change a bit in the copy, we will occupy a hundred megabytes plus the block size. This comes from the fact that the block containing the changed bit can no longer point to the original file, as it is different. The filesystem creates a new pointer to a different block, which contains the difference between the original and the copy. The size occupied in memory by the copy will always be a multiple of the block size. This multiple depends on the number of data blocks that differ in each file.

2.5 Types of Bugs

To correctly program with persistent memory, one should be aware of how its persistence can also bring challenges. We can misuse PM in several ways, either affecting performance or correctness. Section 2.1 describes an example of the order in which two variables are persisted in memory, corresponding to a type of correctness bug. We will expand this notion of PM misuse and present other scenarios of bugs, starting with performance bugs and finishing with correctness bugs.

2.5.1 Performance Bugs

We consider performance bugs to be any misuse of PM that only affects the efficiency and not the semantics of a program. To ensure the persistence of a store, the programmer must issue a flush and fence. This is not always trivial in more complex applications, and we may have cases where a store is either flushed or fenced unnecessarily. We call these types of bugs redundant flush and redundant fence, respectively.

A redundant flush happens when a flush is issued on an address that was not overwritten since it was last flushed. Let us look at Algorithm 4 to consider another scenario for a redundant flush, where `a` and `b` correspond to variables mapped to addresses in the same cacheline. In this case, the flush in line 6 is redundant as the `flush` in line 5 already flushes the cacheline, acting on both `a` and `b`.

Algorithm 4: Example of a redundant flush in PM.

```
1 ...
2 fence
3 a = 1
4 b = 2
5 flush a
6 flush b
7 fence
8 ...
```

A redundant fence is observed when a fence is issued while no addresses have been flushed to the write-pending queue.

There is one last performance bug that consists in using PM to store transient data. Although this bug is hard to detect and highly dependent on application semantics, there are specific heuristics that may find it through static analysis of an application's code.

2.5.2 Correctness Bugs

A correctness bug is any bug that affects the semantics of a program. It can be related to ordering, durability, and atomicity.

To explain ordering bugs, we can consider the example presented in Section 2.1 with Algorithm 1, where the order in which we persist a key-value pair may affect how the program recovers after a crash. Following this logic, an ordering bug is any series of stores that are persisted in a way that the application may not be able to recover successfully after a crash. This type of bug is entirely dependent on application semantics and is not trivial to detect.

A durability bug removes the guarantee that certain data has persisted. For data to be considered persisted, it must be followed by a set of flush and/or fences. A durability bug includes the cases where a store is either not flushed or not fenced, resulting in the loss of guarantee that it has persisted. There are exceptions to this, such as a non-temporal store that writes directly to PM and does not need to be flushed. A store to a memory address that contains another store (which was not persisted) is considered a dirty overwrite and a violation of the durability guarantee. This is because there is no guarantee about what value may reside in that address after a crash.

Atomicity refers to a set of stores that should be treated with all-or-nothing semantics. This is usually the case with updates to more complex data structures. We give an example of this in Section 2.1. These mechanisms exist in other contexts in the form of, for example, undo logs or journaling. Section 2.2 introduces the concept of transactions within PMDK, which were created to prevent this type of bug. By using transactions, the programmer ensures that a set of stores is persisted as a whole.

Chapter 3

Related Work

The considerations presented in the previous chapter make it hard for a developer to create a bug-free and crash-consistent PM program, and if debugging programs is, in general, cumbersome work, debugging PM programs that use many low-level instructions is a challenging task. As such, a tool that detects these bugs and reports them is crucial to the success of PM programs. Ideally, such debugging tool should be automatic, efficient, and comprehensive, meaning that it should not require input from the developer, have high performance, and find all the bugs in a program.

We will focus on the first requirement, automation, and start by mentioning Yat [LDK$^+$14], a fully-automated tool that was developed in the early stages of the emergence of persistent memory. Yat is a good starting point to understand the tradeoffs in the state of the art of PM debugging. This section will then move on to other tools that also focus on automation through different approaches [GXD21, FKP$^+$21, DLCL21], and discuss the strengths and weaknesses of each approach. Later, works that make use of user annotations [LWZ$^+$19, LSW$^+$20] will be mentioned to show how the drawback of having extra developer work can result in increased performance and completeness. Finally, a tool using custom bug oracles [NRSQ20] will be presented for comparison.

3.1 Automation

Following this idea, an eager way to think about how to design an automatic debugging tool is simple. One can simulate all the possible combinations of writes between any two fences and check for any inconsistencies in the memory state. Yat [LDK$^+$14] is a tool that tries to refine this idea, resulting in a fully-automated validation framework for PM programs.

3.1.1 Yat

Yat [LDK+14] is a tool for developers to validate that programs use PM instructions correctly, ensuring that software is crash-consistent. The key idea behind Yat is simple: track updates to PM, simulate those updates in PM-based storage, and, for each possible memory state associated with those updates, test the application's recovery code and check if the program returns to a consistent state. If it does, then the program is crash-consistent, and if it does not, then it means we have a bug in the segment of updates to PM that Yat is currently testing.

To achieve this, Yat has two phases: record and replay. The first phase logs a trace of all PM-modifying instructions executed in the program. It also stores the type of operation and the memory region being affected. Because specific calls that must be tracked are made in kernel mode, Yat resorts to a specialized virtual machine monitor (VMM) called Hypersim, which can visualize a guest's behaviour to trap accesses to PM. VM exits occur when certain illegal instructions need the VMM to take control (*e.g.:* a write that causes a page fault), which are necessary for the VMM to log the VM's behaviour. Because certain PM instructions do not cause VM exits, Yat has to recompile the PM program in a way that "tricks" the VMM into looking at the VM exit and logging the PM instructions.

The second phase, replay, divides the trace collected in the first phase into segments. Each segment is a set of writes separated by a barrier that guarantees the persistence of all writes before it. Because the segment does not include the barrier, the writes are not guaranteed to have persisted. This barrier is a primitive that does not exist in practice, but we can think of it as a combination of flush and fence instructions for simplicity purposes. Yat creates all possible reorderings of writes in each segment and, inside each segment, all possible subsets of writes to simulate different crashes. Following this, it successively applies each subset of writes to the PM storage used for the simulation. Yat uses copy-on-write for this, which can quickly restore the initial state of the program after each test. It then runs the recovery code to restore the execution of the program, and it runs a data integrity checker, which will report any inconsistencies in the persisted data, along with the point in the trace of the program where this occurred, and the critical writes that caused the inconsistency.

Yat is fully-automated. It does not require any extra input from the developer, which is a desirable feature. However, this is not sufficient to balance its biggest drawback: performance. Because of the exhaustive nature of Yat, it scales superlinearly with the number of writes in the program, more specifically with the number of writes inside each barrier. This second part is important because we could have three writes separated by a barrier (write(A); barrier; write(B); barrier; write(C)) and in this case, there are only two possible reorderings in each segment, one where the write persists and another where it does not. Another scenario is if we have all three writes inside a segment (write(A); write(B); write(C); barrier), which may have up to 16 reorderings if all three writes are flushed with `flushopt`.

Moreover, it also scales with the complexity of the recovery process since it runs this process for every possible reordering, making Yat unpractical for more extensive or complex programs. For this reason, Yat allows users to set a threshold t for the max number of reorderings explored for each segment. This works as follows; Yat first determines how many reorderings are possible and, if this number is greater than t, then it only tests with t reorderings chosen at random [LDK+14]. This threshold is a tradeoff between performance and bug coverage since we may skip reorderings that would lead to an inconsistent state, ultimately leaving a bug to be reported.

Yat uses copy-on-write, which is an excellent strategy to roll back any changes to the simulated PM when testing. This technique increases efficiency and may allow running different reorderings in parallel. Parallelizing would increase the tool's efficiency, but because Yat's reordering approach already explodes the state-space, running a demanding program on parallel requires more computing power, which might not be available to the developer. This consideration highlights the relevance of having an efficient tool without Yat's overhead from testing too many different memory states.

Yat's authors evaluate it with thresholds but give estimates to compare with cases where such thresholds would not exist. It is clear that Yat does not scale: for programs with tens of thousands of writes, it would take years to test all possible reorderings [LDK+14], which is not practical or useful. Even with thresholds, it would take three days for one of the programs tested [LDK+14], but such a big difference in time means that Yat only explores a fraction of the possible states and, as such, is not providing enough completeness. Even if we consider three days for the tool to run all possible reorderings, this timeframe would not be ideal for a DevOps pipeline. In this case, the tool would run overnight for a maximum period of twelve hours (half a day). Another downside of Yat is that it was designed specifically for persistent memory file systems (PMFS). As such, Yat's integrity checking phase makes sense for this kind of application, but it may not be portable to other application semantics.

3.1.2 Jaaru

Let us now take a look at Jaaru [GXD21], which builds on the idea of Yat to create a fully-automated tool with increased efficiency. The main problem with Yat is that it does not scale since the number of states to explore becomes too big. Jaaru tries to address this issue by using a technique called constraint refinement. This technique allows for a reduction in the state space, reducing the time it takes for this tool to test a PM program effectively.

The first improvement that Jaaru has is reducing the number of failure injections. It assumes that the only failure injection points that may lead to an inconsistent state are right before a flush instruction.

Jaaru takes advantage of the commit store pattern, where the persistence of a write guarantees the persistence of another. Such is the case of the correct use of the example in 2 regarding a key-value store: if the key has persisted, so has the value. When Jaaru explores a post-crash execution, it will only consider the reads of PM addresses that were written to in the corresponding pre-crash execution. This means that if a variable is written in a pre-crash execution, but its value is never read post-crash, then the write is irrelevant. Ignoring these cases further reduces the number of states the program needs to explore.

This technique combined with the use of the commit store pattern allows Jaaru to reduce its complexity from $O(2^n)$, where n is the number of modified cache lines, to $O(m^2)$, where m is the length of the execution [GXD21]. This limit comes from the fact that in the worst case, we inject a failure at every line, and consequently, for each failure, we may have to execute at most m lines, which is the whole program.

Jaaru uses an LLVM compiler that instruments the PM program and traces the executed PM operations. Along with these instructions, intervals are stored for each flushed cache line to know when the most recent write to each cache line was made. Storing intervals allow cross-checking the persistence time of different writes in the same cache lines, which constraint refinement does.

When testing the PM program for inconsistencies, Jaaru simulates the PM instructions with software, which allows it to track the state of persistent memory. It also focuses on the TSO persistency semantics, meaning that the order in which store and flush operations appear in memory is the same in which they were issued by a processor [Ora]. For such, Jaaru includes the implementation of store and flush buffers for each thread in its PM simulation. For every execution that it tests, Jaaru keeps a stack of the executed instructions. It then uses this stack to report the origin of a bug in case of a failure. Jaaru implements this simulation with the exclusive full support of the x86 architecture. This choice is limiting if we want to test PM programs in machines with different architectures.

Because of its exhaustive testing nature, the authors of Jaaru [GXD21] suggest that it is better suited to test smaller code that has a higher impact (*e.g.:* library code), instead of more complex programs with specific purposes that may present a higher challenge for Jaaru to test efficiently. It is also worth mentioning that Jaaru's constraint refinement is a technique that has its advantage taken down when the developers do not use the commit store pattern, as this is the only aspect Jaaru explores to reduce the number of states. By reducing the number of failure injection points and applying constraint refinement where possible, Jaaru can reduce the states it explores by several orders of magnitude compared to Yat.

3.1.3 Witcher

Yat and Jaaru are two systems with a similar brute-force, exhaustive approach. We will now analyze Witcher [FKP$^+$21], which is still a fully-automated system but selectively chooses memory states that are more likely to lead to data inconsistencies. This selection is based on likely-correctness conditions, which follow inference rules based on data dependency hints between two variables in a PM program. Memory images of PM that violate these rules are then generated and used for testing. These are called crash images. Finally, output equivalence checking compares the output of the program in normal conditions (*i.e.* no crashes) with the output of the program running with the crash images.

Systematic pruning memory states based on these inference rules increases performance since we have a smaller exploration state space. This technique also leads Witcher to find more application-specific bugs, even without knowing the application. When other tools resort to developer annotations to become more comprehensive (*e.g.:* XFDetector [LSW$^+$20], PMDebugger [DLCL21]), this is a great technique to achieve that same level of comprehension without sacrificing performance or automation.

Like Jaaru, Witcher uses an LLVM compiler to instrument code and trace the application's PM-related instructions. It also stores the address, and, depending on the instruction, the size, and data (for stores). Witcher then performs program dependence analysis to create a Persistence Program Dependence Graph (PPDG). This directed graph contains all the control and data dependencies between different variables in persistent memory for the context of the program.

If we return to the running example of a key-value store, there is a control dependency between the value and the key since the value can only be accessed if the key has persisted. This dependency allows us to infer that any memory state where only the key has persisted results in a bug. To find these memory states, Witcher will run the program following the same trace of execution used to collect information for the PPDG. It will then cross-check each state for any violation of the likely-correctness conditions. When it finds a memory state that violates one of these conditions, it creates a crash image to be tested.

Since a fence instruction changes the state of memory persistence, Witcher only checks for violations of likely-correctness conditions right before a fence. Thus, Witcher identifies a crash image by a fence ID and a set of writes after the last fence. When testing with a created crash image, Witcher will check the program's output with said image and compare it to that of the program if no crash had happened. If the outputs do not match, then we have a data inconsistency since the program's execution with and without a crash is different, ultimately meaning it is not crash-consistent.

Different oracles exist to account for the cases where the program crashes, and a value may or may not have persisted. This means that output equivalence checking considers these different cases, and a bug exists only if none of the cases matches the expected output. Let us consider the following section of a program: we write the value 2 to X, and then we flush the cache line that contains X. If, in our test case, we crash the program just before the flush, we have no guarantees that the value 2 for X has persisted. When we perform output equivalence checking, both the values 2 and null are correct for X.

There are two considerations to make regarding output equivalence checking. First, if the test case results in no output or a visible symptom[1], then Witcher is unable to detect a bug. Second, due to the nature of this technique, if the output is different, then we are guaranteed to have a bug, which means that Witcher does not produce false positives [FKP+21].

Although Witcher bases its pruning on application-agnostic rules, it is designed explicitly with key-value stores in mind, which may not be as suitable for other types of programs. Since its bug detection is based on output equivalence checking, it must perform a load to get a value. Particular application semantics may have other requirements that a simple load and compare cannot test. Applications that change with the system's time or have their results rely on probabilities, such as artificial intelligence, may be hard to test with this tool.

3.1.4 PMDebugger

Although PMDebugger [DLCL21] is not fully-automated, we will still mention it here because it is a good bridge between automation and user annotations. As we will see in more detail, PMDebugger only evaluates the information regarding PM that it keeps in its bookkeeping structures. PMDebugger is not fully-automated because if a developer needs to ensure a specific persistence ordering between two variables, it has to provide PMDebugger with a configuration file. Although the paper mentions a configuration file, in reality, PMDebugger uses annotations. These annotations need to give PMDebugger a hint about the order in which different variables must persist in a given function. The extra work needed by the developer is what excludes PMDebugger from being fully automated.

[1]A visible symptom is any non-silent manifestation at the end of the execution (*e.g.:* output printing, segmentation fault)

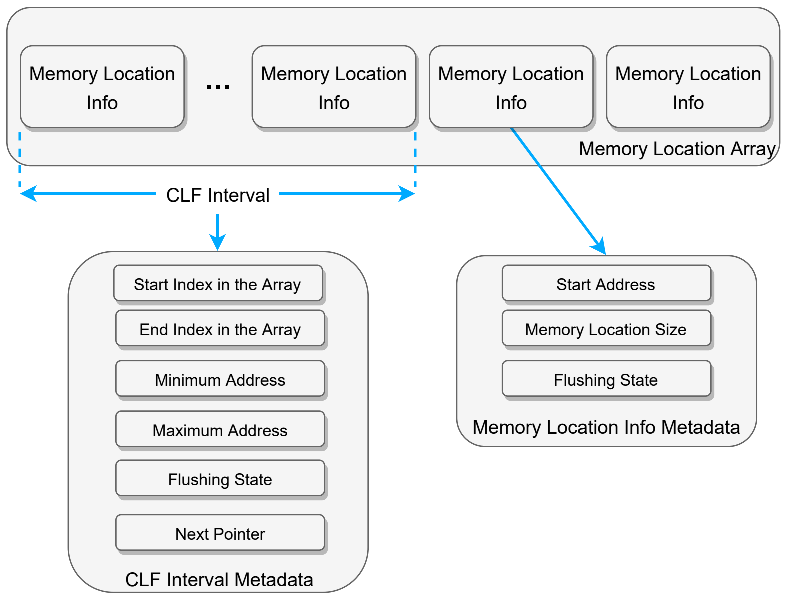


Figure 3.1: Bookkeeping Array of PMDebugger

After analyzing several executions of PM programs, the authors of PMDebugger reached the following conclusions: most stores are persisted with the first fence instruction that follows them. If a group of stores is between two flush instructions, they are likely to be collectively flushed by the same flush instruction. Store instructions are more frequent than flushes and fences.

These observations led to focusing on the efficiency of keeping the information collected when tracing a PM program. Each above observation justifies each of the following design decisions, respectively: Because most stores are quickly persisted, tree structures may not be the best because of the cost of tree reorganization. If most stores are collectively flushed, keeping information about their state of persistence collectively can improve performance. Since stores are the most common operation, efficiently processing these instructions is critical.

PMDebugger instruments store, flush, and fence instructions, splitting them into intervals delineated by fences, which are the instructions responsible for changing the persistency state. PMDebugger uses a combination of an array and a tree structure to store the information. The array is used for the most recent updates to PM because neither its insertion nor deletion has overhead. The tree is used for stores that take longer to persist. Using a tree grants the benefit of fast search, while reorganization is not very costly, since most programs persist stores quickly [DLCL21]. Unless many stores never persist, the tree will not grow too much.

The array, represented in Figure 3.1, contains elements with the address, size, and flushing state of each store, and elements in the array are grouped in CLF intervals. Flush instructions delineate these intervals. Each CLF interval has metadata such as the start and end index that compose such interval in the array of stores. The minimum and maximum addresses of these stores are also kept, as well as the flushing state of the CLF interval (which can be *not*, *partially*, or *all* flushed), and a pointer to the next CLF interval metadata structure. To understand how a CLF interval can be partially flushed, let us resort to Figure 3.1. If the CLF interval depicted has at least one memory location whose state is flushed and another whose state is not flushed, then the interval's state is deemed as partially flushed.

Processing each instruction means adding, deleting, or updating entries in the array and the tree. The key idea of a store instruction is to add an entry to the array with the store address and size and set the flushing state to unflushed. Then update the metadata for the CLF interval of this store, specifically the end index and maximum address. Processing a flush instruction means finding the CLF interval that contains the address being flushed and changing the corresponding CLF's flushing state accordingly. This may become a costly operation if the address range being flushed only partially overlaps a CLF interval. In this case, the unflushed elements in this range are moved to the tree. Finally, in short, fence instructions delete entries from the array or tree if their elements are flushed.

Finding a bug with this approach consists of looking at the information in the array and tree and drawing conclusions. For example, if PMDebugger finds that an entry exists in the array at the end of execution, it means that a particular memory location is missing a flush or a fence. If its state is not flushed, it is missing a flush and, if the state is flushed, it is missing a fence. Another example regards processing a store instruction. If the memory location already exists in the array, it means the program is trying to write to a location before guaranteeing the persistence of the last write to it.

These bug patterns are model-agnostic, meaning they work for more relaxed models, resulting in a more flexible application of this tool. As mentioned at the beginning of this section, PMDebugger does not know the application and is not able to check for persistence ordering [DLCL21]. This means that it needs developer annotations, stating the order variables should persist for each function.

3.2 User Annotations

So far, we have looked at systems that require little to no effort from the developer by being automated. We will now analyze systems that resort to user annotations, sacrificing automation for better performance and comprehension. Although user annotations may increase comprehension by helping tools find more application-specific bugs compared to fully automatic systems, they are error-prone. Misusing these annotations may result in false negatives, leaving the developer with a false sense of security when their software may not be crash-consistent. False positives can also be an issue, should the developer annotate a variable that is intended to be volatile. This case would cause the tool to flag the variable as not guaranteed to be persisted.

On top of this, many of the tools that use annotations have support for custom annotations for more specific program flows [LWZ+19, LSW+20], claiming that they are easy to extend for more comprehension and flexibility. Writing these annotations is not always trivial since it can pose a complex challenge, which can result in incorrectly configured test cases. With this consideration, the main tradeoff in the state of the art is finding a correct balance between automation and user input while not sacrificing performance or comprehension.

3.2.1 PMTest

The first tool we will analyze that uses annotations is PMTest [LWZ+19]. The main idea of PMTest is having the developer annotate the code with checkers of various levels and nature. The two low-level universal checkers it provides are `isPersist A` and `isOrderedBefore A B`. The former checks if A has persisted and the latter if A has persisted before B. These two are generally enough to assure we have the two guarantees we need in a crash-consistent PM program: persistence and ordering.

PMTest tracks PM operations at runtime and keeps relevant information in a shadow PM. For each flush or fence instruction that affects a store, PMTest will infer an interval for the flushing and persistence, respectively, of said store. For example, if we have a store `write A`, its flushing interval will be $(global\_timestamp, \infty)$. If we then flush it with `clwb A`, the flushing interval changes to $(global\_timestamp, global\_timestamp + 1)$. Note that the global timestamp is a positive number that increments with each operation.

Checking if certain stores follow the rules dictated by the annotations means checking the intervals for the memory addresses. If we want to check whether A has persisted, we need to look at the end of its persist interval. It has persisted if the interval ends in any finite number smaller than the current global timestamp. Checking if A has persisted before B means checking two things: the persist intervals of A and B do not overlap, and if the end of interval A is smaller than, or at most equal to, the beginning of interval B.

PMTest affirms to be both fast and flexible. Its speed comes from PMTest decoupling the execution engine from the checking engine. The execution collects traces of the program and sends them to the checking engine. The checking engine has a master thread that receives these traces and dispatches them to worker threads, sending the result back to a queue in the master thread. This flow allows the program to execute and be tested simultaneously.

PMTest can be used or adapted to different libraries and systems, hence its flexibility. The tracking system can be extended to be sensitive to different PM instructions and work with different PM models. Also, custom checkers can be developed to suit any specific needs an application may have. Although granting extensibility, it is not trivial in practice because it is hard to correctly add new checkers or behaviour to the tracking and checking engines. This puts the quality of testing of this tool in the developer's hands, meaning that if errors exist in the custom checkers or rules, we may get false positives or negatives.

PMTest offers two high-level checkers that check for the correct operation within PMDK transactions to be more user-friendly. It automatically injects the low-level checkers to check for the persistence of stores inside a transaction. Custom high-level checkers are also possible, allowing more experienced programmers to create them so that programmers with less knowledge of the code can take a black-box approach and quickly test their software.

Because PMTest does not simulate crashes and tries to infer the state of the PM at any given point in the execution, it does not know what a post-failure execution implies. This can lead to false positives, such as the following case: suppose we have a store `write A` followed by the checker `isPersist A`; this is going to be warned by PMTest because there is no guarantee that A has persisted; however, if A is not loaded in the future, we do not need to guarantee the persistence of it, so we do not have a bug. This is one of the cases that Jaaru avoided through its constraint refinement technique.

PMTest allows for more annotations besides the ones regarding checkers. One can select where the testing begins and ends in the code, which focuses on a specific portion of a program. Let us think about a file with hundreds of lines of codes of a complex program. Imagine we are working on a function comprising only tens of lines. PMTest allows selecting the testing region, which is very good for exhaustively testing a specific part of the program without having the overhead of testing the whole program. One can also add or remove specific variables to the testing scope. These possibilities add value and flexibility to this tool and give the developer more power. However, they also create more room for error.

3.2.2 XFDetector

The following tool we will analyze, and one that has many similarities with PMTest, is XFDetector [LSW$^+$20]. XFDetector stands for Cross-Failure Detector and validates PM crash consistency by detecting inconsistencies across both the pre- and post-failure stages. The general idea behind this system is having the developer annotate the code, which is then used by XFDetector to instrument the program to collect a trace and create a shadow PM. For each failure point, it resorts to the shadow PM to check the execution traces between the pre- and post-failure executions and, using the logic of XFDetector, find and report any inconsistencies.

The two main types of bugs this tool detects are cross-failure race and cross-failure semantic bugs. Each is defined as follows.

Cross-Failure Race Bug: Post-failure execution reads data that is not guaranteed to have persisted in the pre-failure execution. For example: if we have the operations {*write A; persist A; write B; fence;*} and the crash happens just after *write B*, the post-execution may try to read B, which is not guaranteed to have persisted, resulting in this kind of bug.

Cross-Failure Semantic Bug: Post-failure execution accesses stores that were persisted during the pre-failure execution but are semantically inconsistent according to the program. An example of this is a badly persisted key-value store, where the key has persisted, but the value has not, which causes the read to return null or an incorrect value.

To perform instrumentation and keep track of the state of PM, XFDetector uses Intel's Pin [Int12] tool to trace PM instructions such as writes, flushes, fences, and library function calls. These instructions update the state of a shadow PM, which is used to check the consistency of the persisted data. For each PM address, we have one possible state out of the following: unmodified, modified, *wb-pending* (writeback-pending, meaning either in or on the way to the write pending queue), and persisted (guaranteed to be in the write pending queue or the PM). Based on this, XFDetector can report bugs by looking at the persistency state of an address that is being accessed. For example, if the post-execution tries to read a PM address whose state is modified, it has no guarantee that it persisted, which is reported as a cross-failure race bug.

XFDetector injects failure points before ordering points. An ordering point is a sequence of instructions (or a single one) that changes the state of PM. A persist barrier is used as an example, consisting of a flush and a fence combination. A failure is thus injected before the combination of these two instructions, or just a fence, in case there is no preceding flush. It can also be injected for high-level calls that force write-backs such as the `TX_ADD()` in PMDK [Intd]. XFDetector does not inject a failure between two ordering points with no PM modifying instructions between them. This means that, for example, two consecutive fences will not have a failure injected between them.

For every simulated failure, XFDetector needs to run the post-failure execution, which includes the recovery code. The complexity of this process is limited by $O(F \cdot P)$, where F is the number of failure points and P is the average length of the post-failure execution. Just like PMTest [LWZ$^+$19], XFDetector offers annotations that allow the developer to limit the region of testing. Such a region is denoted as Region of Interest (RoI). Selecting a specific RoI can be advantageous to test only a particular part of the program (*e.g.:* the part in development). At the same time, if we incorrectly limit the RoI, we may leave out a part of the program containing bugs related to the part being tested.

If the post-execution has multiple reads to the same address in PM that would result in a cross-failure bug, XFDetector only reports the first one. This is a good optimization, as one report is enough to know what is to fix, and fixing that bug will most likely make all the reads (the reported one and the following) consistent.

Because the tracing mechanism is decoupled from the detection, XFDetector claims extensibility in both frontend and backend. Since Pin [Int12] is limited to user-level programs, other tools could be used to support kernel-level. Additionally, the developer can create extra annotations to support other PM libraries. Although extra work for the developer is never desirable, the room for extension is positive.

3.3 Bug Oracles

There is one system with a different approach than the previously discussed ones. Instead of relying on user annotations, it uses bug oracles to update and track the state of PM. It allows the creation of custom bug oracles so that this tool can be extended to test any PM software and find application-specific bugs. Given this, it could be included in the automated group of tools. This document mentions it separately because it is not a complete testing tool for a PM program with specific needs without the custom bug oracles. The same could be said for PMTest and XFDetector, but these two systems necessarily require user annotations, even if to limit the region of testing. We decide to discuss it after introducing these two systems because one can now reason about the extra work of developing custom oracles.

3.3.1 Agamotto

Agamotto [NRSQ20] takes a different approach from the systems previously discussed as it resorts to symbolic execution and bug-finding oracles. It has three main components: a PM model which executes and tracks instructions, the PM bug oracles that check the updates to PM and find bugs, and a PM-aware search algorithm that steers the symbolic execution of the program. We will look at each in more detail.

The PM model allows Agamotto to track the persistency state of each cacheline. Looking at allocations and instructions, it assigns a state to a cacheline, much like other already discussed systems. For example, a write to an allocated cacheline changes its state to dirty (*i.e.* modified). Agamotto's PM model also supports non-temporal stores (mentioned in 2), changing a cacheline's state directly to pending (*i.e.* flushed but not ordered).

The oracles assert the memory states, checking for bugs according to the cachelines' state. The universal bug oracle provided by Agamotto checks for objects not guaranteed to have persisted (unflushed), as well as extra flushes or fences. For example, to detect an extra flush, it checks if the flush is being performed on a cacheline whose state is pending or clean (*i.e.* has already been persisted, or both persisted and ordered, respectively). Custom oracles may be created to assert any desired behaviour with custom structures or PM libraries. The authors of Agamotto implement two custom oracles. The first one detects redundant undo entries inside transactions in PMDK, and the other verifies if a custom structure is updated within a PMDK transaction, guaranteeing the atomicity of the operation.

Agamotto takes a step further with the symbolic execution since it does not use the default heuristic of the engine it uses, KLEE [CDE08]. KLEE focuses on code coverage, meaning that on a branching execution (*e.g.:* an if block), it will try each branch in order, and then each branch from there, and so on. Another way to look at this is to imagine an acyclic tree with the various possible executions. To achieve a higher code coverage, KLEE will search it as if it is running a BFS (breadth-first search). Agamotto searches it using a custom heuristic. To achieve this, it assigns a weight to each instruction, and it chooses to search first the branches with instructions with the highest weights.

Agamotto performs static analysis of the code to assign weights to each instruction. This analysis looks at each instruction and checks whether it modifies PM or not. It assigns a weight of 1 or 0 to the instruction, where 1 means that it does modify PM. Then, with a back-propagation algorithm, it starts on exit points in the program (*e.g.:* return in a function) and, for each instruction, it assigns a weight to it corresponding to the number of PM-modifying instructions reachable from it. Let us look at the following code to understand this.

Algorithm 5: Example of PM-modifying code.

```
1 $char * pmemfile = mmap(< PMFilename >);$
2 $a = \ldots;$
3 if $a$ then
4     $cout << a;$
5 else
6     $pmemfile[x] = a;$
7     $clwb(pmemfile[x]);$
8     $sfence();$
9 end
```

We can follow Algorithm 5 to look at how this works. In this case, the `if` instruction on line 4 will have weight 0, as it has no instructions reachable from it that modify PM. The `else` branch, however, has 3 instructions that modify PM. When the back-propagation algorithm runs, it will combine the weights of lines 7 and 8, assigning 2 to line 7 and then 3 to line 6 by the same logic. The `else` instruction will then have weight 3 since 3 PM-modifying instructions are reachable from it. When Agamotto tests the program, it will choose to enter the `else` first since it has a higher weight.

This static analysis and change in state exploration steer the symbolic execution of the program toward more critical areas in terms of PM [NRSQ20]. It results in Agamotto being able to find bugs more quickly. The evaluation carried on by the authors of Agamotto shows that it excels at finding performance bugs (*i.e.* extra flushes/fences). It makes sense to observe this since application-specific bugs will usually require custom oracles, and the universal oracle provided by Agamotto focuses on performance bugs or trivial unflushed object bugs.

Given the nature of symbolic execution, we suffer from the same problem as Yat and Jaaru regarding state space size. If a program is too complex (*i.e.* has too many branches or loops), the size of the state space will explode. This causes a considerable slowdown in the execution of the testing tool, which may render it useless when time is always a factor. Another note is that Agamotto cannot execute kernel code, which is a disadvantage, considering that other systems offer (at least) extensibility for it.

3.4 Discussion

As discussed throughout this chapter, there is a tradeoff between efficiency and automation. Looking at Table 3.1, we can see that no tool in the state of the art has both high efficiency and automation. Tools focusing on automation rely on less efficient state exploration and bug detection techniques. Such is the case of Yat, Jaaru, and Witcher. As we will detail in Section 4 and Section 5, the approach for state exploration taken for this work will make the tool scale with $O(P + F \times R)$ instead of $O(P \times F \times R)$ (in the case of Yat, Jaaru, and Witcher), where P is the length of the program, F the number of faults, and R the length of the recovery process. Reducing the execution time will increase the efficiency further compared to other systems in the state of the art. On the other hand, more efficient tools have to resort to user annotations

[1]Jaaru's efficiency greatly depends on the program it tests. Because this work focuses heavily on the commit store pattern, the efficiency may skyrocket when analyzing a program that frequently uses this pattern. It can also plummet when a program never uses it.

[2]Because our oracle is the application's recovery, we assume the best-case scenario in which the recovery can detect all bugs associated with an application. In this scenario, our tool performs as best as expected in completeness.

[3]Using the recovery as the bug-finding oracle does not allow us to find performance bugs. This work focuses on efficiently finding correctness bugs, so we do not concern with performance bugs.

Tool Name	State Exploration	Bug Detection	Efficiency	Automation	Completeness
Yat [LDK+14]	exhaustive	data integrity checker	Very Low	Very High	Medium
Jaaru [GXD21]	model checking with constraint refinement	PM software simulation	Medium[1]	Very High	Medium
Witcher [FKP+21]	systematic pruning based on inference rules	output equivalence checking	Medium	Very High	High
PMDebugger [DLCL21]	N/A	bookeeping data analysis	High	Medium	High
PMTest [LWZ+19]	user annotations	persist and flush interval matching	High	Low	High
XFDetector [LSW+20]	user annotations	fault injection and shadow PM consistency checking	Medium	Low	High
Agamotto [NRSQ20]	symbolic execution with PM-aware search algorithm	bug oracles	Medium	Medium	Medium
This work	single-pass snapshot creation	recovery as oracle	Very High	Very High	High[2]

Table 3.1: Different tools and their characteristics.

Tool Name	Efficiency			Automation	Completeness	
	Offline Overhead	Execution Time	Recovery Time	Developer Effort	Correctness Bugs	Performance Bugs
Yat [LDK+14]	-	Very High	High	None	High	Low
Jaaru [GXD21]	-	High	Medium	None	High	Low
Witcher [FKP+21]	-	Medium	-	None	High	Medium
PMDebugger [DLCL21]	-	Low	-	User Annotations	High	High
PMTest [LWZ+19]	-	Low	Medium	User Annotations	High	Medium
XFDetector [LSW+20]	-	Medium	High	User Annotations	High	High
Agamotto [NRSQ20]	Medium	Medium	-	Oracles	Low	Very High
This work	-	Low	Low	None	Very High[2]	-[3]

Table 3.2: Efficiency, automation, and completeness analysis of the state of the art.

[LWZ+19, LSW+20], which reduces the number of verifications the tool has to do or the number of lines in the code to explore.

Although this work focuses more on efficiency and automation, Table 3.1 has a column for completeness. It is hard to fairly compare various systems regarding completeness without testing them under the same conditions and using the same test cases and workloads. Moreover, most widely used benchmarks do not have a dedicated recovery process that we can use to test our tool. The values presented for each are a qualitative view of each system's bug detection capabilities. Yat and Jaaru score medium because they lack techniques that find specific performance bugs, such as redundant flushes. Agamotto scores medium since it finds few correctness bugs without the necessary oracles for each application semantic. The rest of the tools score high because they find a good variety of performance and correctness bugs. The goal is to provide the developer with a tool to report bugs as fast as possible, ultimately speeding up the development of crash-consistent software.

We present a more detailed look at each tool's efficiency, automation, and completeness in Table 3.2. Efficiency is split into three aspects: offline overhead, execution time, and recovery time. Offline overhead regards any pre-processing the tool may do before testing a program, such as the static analysis Agamotto [NRSQ20] performs. Execution time is the total time the tool takes to test a program, which includes all executions and recoveries the tool has to perform. Recovery time refers to improving the recovery process (only applies to tools that perform recovery). Scoring lower on this metric means using techniques that can reduce the overhead caused by repeatedly executing the recovery process (*e.g.:* running multiple recoveries in parallel as is our case). Automation refers to the developer's additional efforts to use the associated tool. The more automated a tool is, the less effort a developer needs to use it. We split completeness into correctness and performance bugs. The former regards application semantics and crash consistency, while the latter contemplates misuse of PM instructions such as redundant fences.

We can notice a correlation between lower execution time and annotations or other techniques requiring extra developer effort. Yat and XFDetector score high on recovery time since both systems have a high count of fault injection points. Running the recovery process numerous times causes them to have an impacting slowdown in execution, which is why XFDetector scores worse in execution time compared to PMTest, its closest competitor.

This work has low execution time and full automation. We score low on recovery time because we parallelize the execution of multiple recovery processes, reducing its footprint on the overall efficiency of the tool. We assure full automation by not resorting to annotations, custom oracles, or similar techniques. We do not focus on new bug-finding techniques or finding performance bugs since we use the application's recovery as a bug oracle. Assuming a recovery that correctly identifies an inconsistent state after a crash, we find all correctness bugs in a program. One may argue that we still incur developer effort by using the recovery as a bug-finding oracle. We highlight that the recovery procedure is an integral part of developing PM software. Also, the effort to develop a good recovery program is lower than creating specific oracles or using user annotations. Moreover, the developer is familiar with their code, so developing a good recovery is more manageable than investing time into learning how a specific testing framework works and properly using its annotations for a specific application.

Chapter 4

Approach

In this chapter, we start by presenting some considerations that motivate the design choices of this work. We then introduce an overview of how the tool works to achieve the desired goal of efficiently detecting PM bugs. After, we look at each component in the tool's architecture and briefly explain how it works and contributes to the system. Finally, this section ends with a brief discussion of how the presented architecture achieves the projected goals.

We will use the concept of failure point from this point. A failure point is a point in the application where we simulate a crash. When creating a snapshot from a failure point, we can simulate a post-crash scenario from that point by executing the recovery process on that snapshot. We use stores as our failure points, as they change the state of memory. When we instrument a store, we create a snapshot of the memory state after that store was issued. Since a store changes the contents of memory, creating snapshots at these points generates a collection of snapshots that portray the evolution of PM as the application executes. This collection includes only the snapshots related to the program order, meaning the observable states if we execute the application in the order we instrument the instructions. Due to possible hardware reordering, the state we observe after a crash may not be included in this collection of generated snapshots. As such, we must generate more states that correspond to all the possible memory states when said reorderings are considered. However, as previously discussed in Section 3.1, generating all possible reorderings does not scale well for more complex applications. We use a series of heuristics that allows us to prune certain states, either because they have a lower chance of leading to a bug or because they do not portray a reachable memory state.

Our tool uses the recovery process as the consistency checker. This means that after creating our collection of snapshots, we run the recovery on each snapshot. We assume the recovery is correct. This means that if the program recovers correctly from every snapshot we have no bugs. Regarding correctness bugs, it depends on the quality of the recovery process. A correct recovery can detect all bugs mentioned in Section 2.5.2. By running the recovery on every created snapshot, we simulate how the application recovers from every possible state after a crash. This way, we can detect any bugs should the application not be able to recover from a certain state. We do not concern with performance bugs as we do not perform trace analysis. We mention in Section 7.1 how we could improve the tool to detect this type of bugs.

4.1 Design Considerations

As discussed throughout Chapter 3, many approaches exist to create a tool that tests persistent memory applications. There are three main aspects to consider: efficiency, automation, and completeness. To focus on efficiency, we had to think about optimizing each step of the testing process and designing components that would not add overhead. Some available tools [LDK$^+$14, GXD21, LSW$^+$20] execute an application to a certain point, simulate a crash, execute recovery, and continue the execution. They then repeat this for every failure point. Doing this results in executing the application `N` times, where `N` is the number of failure points. This observation motivated us to develop a solution that only executes the application once, improving efficiency.

There is a problem with having all snapshots in memory simultaneously: space. To solve this second issue, we decided to resort to copy-on-write (known as CoW, refer to Section 2.4 for more information). Using CoW allows us to create thousands of snapshots without incuring in a linear memory usage. Because each snapshot only differs by a few bytes from another, each snapshot created with CoW uses very little space.

Regarding automation, we want our system to follow a *plug and play* approach, with minimal effort to run the tool. As such, the user does not need to recompile any code, nor do they need to add custom annotations or create custom oracles.

To ensure completeness, we generate as many possible states as possible with the right balance between generating states that can lead to bugs and not generating too many states. If we generated all states by reordering the instructions, we would have a state-space explosion, as the authors of Yat [LDK$^+$14] demonstrate. At the same time, if we do not consider enough states, we may skip memory states which would lead to bugs, affecting completeness. To find this balance, we implement a series of heuristics that we apply to our reordering algorithm, which we explain in more detail in Section 4.7.

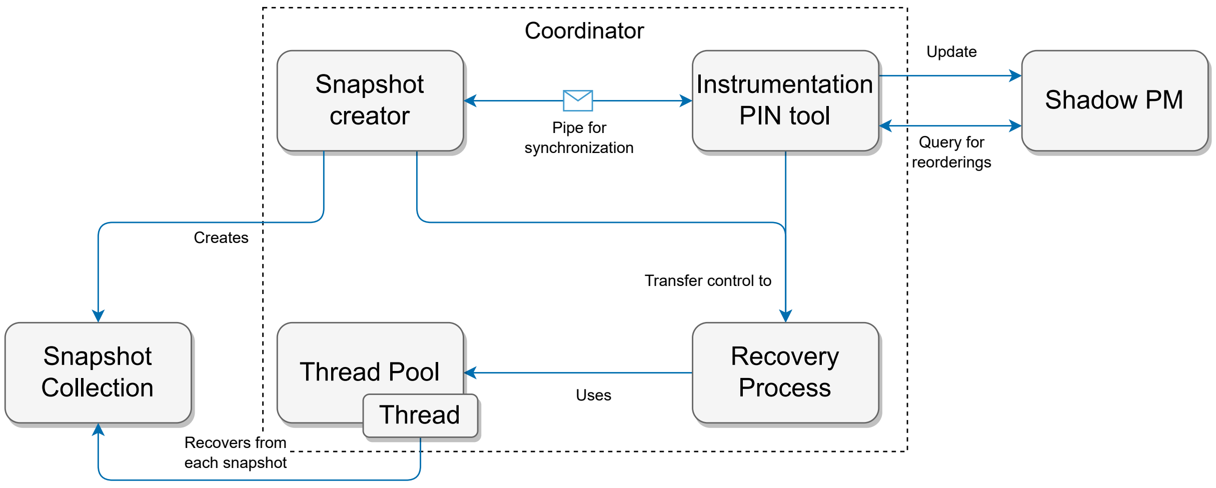


Figure 4.1: Architecture of the debugging tool

4.2 Overview - how does the system work?

The process of testing an application comprises three main phases: the configuration of the tool by the developer, the execution of the tool itself, and the analysis of the results. Since one of our main priorities is automation, the configuration must be simple. We still offer flexibility by giving a choice to the user of configuring certain aspects of the tool (explained in detail in Section 5.2).

Figure 4.1 presents the architecture of our tool. We can follow it to better understand how the system works:

1. The coordinator launches both the snapshot creator and the PIN tool that instruments the code.

2. The snapshot creator opens a pipe to listen to the PIN tool and vice-versa.

3. The PIN tool instruments the application, injecting the analysis routines at stores, flushes, and fences. When it finds one of these instructions, it updates the shadow PM. On stores, it signals the snapshot creator to create a snapshot.

4. At each fence, the PIN tool queries the shadow PM for information on the states of the memory addresses, which are used to obtain the states that are not observed in the program order.

5. The coordinator assigns each thread in a thread pool with a series of snapshots so they execute the recovery process on each.

4.3 Coordinator

The coordinator is not so much a component but a pipeline. It is important because it manages all the system components and allows the recovery process's parallelization. Figure 4.2 depicts how the coordinator works. It first issues the execution of the PIN tool and a utility responsible for creating the snapshots, as these work in parallel. After they finish executing, it counts the number of snapshots created and feeds this information into the recovery process. This last process divides the work evenly between threads to recover all snapshots.

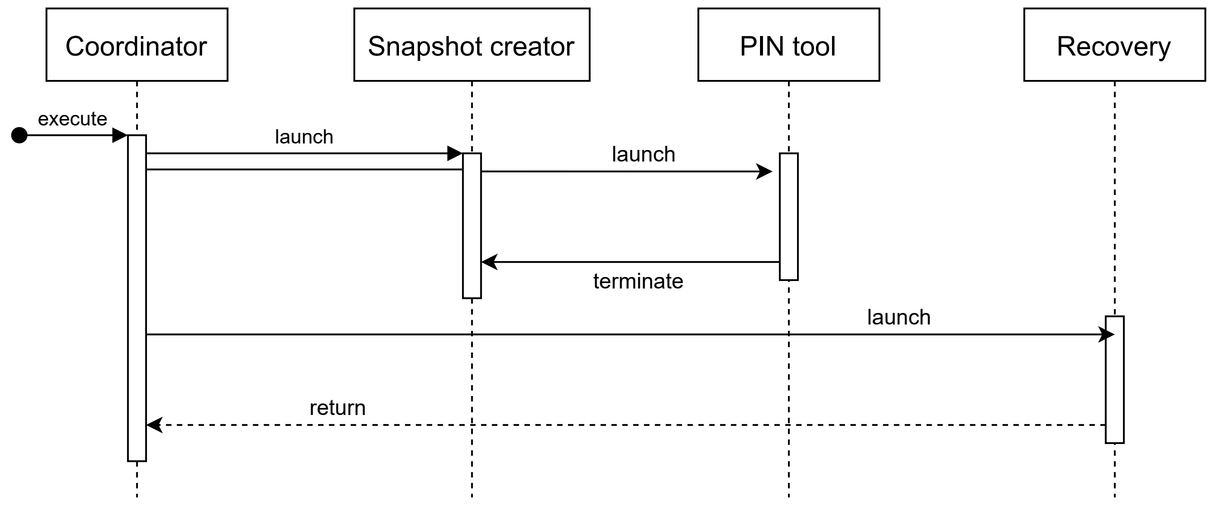


Figure 4.2: Time diagram of the coordinator

4.4 PIN tool

We resorted to PIN and created our own tool to instrument the application. We need instrumentation to know where our failure points are, as these will allow us to create relevant snapshots. Additionally, instrumenting other instructions, such as flushes and fences, allows us to keep track of the state of PM.

This tool looks for stores to PM, flushes, and fences. When it reaches a failure point (*i.e.* a store), it halts the execution of the application and sends a signal to an external utility which creates a snapshot. This snapshot captures the state of PM at that point in the execution. Doing this for every failure point nets us the collection of snapshots regarding the observed execution of the application. To consider the other possible states due to the hardware reordering of instructions, we need a shadow PM which keeps track of the state of memory addresses (explained with more detail in Section 4.6).

When the PIN tool finishes instrumenting the application, it sends a message to terminate the external utility that creates the snapshots, as depicted in Figure 4.2 as `snapshot creator`.

To help the developer with the debugging process, our PIN tool creates a backtrace of the execution that led to the state portrayed in a snapshot. Snapshots have unique IDs, and each backtrace is linked to an ID. These IDs are assigned in a monotonically ascending order. When a bug is reported after recovering with a specific snapshot, the tool presents the corresponding backtrace for more information on the source of the bug. This allows the developer to determine the execution path that led to the bug.

4.5 Snapshot creator

An external program is responsible for creating the snapshots. This program listens through a pipe to what the PIN tool sends, so it knows when to create a snapshot. When the PIN tool reaches a failure point, it halts the execution and sends a message to this pipe. The snapshot creator receives this message, creates a snapshot, and sends a message to the PIN tool, so it knows that the copy has been created. We use this messaging mechanism while halting the program's execution to ensure synchronization between the PIN tool and the snapshot creator. This synchronization guarantees that a snapshot portrays a specific state in memory. If we did not halt the execution, the snapshot could contain other values in memory which could not correspond to the failure point in question. We designed this program with efficiency as the main priority. Each snapshot must be created as fast as possible because the application being tested is halted during the snapshot creation. This is reinforced with the use of copy-on-write. The snapshots are created very efficiently because each snapshot differs only a few bytes from the others.

4.6 Shadow PM

We have mentioned that creating the snapshots corresponding to the observable states during a single execution with no crashes is not enough to get a collection of snapshots that portray all possible states. We have also discussed how it is not feasible to consider every possible state. As such, we must find a balance. We can generate more possible states using a shadow PM, and we can prune several impossible or less interesting states with a series of heuristics. We consider less interesting states those that have a lower chance of leading to a bug.

The shadow PM is a structure that keeps track of the state of persistent memory mapped to the application being tested. As such, it maintains the state of each memory address affected during the execution of said application. We need the PIN tool to instrument stores, flushes, and fences to maintain the shadow PM, as each instruction impacts this structure differently.

The shadow PM contains a data structure that maintains information about memory addresses during the execution of the application. Such information includes the state in which the address is and all the possible values it may contain in the event of a crash. The state of the address can be one of three. Modified, it is is not flushed nor fenced. Writeback-pending, if it was flushed but not fences. Persisted, if it was flushed and fenced. The PIN tool uses this information to check what other states are possible if we consider a post-crash execution at a certain point in the application. We know that a fence guarantees that all flushed stores are persisted after it [RWNV20]. As such, we can consider these additional states and create the corresponding snapshots before each fence, where an address may hold different possible values in a post-crash execution.

Let us look at the example code in Algorithm 6 to understand the importance of creating these states through reorderings. When using PIN to instrument that code, variable `a` will always have the value 1 after line 6 as the execution is linear and follows the order of the instructions as they appear in the code. However, due to hardware reordering and the nature of PM, we lose some guarantees. In particular, in an actual execution outside testing, the store to `a` in line 6 may not persist before the fence in line 10 is executed. As such, we need to generate a state where `a` has a value corresponding to its previous value in memory (which is 0 because of the store in line 1 that has persisted from line 5 onwards), which we can get from the shadow PM.

Algorithm 6: Multiple stores in a segment.

```
1  a = 0
2  b = 0
3  flush a
4  flush b
5  fence
6  a = 1
7  flush a
8  b = 2
9  flush b
10 fence
```

Once again, to keep the focus on efficiency, the data structure has to be optimized to incur as little overhead as possible during instrumentation. We have insertion, search, and delete times of O(1) for entries in the shadow PM to achieve this. We will discuss this in more detail in Chapter 5.

4.7 Snapshot generation and heuristics

Most of the logic of this tool happens between two consecutive fence instructions. A fence changes the memory's persistency state, as it guarantees that any subsequently flushed addresses are always observed after that fence. This means it makes sense for a debugging tool to explore the state of memory just before this change happens, which is before a fence. To better explain the following concepts in this document, we will use the term `segment` to denote any part of an application that resides between two consecutive fences.

The base idea for generating all the possible states is creating all combinations for the possible values each address may hold at a certain point in the execution. Ideally, we would like to explore all possible states and recover from each state, as that would grant us total coverage. However, if we want a tool that can test an application in an acceptable timespan, we have to prune certain states.

Let us look at Algorithm 6. In the second segment, we have two stores to variables `a` and `b`, which reside in separate addresses in memory. When we reach line 10, we know that `a` has the value 1 and `b` has the value 2. However, if we simulate a crash just before the `fence` in line 10, we may observe different states in memory.

	a	b
1	0	0
2	0	2
3	1	0
4	1	2

Table 4.1: Reorderings for a and b in Algorithm 6.

Table 4.1 shows the possible reorderings if no ordering restrictions exist between the two addresses. This means we go from one possible state from a single non-crash execution instrumented by PIN to four possible states in memory if we consider all post-crash execution scenarios. For every different address with possible values in the same segment, we have to multiply the number of possible reorderings by the number of possible values in that address. This does not scale well for more complex applications, which is why we must find certain heuristics that can help reduce the number of possible states to explore at each segment.

After analyzing some PM programs, we came up with three heuristics:

- Limit number of stores at segments;

- Limit age for a store to persist;
- Reordering restrictions with timestamps.

We will now go over each heuristic, explaining its rationale and the projected impact on the number of snapshots created through reorderings. For more flexibility, the user can decide whether to create these possible reorderings and which heuristics to consider. This creates a tradeoff between efficiency and completeness, and it is up to the user to choose the balance they desire.

4.7.1 Heuristic 1 - Limit number of stores at segments

In the examples provided so far, we used PM code examples that changed a few addresses' values. In particular, we had only two variables with Algorithm 6, which can lead to four possible states as seen in Table 4.1. In actual PM programs, we have some segments which contain many dozens to thousands of variables. This leads to an explosion of possible states should we reach a certain number of addresses written in a segment. At the end of a segment, the least possible reorderings without ordering restrictions are $r = 2^w$, where w is the number of writes to unique addresses. This is a consequence of both the cartesian product and any address having at least two possible values: the one last written to it and its previous value.

With the previous considerations, we introduce our first heuristic. It consists in limiting the number of stores in each segment. This means that we only create reorderings at segments that contain, at most, the limit of stores defined by this heuristic. This heuristic can prove very useful if we test with programs that contain few stores at most segments but have few segments with thousands of stores.

To understand the possible impact of this heuristic, let us look at the example in Algorithm 7. If we considered the reorderings of all segments, we would have $2^n + 2^1$ possible reorderings. This number comes from the fact that we have n stores to unique addresses in the first segment and one store in the second segment. Let us assume, for instance, that $n = 30$. Then, the total number of reorderings is $2^{30} + 2^1 \sim 1.07 \times 10^9$. Applying this heuristic with a limit that is smaller than 30 reduces the total number of reorderings to 2 since we only consider the second segment.

Note that we are only mentioning reorderings. This heuristic does not affect the snapshots created by the execution in the program order.

Algorithm 7: Program with a segment with high store count.

```
fence
var_1 = 1
flush var_1
...
var_n = n
flush var_n
fence
var_m = m
flush var_m
fence
```

4.7.2 Heuristic 2 - Limit age for a store to persist

With this heuristic we introduce a new concept: age to persist. We consider the age of a store as the distance in segments between the segment where the store is issued and the segment in which that store is persisted. To better understand this, let us look at Algorithm 8, where the store to `a` is persisted in the same segment that it is issued. In this case, the age to persist is 0. If we now look at Algorithm 9, we can see that there is a store to `a` in the first segment, but this store is only guaranteed to persist two segments after. Therefore the age to persist is 2, as it took two segments between the issue of the store and its guaranteed persistence.

Algorithm 8: PM code example that shows an age to persist of 0 for `a`.

```
...
fence
a = 1
flush a
fence
...
```

Algorithm 9: PM code example that shows an age to persist of 2 for `a`.

```
...
fence
a = 1
fence
...
fence
flush a
fence
...
```

When we create the possible reorderings at the end of a segment, we may have to consider specific addresses that have not persisted for several segments. We analyzed programs that do not guarantee the persistence of certain stores through the use of flush and fence instructions.

Again, in line with the logic from the previous heuristic, we know that some of these stores regard initialization. We also noticed a pattern which let us conclude that many stores that are not persisted with flush and fence regard memory allocation (detailed in Chapter 6). To mitigate having to consider certain stores that may not have persisted for several segments, we implement this heuristic by defining a limit to the age for a store to persist. This means that if a store has not been guaranteed to persist for a certain number of segments, then we do not consider it anymore for future reorderings. We take inspiration from a study performed by the authors of PMDebugger [DLCL21] which concludes that most stores are persisted in the segment they are issued (*i.e.* their age to persist is zero). This helps ensure that we do not discard too many states. To avoid fully ignoring these cases, we can warn the user that this situation occurs, which can give better insight into a possible bug we may be overlooking when applying this heuristic (also mentioned by the author of PMDebugger [DLCL21]).

4.7.3 Heuristic 3 - Reordering restrictions with timestamps

When we first looked at the possible reorderings and the resulting possible states in Section 4.7, even though we observed only one state during program execution, we can, in fact, observe 4 in a post-crash execution (see Table 4.1). This may be true for some scenarios but may not be the case for others. Let us resort to Algorithm 6 and think about two possible scenarios to understand this better.

1. Stores to the same cacheline - in this scenario, a and b are two addresses that reside in the same cacheline. If we have two stores, one for each address in this cacheline, then the persistence of one address means the persistence of the other. Because the store to a happens first, we invalidate the state where $a = 0 \wedge b = 2$ (0 being the previous value of a). Following the persistency semantics of Intel-x86 architecture [RWNV20], a flush is atomic at the level of a cacheline, which means that $b = 2 \implies a = 1$.

2. Ordered flushes - in this scenario, the flushes in lines 7 and 9 are normal flushes (*i.e.* not `clflushopt` or `clwb`) which cannot be reordered with stores. Now we have the same implication as the previous scenario but for a different reason. Because we know that the flush on line 7 has to happen before the flush on line 9, then $b = 2 \implies a = 1$, which again excludes the case where $a = 0 \wedge b = 2$.

Considering these two scenarios, we can apply a monotonically ascending timestamp to a store at the time the store is flushed with a flush instruction that cannot be reordered. We timestamp the same value to other stores to addresses in the same cacheline to use the rationale in the first scenario. This timestamp is then used to limit the stores from other addresses with which the store can be reordered. Analyzing Algorithm 6 once more, and considering both flushes to be flushes that cannot be reordered, we know that $ts_a(1) < ts_b(2)$ in scenario 2, where $ts_x(v)$ means the timestamp of x for value v. This means that we can combine the value 1 of a with any value of b whose timestamp is higher than $ts_a(1)$ since it happened later. The opposite is not true, resulting in $a = 1 \implies b \in \{0, 1\}$. Because $ts_b(2) > ts_a(1)$, we can only combine the value 2 of b with the highest timestamp for a store to a, resulting in $b = 2 \implies a = 1$.

To fully consolidate this idea, let us now consider Algorithm 6 but assuming the flushes are `flushopt`, meaning they can be reordered. When a flush can be reordered, we timestamp the associated store with a default value for that segment. The default value has to depend on the segment to prevent reorderings between segments when a store may have been issued and persisted for a certain address in a previous segment[1]. In this scenario we have $ts_a(1) = ts_b(2)$. Because the timestamps are equal, we can observe the following: $a \in \{0, 1\} \wedge b \in \{0, 2\}$.

4.8 Discussion

We have discussed how testing a persistent memory application has many challenges and is not trivial. The available tools in the state of the art either test too many states and take too much time or ignore possible erroneous states and miss some bugs. Some tools find a compromise in the middle but can do so only thanks to user annotations or custom oracles that incur extra work for the developer. We believe it is better for the developer if their additional effort in testing their application is minimal. In particular, our design choices contribute as follows:

- Using the recovery as the bug-finding oracle removes the ability to find performance bugs but grants automation. We focus heavily on automation and efficiency and therefore consider this an acceptable tradeoff;
- Executing the application a single time, resorting to copy-on-write for creating the snapshots, and parallelizing the recovery process boosts efficiency;
- Generating snapshots through reorderings enhances completeness as we explore more states than the ones observed by the program order. We mitigate state-space explosion with the implementation of three heuristics. These heuristics focus on pruning less interesting states

[1]Let us assume a default value of 0, regardless of the segment. If we had a store that was flushed in segment N with a flush that can be reordered, then it could be reordered with any store that was flushed in a segment $N - M, M \in [1, N[$, as the timestamp would be smaller. This would break ordering guarantees, as any store flushed in a previous segment cannot be reordered with a store flushed in a following segment.

as they have a higher chance of being related to the program initialization and memory allocation or being states that are not observable.

Chapter 5

Implementation

In Chapter 4, we introduced each of the components that make up our tool and the contribution of each to the desired goal of debugging a PM application. Focusing on efficiency means carefully designing the system so that each component incurs as little overhead as possible. In this chapter, we will start by reiterating how the system works and then detail how we implemented each component with efficiency and automation in mind.

Our tool starts with the coordinator which is a bash script that orchestrates each component to perform its function. The main element is a custom PIN tool we use to instrument the application we are testing. PIN halts the execution of the application at stores, flushes, and fences. While the execution is halted, it communicates with an external program to create snapshots of that specific point in the execution. A shadow PM is maintained during instrumentation and is updated by the PIN tool each time it encounters one of the instructions mentioned above. This shadow PM will aid in creating all possible instruction reorderings between two fences so that we test all states that can lead to an erroneous post-crash execution. When all snapshots have been created, we proceed to the recovery process. To increase the efficiency of the recovery, we parallelize it. By attributing a slice of the snapshots to each thread available, we achieve considerable performance improvements in efficiency in this last process.

5.1 Components

We will now look at how each component was implemented in detail. We exclude the pipeline from this section because the document details it in the previous chapter. We will explain the possible configurations at the end of this section after going through all relevant details about the implementation. This will make it easier to understand each possible parameter in the configuration.

5.1.1 PIN tool

The PIN tool is the central component; it encapsulates the logic behind the instrumentation and when to create copies and backtraces. It manages the updates to the shadow PM and the creation of the possible reorderings between fences. To safely explore the values in memory without altering the application's logic, PIN exposes a function to copy contents from memory [Intc]. Algorithm 5.1 shows how we use this function to retrieve the value in any memory address. Regarding instrumentation, our tool injects code to run before every store, flush, and fence instruction. This code depends on the instruction and can create a snapshot of memory and a backtrace of the program at a point in the execution or update the shadow PM as we describe next.

```
int GetValueFromMemory(ADDRINT address) {
    ADDRINT * addr_ptr = (ADDRINT*)address;
    int value;
    PIN_SafeCopy(&value, addr_ptr, sizeof(int));
    return value;
}
```

Algorithm 5.1: Reading the value from a memory address.

Instrumenting stores

When instrumenting stores, we first need to know whether the store was to an address in persistent memory. We perform a range check: if the address plus the size of the store is inside the address range assigned to PM, then it is PM. To be able to know what memory ranges correspond to PM, we instrument calls to `mmap` which allows us to create a structure that maintains this information. Being a store to PM, we need the address and the value. We cannot get the value immediately because we instrument the store before it happens, so the value is not in memory yet. As such, we keep track of the last address which had a write to it, and when instrumenting the next instruction, we check that value, saving it. When we save this value, we add it to the shadow PM as a modified address. For every instrumented store, we send a signal to the snapshot creator, in order to create a snapshot of the newly modified memory state.

Instrumenting flushes

The code we inject when instrumenting flushes is to update the shadow PM. We obtain the target address of the flush instruction and modify the corresponding addresses in the shadow PM to display a flushed state. Algorithm 5.2 shows the code that executes at each flush. We call the update to the shadow PM regarding flushes (detailed in Section 5.1.3). Then, for statistics, we increment the number of instrumented flushes.

```
void FlushHandler(const CONTEXT *ctxt, ADDRINT address) {
    shadowpm.processFlush(address);
    flush_count++;
}
```

Algorithm 5.2: Instrumenting a flush.

Instrumenting fences

Instrumenting fences contains two parts. First, we update the entries in the shadow PM to mark any flushed addresses to persisted. Second, we create the possible reorderings that will lead to more possible states besides the one directly visible from the executed program order. Algorithm 5.3 shows the code that instruments a fence. Compared to flushes, we do not need an address, as a fence acts on the memory as a whole, persisting any flushed addresses. If we are considering reorderings, we call `ApplyPossibleReorderings` on the shadow PM. This function will loop through all possible states, which are groups of address-value pairs. For each pair, it will write the value to the paired address. After doing so for all pairs in a group, it will signal the external program to create a snapshot. It repeats this for every group of pairs until we get all snapshots of the possible reorderings. Section 5.1.3 details how the reordering groups are calculated. Notice that we only process the fence instruction on the shadow PM after considering the reorderings. This is because we need to know which addresses were not persisted before the reorderings, as those are the ones whose value is not certain. If we calculated the reorderings after changing the statuses of the flushed addresses to persisted, then we would not consider all possible states. An important note here is that we cannot create backtraces corresponding to the reorderings' snapshots. A backtrace indicates how the execution reached a certain state, and, in this case, we are simulating said state by writing values directly to memory. This means that every reordering between two fences would have the same backtrace.

Moreover, this backtrace would be the same as the one created before creating these reorderings - the snapshot before the fence. After processing the fence at the level of the shadow PM, we restore the memory to the state it was before creating the reorderings. This means writing to memory all the previous values through the function `ReapplyPreviousValues`.

```
void FenceHandler(const CONTEXT *ctxt) {
    if (test_reorderings) {
        ApplyPossibleReorderings(shadowpm.getPossibleReorderings());
        shadowpm.processFence();
        ReapplyPreviousValues();
    } else {
        shadowpm.processFence();
    }
    fence_count++;
}
```

Algorithm 5.3: Instrumenting a fence.

5.1.2 Snapshot creator

The snapshot creator is a utility responsible for creating a snapshot when the PIN tool requests so. It is executed at every store and must be extremely efficient to incur as little overhead as possible. It is implemented in C++ to access lower-level functions closer to assembly, reducing overhead. Initially, our idea was for the PIN tool to create the snapshots. However, PIN replaces a series of C runtime headers and only offers support for a limited set of libraries which Intel calls PinCRT [Int12]. This set of libraries does not include support for the `cp` [Kera] command, which we need to create a snapshot. This led us to implement the snapshot creator as an external program that communicates with the PIN tool via pipe messaging. It is a program on a loop that waits for a signal on this pipe, creates a snapshot with a specific ID, and returns to waiting. This behaviour is detailed in Figure 5.1. It creates a snapshot by issuing the `cp` [Kera] command. We increment the snapshot number locally instead of receiving it from the pipe for smaller pipe messages.

Since PM programs have an average magnitude of thousands of instructions, we will create thousands of snapshots. As such, we needed a space-efficient way to simultaneously store all snapshots in memory. We use copy-on-write (or CoW, see Section 2.4 for more detail) to achieve this. Ideally, we would use ext4 as it is a widely used and supported filesystem along with direct access (Dax [pme]) for performance. However, we found two issues when trying to use ext4 with Dax. First, ext4 does not fully support CoW. Second, Dax is not compatible with CoW. This incompatibility comes from the fact that these two features use different techniques to manipulate pointers to data blocks, making them mutually exclusive. We resort to xfs as a filesystem because it fully supports CoW.

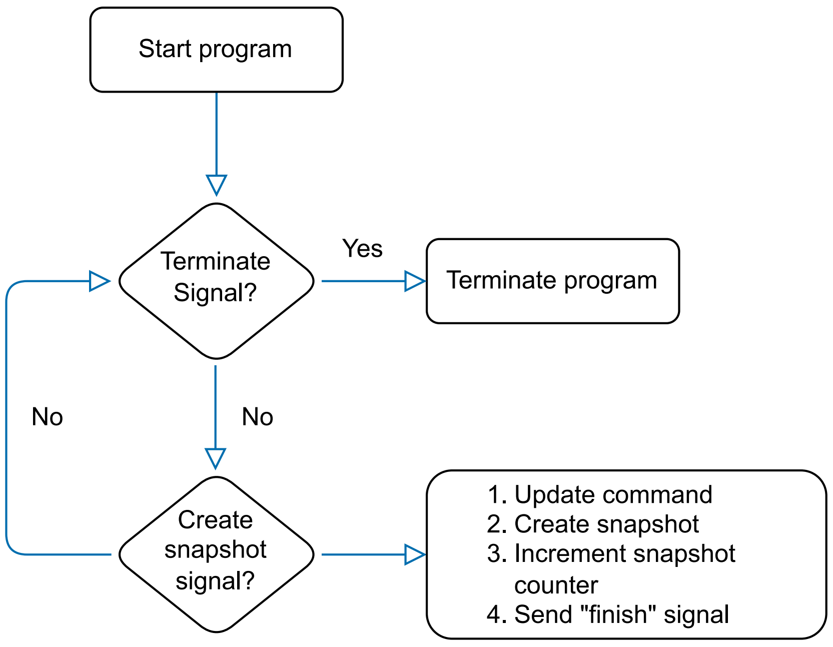
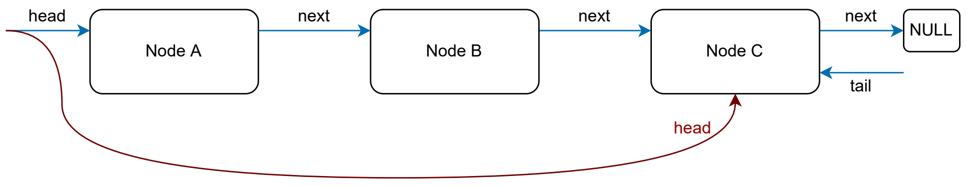


Figure 5.1: Diagram of the program that creates copies.

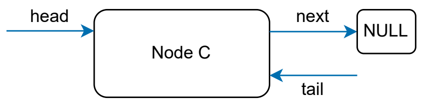
5.1.3 Shadow PM

A shadow PM will save the state of all the addresses modified during the execution of an application. As such, it may contain thousands of entries. We chose to go with a hash table as a data structure because it provides time complexity of O(1) for insertion, access, and deletion [CLRS09], and we want to be as efficient as possible bookkeeping the shadow PM. In this table, the key is the address, and the entry is a custom data structure. This data structure contains a state which can be `modified`, `writeback-pending`, or `persisted`. A `modified` address has a store which has not been flushed or fenced. A `writeback-pending` address has been flushed, but not fenced, and a `persisted` address has been flushed and fenced, hence being persisted.

The structure also contains a sequence of all the values that can be observed in that address due to possible reorderings given the ordering constraints enforced. Again, to focus on efficiency, we implemented a linked list (with a reference to the last element - `tail`) as the structure that will store the possible states. This allows for the insertion time complexity of O(1) and the deletion of all but the last element of O(1). The second is important because when we process a fence, all flushed addresses will change to persisted, and their possible values will reduce to only the last value. This reduction means deleting all nodes in the linked list but the last one. Figure 5.2 shows this operation. Each node corresponds to an element of the linked list containing a value. By changing the reference as marked by a red arrow in Figure 5.2(a) (*i.e.* changing the head to the last node), we successfully delete all values except the last one. The result is the list in Figure 5.2(b). Because we have a linked list with a reference to the last element, changing the head reference to the last element is simply changing the head to be the same as the tail. In a conventional linked list, we would not have a reference to the last element, so we would need to traverse the whole list.



(a) Before deletion



(b) After deletion

Figure 5.2: Deleting all possible values but the one persisted.

We have detailed how the PIN tool instruments each type of instruction. This tool calls different methods on the shadow PM to update its status. We will now detail these methods and see how they affect the shadow PM.

Processing stores

We have one of two scenarios when we process a store to an address. Either there is already a store to that address, or there is no such address in the shadow PM. In the first case, we solely need to add the new store to the list of possible values. In the second case, we create an entry in the shadow PM with a list of possible values with only one value: the one written in the store. Algorithm 5.4 shows both cases. Lines 4 to 6 correspond to the case where the head of the list is defined, meanin we already had a value in that address. The head of the list corresponds to the first element in the list. In this case, we need to change the references around the tail, as the new value becomes the new tail of the list. Lines 7 and 8 refer to the case of creating the list where the new value is both the head and the tail, as it is the only element in the list.

```
void addValue(int value) {
    Node* new_node = new Node(value);
    if (head) {
        tail->next = new_node;
        tail = new_node;
    } else {
        head = new_node;
        tail = new_node;
    }
    size++;
}
```

Algorithm 5.4: Adding a value to an address in the shadow PM.

In both cases, we change the state of the address to `modified` as we have a new store to that address. This change is not depicted in Algorithm 5.4 because it only shows the code inside the scope of the data structure that maintains the possible values for that address. The state of the memory address resides outside this scope.

Processing flushes

The process of updating the shadow PM with a flush is as follows: if the target address of the flush is `modified` or `writeback-pending`, then the state becomes `writeback-pending`. Algorithm 5.5 reflects this operation. Since we are maintaining a shadow PM and keeping track of each address's state, we could detect redundant flushes and fences. In this case, flushing an address whose state was already `writeback-pending` would mean the presence of a redundant flush. We will refer to this in future work as it is not within the scope of this work but can be added to improve the tool further and detect some performance bugs.

```
void processFlush() {
    if (getState() != AddressState::PERSISTED) {
        changeState(AddressState::WB_PENDING);
    }
}
```

Algorithm 5.5: Processing a flush on the shadow PM.

Processing fences

Algorithm 5.6 demonstrates how we process a fence on the shadow PM. Because fences act on the whole memory, we run this function for every entry in the shadow PM. We start by checking the state of an address, and if it is `writeback-pending`, then we change it to `persisted`. Then we remove all the values from the list of possible values except the last one, the persisted value. This operation is depicted in Figure 5.2.

```
    void processFence() {
        if (getState() == AddressState::WB_PENDING) {
            // Change to persisted if it is wb-pending
            changeState(AddressState::PERSISTED);
            // Remove all possible values but the last one
            possible_values.remove_all_but_tail();
        }
    }
```

Algorithm 5.6: Processing a fence on the shadow PM.

Creating the possible reorderings

Now that we have seen how the shadow PM works, let us look at how we create the possible reorderings between instructions. Reordering instructions means that, in the event of a crash in a segment, we may observe different values in the addresses that were written in that segent. We explained how we store the possible values for each address in the shadow PM in a linked list. To get all possible states of memory would mean getting all combinations of these values. Creating all reorderings would mean applying the cartesian product to the lists of possible values [Viz63].

However, since not all reorderings correspond to possible states, and because we cannot create all possible states due to exploding the state space, we apply the heuristics introduced in Section 4.7. By imposing a series of conditions during the creation of each combination, we are able to significantly reduce the number of snapshots created while only slightly reducing completeness, as we may overlook certain states that can lead to an erroneous behaviour.

5.1.4 Recovery process

We explored how running the application once and creating all snapshots in one run has its advantages in efficiency. We will now detail the biggest advantage: parallelizing recovery. Because we reach the recovery process with all snapshots created, we can issue the recovery on each snapshot in parallel. Theoretically, we could have a completely parallelizable scenario since there are no dependencies between different snapshots. The lack of dependency means the only synchronization necessary is having each thread wait for the recovery to finish on a certain snapshot to then move on to another snapshot. However, it is not as straightforward since distinct hardware configurations, kernels and filesystems behave differently under different workloads.

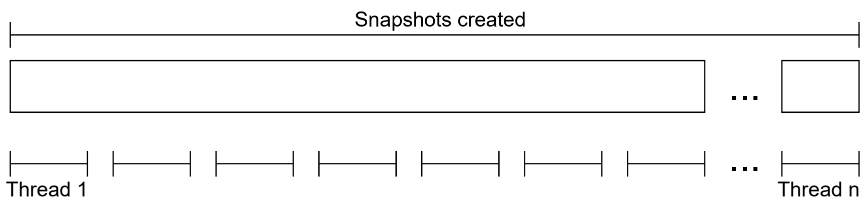
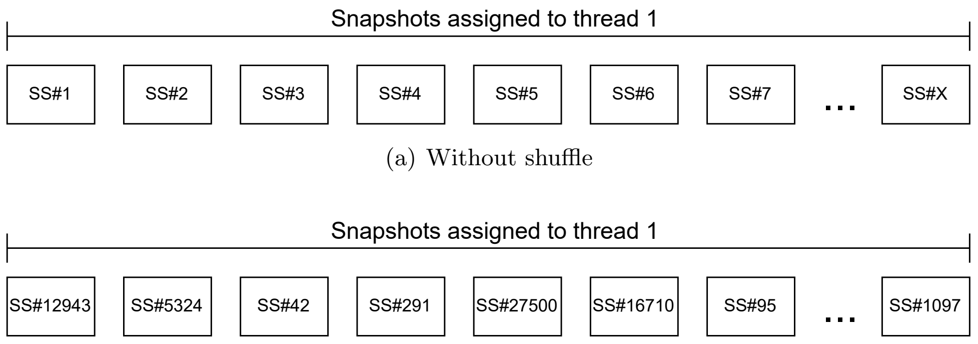


Figure 5.3: Assigning the different snapshots to threads.

Chapter 6 will detail these observations.

Because different snapshots may lead to different times in recovery, assigning the snapshots orderly to the different threads may have a cost on overall efficiency. To mitigate this issue, we have the option to shuffle the different snapshots. This ensures an evenly distributed workload between all threads. We noticed this behaviour when using PMDK's applications since there is an initialization of the PM pool with thousands of stores and flushes. These stores would each create a snapshot. However, these snapshots are within the initialization, which causes the recovery to run quickly. If we assigned the snapshots to the different threads in the order they were created, the first threads would finish running the recoveries much faster than the other threads.



(a) Without shuffle

(b) With shuffle

Figure 5.4: Example of a thread's assigned snapshots.

Figure 5.3 depicts how we assign each snapshot to each thread. Figures 5.4(a) and 5.4(b) show an example of the assigned snapshots to a thread, without and with shuffle, respectively. `SS#N` means snapshot number `N`. The last snapshot in Figure 5.4(a) is numbered with a `X` as the number depends on the total number of snapshots and available threads. If we have `S` snapshots and `T` threads then each thread is assigned `X` snapshots where $X = \frac{\mathtt{S}}{\mathtt{T}}$.

5.2 Configuration

We now detail the configuration options available to the user. We will divide this Section into two parts: the mandatory configuration, which includes the necessary user inputs for the tool to run correctly, and the additional features.

5.2.1 Mandatory configuration

For someone to use our tool, there is only a single step they have to take: indicate in a configuration file how to run the application and the recovery procedure, and indicate the name of the file to which the application will write. Additionally, we assume the name of the mount of PM, but that is very easy to change through an argument when using the tool.

5.2.2 Additional features

We offer flexibility by allowing the user to customize many aspects of how the tool works. We will list only those with a relevant impact, as the rest regards ergonomics and is not a key aspect in the scope of this thesis.

- Test reorderings - a flag passed to the tool defines whether we consider reorderings or not. Ignoring reorderings reduces the testing time but may ignore several states, which could lead to erroneous states.
- Parallel recovery and the maximum number of threads - the user has the option to choose whether they want a linear or parallel recovery and what is the maximum number of threads to use in the recovery process.
- Shuffle snapshots - see Section 5.1.4.
- CPU statistics collection - the user may choose to enable a flag which tells the tool to gather statistics regarding the execution, namely the CPU share of user, idle, system, and others.

5.3 Discussion

Now that we have touched all aspects of the tool, we can compare it to the state of the art once more. In Section 3.4, we analyzed the different available systems and concluded that no system had both high automation and efficiency. We now present a tool that offers both, while improving the state of the art in the following regards:

- The efficient design of the creation of snapshots and the use of heuristics which improve our work over Yat [LDK+14] and Jaaru [GXD21];
- The lack of additional effort by the developer which increases automation when compared to PMTest [LWZ+19], PMDebugger [DLCL21], XFDetector [LSW+20], and Agamotto [NRSQ20];
- A single passage through the application and the parallelization of the recovery process which may increase efficiency when compared to Witcher [FKP+21] and XFDetector [LSW+20];
- A tool that can test any type of PM program unlike Yat [LDK+14] or Witcher [FKP+21].

These considerations create a tool that stands out in the state of the art for bringing together efficiency and automation without sacrificing completeness.

Chapter 6

Evaluation

This chapter will show our tool's evaluation and how efficient it is, as this is the main contribution of this work. It starts with the specifications of the machine used for testing and the tools used. It then specifies what metrics we use for each type of test and what tests and workloads we consider in the evaluation. It follows with a section detailing the results and their analysis. It ends with a brief discussion of the presented results and how they address the proposed objectives in Chapter 3 and Chapter 4.

6.1 System Configuration

We tested our tool with the configuration in Table 6.1.

We use PM in App Direct mode since it allows applications to use it directly as a block device. Our tool requires xfs with reflink enabled in a PM mount to be able to use copy-on-write. We use xfsprogs 5.19.0 in our tests unless specified otherwise for any comparison between versions. We have noticed how different versions may affect multithreaded performance, so performance can differ for different versions.

CPU	Intel(R) Xeon(R) Gold 6338N CPU @ 2.20GHz, 128 cores
PM	8x128GB Intel DCPMM, App Direct Mode
DRAM	8x32G DDR4, 2666MT/s
OS	Ubuntu 22.04.1, Linux kernel 5.15.0
Tools	gcc/g++-11.2.0, xfsprogs-5.19.0, Intel's PIN-3.24, Dstat

Table 6.1: System configurations.

Our tool uses Intel PIN 3.24 to instrument the application's code.

We use dstat to obtain metrics regarding the use of CPU, which gives us insight into the performance when using multiple threads, so we can understand the machine's behaviour and improve efficiency. Dstat captures CPU usage at the system level, so there is always some degree of noise in our values. We always tried to have nothing but the essential system processes running simultaneously as our tests, but it is not always guaranteed that we fully control these factors.

6.2 Metrics

Ideally, we want to evaluate automation, completeness, and efficiency. Evaluating automation is hard in our context. We could have users use our tool, check the time they take to configure it for different applications and compare it to other tools in the state of the art. However, this also regards ergonomics, which is not relevant to the scope of this work.

Regarding completeness, since we use the recovery process as our oracle, we need to test with programs that have a correct recovery process. We do not do this because most widely used programs for PM benchmarks do not offer a specific recovery program.

Efficiency can be evaluated directly through some metrics. We will focus on `execution time` to evaluate how the parallelization of the recovery can boost testing speeds. To evaluate how our heuristics perform, we will look into the `number of snapshots created`, which is directly proportional to the `execution time` if we consider only the recovery process.

We use these two efficiency metrics to compare the baseline, our tool with linear recovery and no heuristics, with our proposed solution: our tool with parallel recovery and heuristics. We will also divide the execution time into two parts: total execution and recovery time. This allows us to obtain the speedup of just the recovery using parallelization and the overall execution of the tool.

We do not compare our tool with others regarding execution time since other tools use different techniques and specific library versions, which would imply an extreme experimental effort, which is not possible in the timespan available for this work.

In Section 5.1.2, we introduced the tradeoff between using ext4 with Dax and xfs with CoW. We define the following metrics to compare this tradeoff: `execution time`, `CPU usage`, and `memory` used to store the snapshots. These metrics should allow us to justify why we choose one over the other and how each performs to contribute to the overall efficiency.

6.3 Tests and workloads

To obtain the `number of snapshots created`, we will test with different benchmarks from PMDK that are included with the `libpmemobj` library [Intd] and from pmemkv [Inta]. Namely, we will use `Rbtree`, `Hashmap tx`, and `Skiplist` for PMDK. As for pmemkv, we use the `Cmap` and `Stree` engines.

`Rbtree` uses a red-black tree in PM, which reduces search time. `Hashmap tx` uses a hashmap and resorts to PMDK methods to ensure the atomicity of the operations. `Skiplist` uses parallel linked lists and has an efficiency similar to a btree. We use these three benchmarks to evaluate our tool with applications that have different logics built into them and use two types of crash-consistency mechanisms: low-level (`flushes` and `fences`) and transactional, such as the one PMDK implements (see Section 2.2). For these three examples from PMDK, we use workloads of 1000 and 5000 operations. Specifically for the `Hashmap tx`, we use only a workload of 1000 operations, as using 5000 operations results in an unknown memory allocation error.

Pmemkv is a key-value data store optimized for persistent memory and supports many different storage engines. In our tests, we will use `Cmap` and `Stree`. `Cmap` is a persistent concurrent engine which uses a persistent hashmap and persistent string from PMDK's libpmemobj library. `Stree` is a sorted engine that uses a B+ tree. We chose two very different approaches to broaden our testing. With pmemkv, we use workloads of 1000 and, in some cases, 10000 random write operations.

When obtaining the `execution time`, `CPU usage`, and `space occupied in memory` to test the efficiency of copy-on-write, we resort to PMDK's `Skiplist` benchmark with 2000 operations.

Next, we will present the results from the tests we performed in this work. We start with the tests using different filesystems, namely ext4 and xfs. Then we will go through the results of applying each heuristic we developed to the snapshot generation. To close this section, we will look at how parallelizing the recovery affects efficiency.

6.4 Ext4 with Dax versus xfs with CoW

To evaluate how each filesystem performs with our tool, we look into three metrics: `execution time`, `CPU usage`, and `memory usage`.

6.4.1 Execution time

To analyze execution time, let us look at Figure 6.1. Each line refers to a different scenario. We have two scenarios: xfs with CoW and ext4 with Dax. Each line depicts the tendency for the

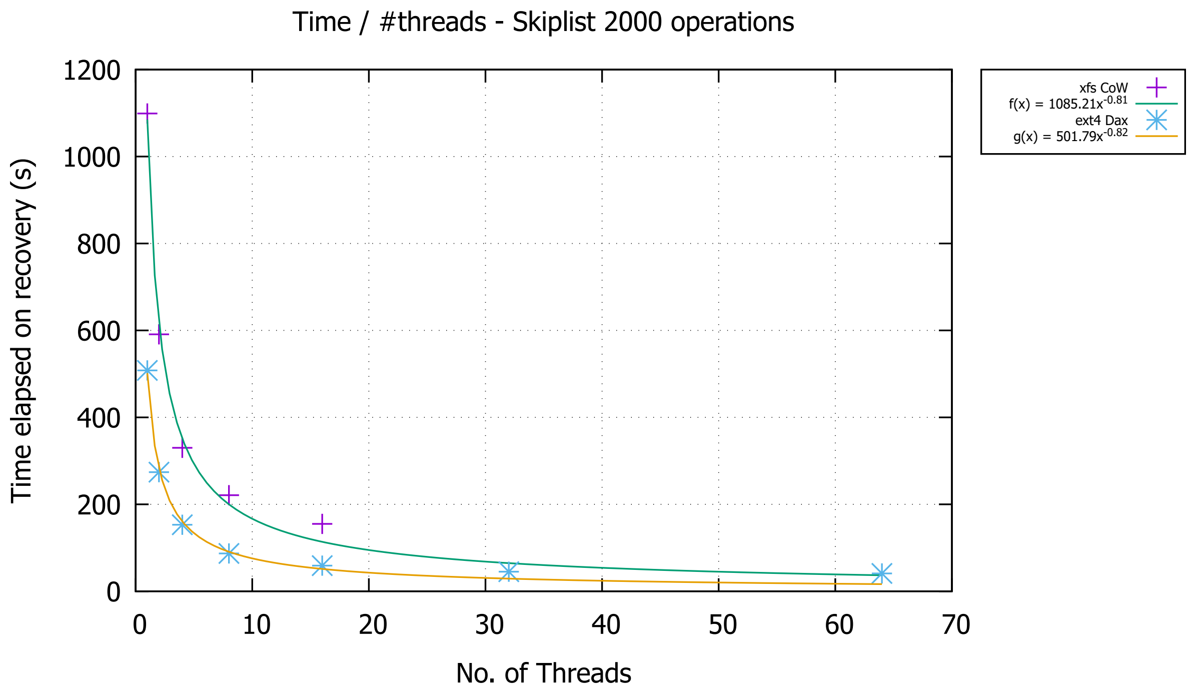


Figure 6.1: Recovery time with PMDK's skiplist 2000 operations with different threads

execution time of the recovery process depending on the number of threads. We can make the following observations:

- Comparing xfs with CoW (green) and ext4 with Dax (blue) lines, we can conclude that using ext4 with Dax is more efficient than xfs with CoW. For every thread count, the time elapsed on recovery is shorter in the case of ext4 with Dax. This is expected since Dax was explicitly created to eliminate a degree of abstraction between the kernel and the memory hardware, increasing the efficiency.

- Looking at the equations of each line obtained through a power regression, we can see that the coefficients on the green and blue lines are very similar. This means that, although in flat values using ext4 with Dax is faster than xfs with CoW, their scaling is almost the same as we increase the number of threads. This is visible by the lines becoming parallel for higher thread counts.

We conclude that using ext4 with Dax results in shorter execution times, but xfs with CoW scales similarly to ext4 with Dax for higher thread counts.

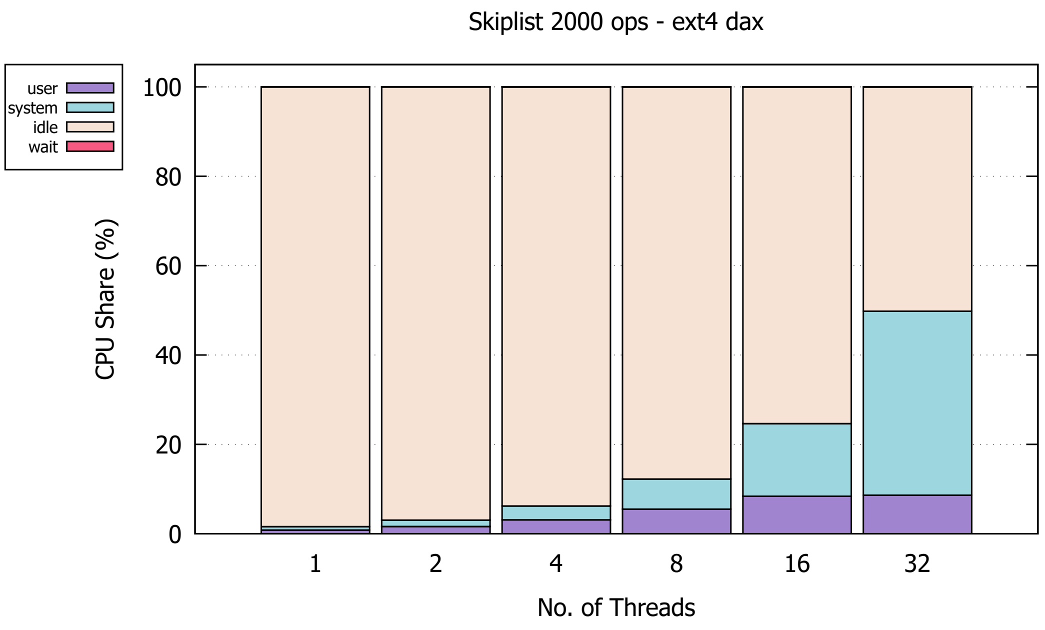
6.4.2 CPU usage

Let us look at Figure 6.2 to understand how both scenarios under analysis (ext4 with Dax and xfs with CoW) perform. We have a bar plot for each scenario, where each bar corresponds to a thread count. We present the CPU share for user, system, idle, and wait for each bar. The higher the user share, the more efficiently a filesystem uses the CPU. On the other hand, the higher the wait, the more contention we have, and it negatively impacts performance. We make the following observations:

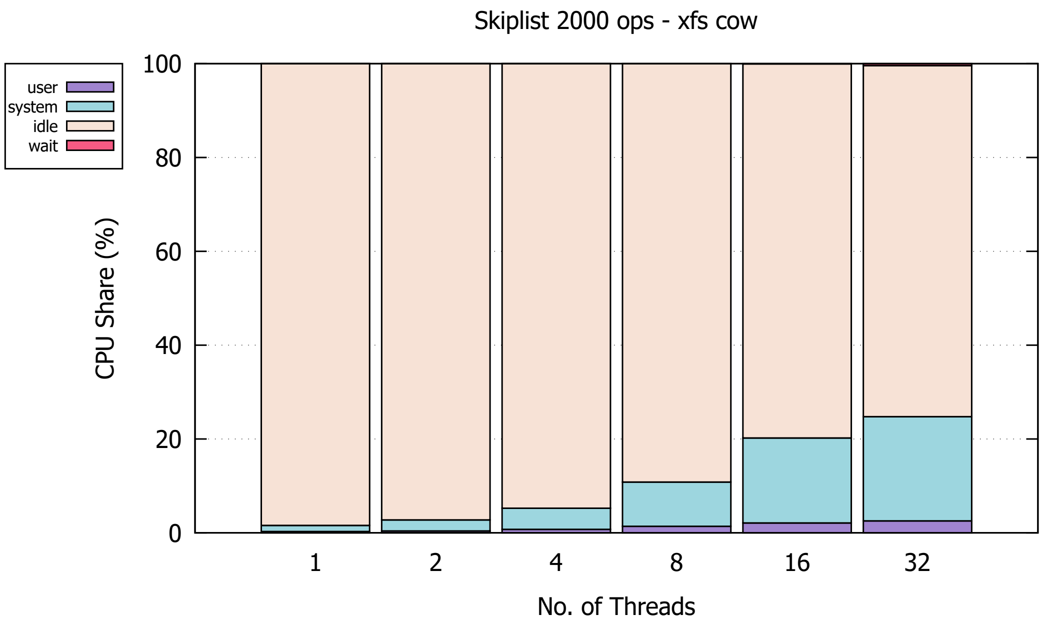
- Looking at Figure 6.2(a), we check that the user share is higher for all thread counts compared to the one in Figure 6.2(b). This means that ext4 with Dax uses the CPU better, increasing efficiency.

- We add a third bar plot depicted in Figure 6.3, which is the same as Figure 6.2(b) but focused on the values for the user share, to visualize the increase better as we increase the thread count. We notice a pattern for both scenarios: the increase in user share diminishes as we increase the threads, meaning we do not have a completely parallelizable scenario. The coefficients in the equations of the lines in Figure 6.1 verify this since they are all greater than -1. A coefficient of -1 would indicate that when we doubled the thread count, the recovery time would halve.

The observations from the presented bar plots confirm our conclusion from the previous section: ext4 with Dax is more efficient but scales similarly to xfs with CoW.



(a) ext4 with Dax.



(b) xfs with CoW.

Figure 6.2: CPU usage for different thread counts using PMDK's skiplist with 2000 operations.

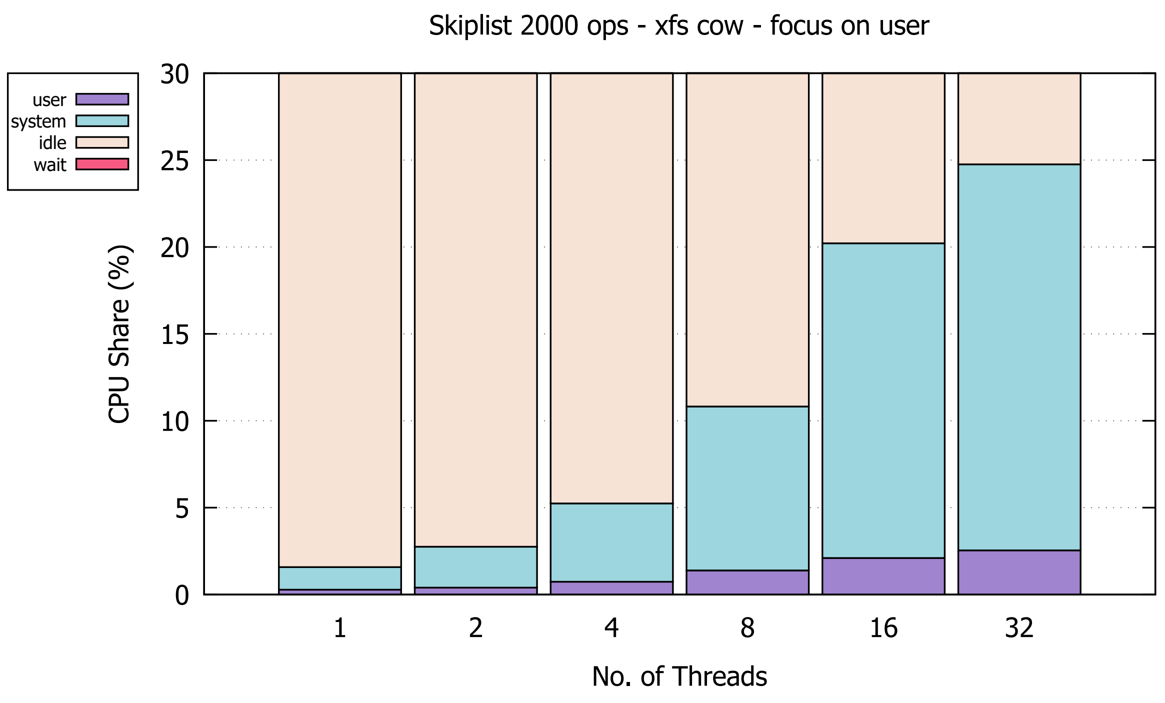


Figure 6.3: CPU usage for different thread counts using PMDK's skiplist with 2000 operations for xfs with CoW (focus on user).

6.4.3 Memory usage

scenario	memory per snapshot (MB)
ext4 with Dax	8.0046
ext4 without Dax	8.0046
xfs with Dax	8.0055
xfs with CoW	0.9890

Table 6.2: Average memory per snapshot in megabytes.

We now present four scenarios: the two we previously mentioned plus ext4 without Dax and xfs with Dax. As previously discussed, the incompatibility between Dax and Cow implies that xfs with Dax does not have CoW. Ext4 without Dax does not have CoW either. To understand the memory usage of each scenario, let us look at Table 6.2. The table presents the average memory a snapshot uses for each scenario. We make the following observations:

- Using Dax has no impact on memory usage since ext4 with Dax and ext4 without Dax have the same average memory usage per snapshot.
- As expected, different filesystems save different metadata for each file, which leads to the small difference between using ext4 with Dax and xfs with Dax.
- Using CoW reduces the average memory consumption by eight times.

We conclude that CoW is very efficient in memory usage, as it reduces the total memory used to about an eighth compared to not using CoW.

6.4.4 Discussion

We have looked into the `execution time`, `CPU usage`, and `memory usage` of the different filesystems with different techniques. We conclude that the slight decrease in recovery time when using ext4 with Dax is not worth it compared to the massive increase in memory efficiency when using xfs with CoW. Therefore we justify the use of the latter in our work. It is worth noting that for some programs, we may reach hundreds of thousands of snapshots, if not more. If we consider a case where we have 100000 snapshots, then, for ext4 with Dax, we would need $100000 \times 8.0046MB = 800460MB \sim 800GB$. Many developers may not have a machine with this much PM, so this option can be immediately excluded in some cases.

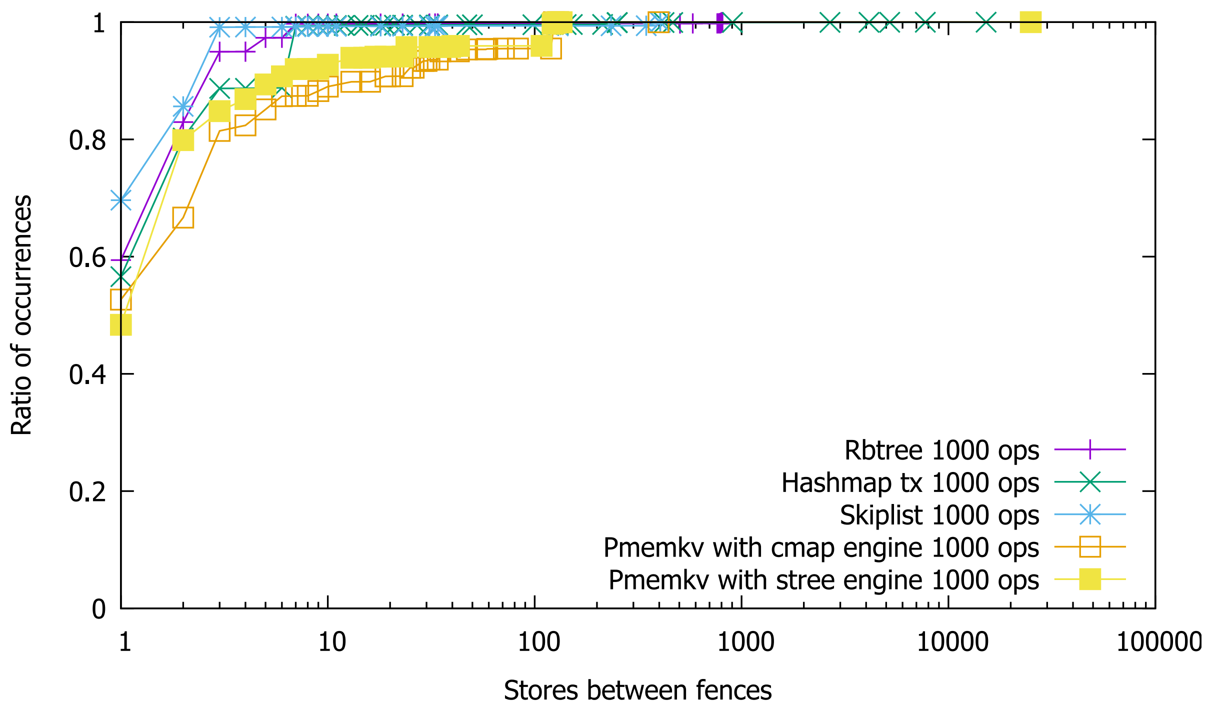


Figure 6.4: Number of written addresses at each segment.

It is worth mentioning that the tests analyzed throughout Section 6.4 were performed at different stages of the tool's implementation to justify our design choices. This means that the final work has different values for the recovery time and CPU usage since the tool's logic has modifications. This, however, does not invalidate any of the conclusions mentioned above since the tests do not regard the tool's performance but how different environments (namely, filesystems with different techniques) affect the tool's efficiency.

6.5 Heuristics

We presented three heuristics in Chapter 4 that would help us reduce the number of snapshots generated through reorderings. We now present our analysis of these heuristics to assess the impact of each.

6.5.1 Heuristic 1 - Limit number of stores between fences

Figure 6.4 shows our analysis as a cumulative distribution function of the number of written addresses at each segment for five different programs. We run each program with 1000 operations to normalize each test. These programs include examples from PMDK's libpmemobj [Intd] and pmemkv [Inta]. We can see that for all tested programs, around 90% of segments contain at most eight written addresses. We can also see that we have some segments that include up to thousands of writes to unique addresses. Figure 6.5 details the distribution if we consider only segments which contain writes to more than eight unique addresses. In this scenario, most segments have between a hundred to a thousand stores with unique addresses. We know that some of these segments typically regard initialization due to the nature of the analyzed programs. We could not confirm this, but we hypothesize that the rest of the segments with higher store counts regard memory allocation. Although we should not discard these cases, it is not feasible to create, for instance, 2^{1000} snapshots since

$$2^{1000} \sim 1.07 \times 10^{301}.$$

At the same time,

$$2^8 = 256,$$

which means that applying this heuristic with a limit of 8 reduces the number of created snapshots by a factor with a magnitude of hundreds while only reducing the explored code by around 10%[1].

To demonstrate that the limit suggested above is universal for programs with more operations, we can look at Figure 6.6. This figure depicts the distribution of the number of stores between segments in two scenarios. In Figure 6.6(a), we use PMDK's Rbtree with 1000 and 5000 operations. In Figure 6.6(b), pmemkv's Stree engine with 1000 and 10000 operations. Both figures show that the lines are almost entirely overlapping, indicating that the distribution of stores in segments is independent of the number of operations.

Table 6.3 demonstrates the reduction in states using this heuristic (with a limit of 8 stores per segment) for different programs. We observe a massive reduction in the number of generated snapshots for all programs. However, the number we obtained is still too big to account for, as it would take a very long time if we were to execute the recovery process on millions of snapshots. This justifies the need for the other heuristics as we have to reduce the number of snapshots further.

[1]We consider a reduction of 10% when dividing the code into segments. Naturally, segments have different store counts. Considering that we are not taking into account the segments with the highest store counts, the reduction is much greater if we look at the code line by line. We still consider this positive since, as explained, these regions of code typically regard initialization or memory allocation, which is not as interesting to the semantics of a program.

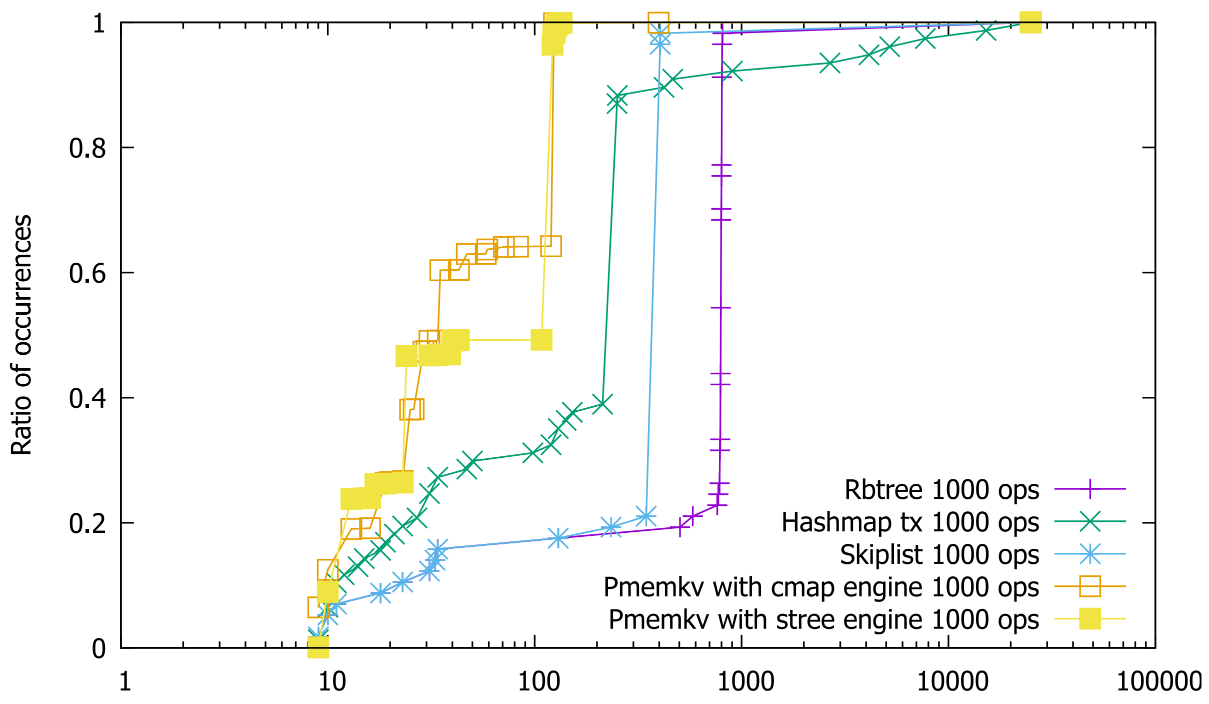
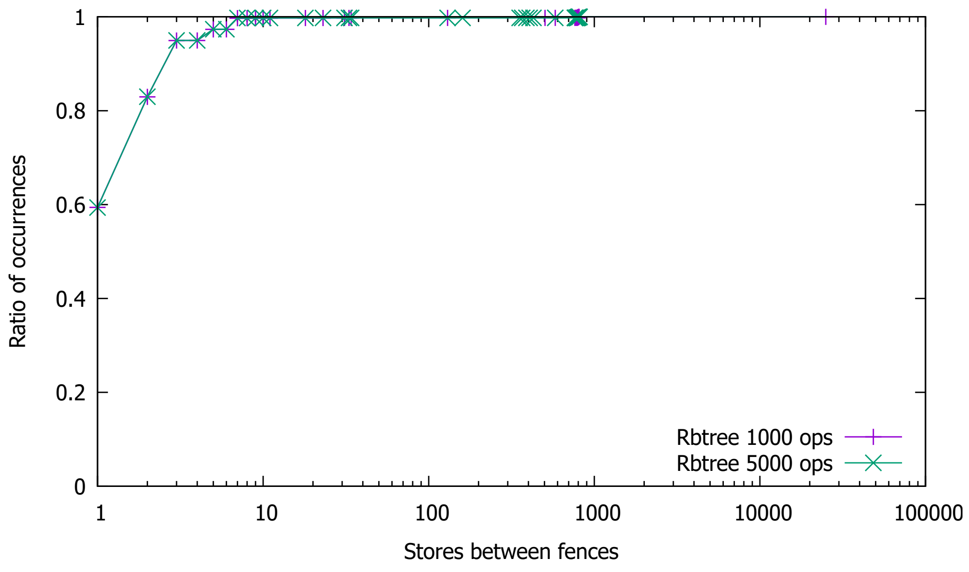


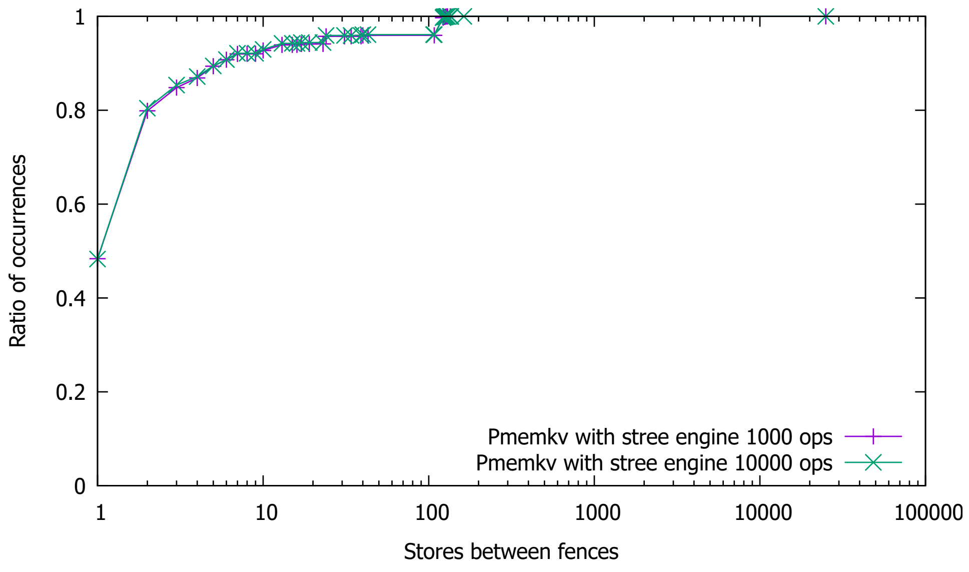
Figure 6.5: Number of written addresses (> 8) at each segment.

program	stores	flushes	snapshots wo/ heuristics	snapshots w/ heuristic 1 (limit 8)
rbtree 1000 ops	95889	20765	$> 4.50 \times 10^{7526}$	4.23×10^{12}
rbtree 5000 ops	380312	104208	$> 4.50 \times 10^{7526}$	4.23×10^{12}
hashmap tx 1000 ops	96271	12074	$> 4.50 \times 10^{7526}$	1.74×10^{19}
skiplist 1000 ops	53191	7405	$> 4.50 \times 10^{7526}$	4.23×10^{12}
skiplist 5000 ops	164768	36882	$> 4.50 \times 10^{7526}$	4.23×10^{12}
pmemkv cmap 1000 ops	150022	13892	$> 4.50 \times 10^{7526}$	4.23×10^{12}
pmemkv stree 1000 ops	137742	15415	$> 4.50 \times 10^{7526}$	1.51×10^{19}

Table 6.3: Snapshots generated with heuristic 1.



(a) PMDK's rbtree



(b) Pmemkv using stree

Figure 6.6: Number of written addresses at each segment.

6.5.2 Heuristic 2 - Limit age for a store to persist

To understand the importance of this heuristic, let us look at Figure 6.7. It presents a cumulative distribution function of the age to persist (up to 20 segments) for stores in the programs we are testing. This figure contains only the programs executed with 1000 operations to normalize the observations. We conclude that for the analyzed programs, around 75% of the stores are persisted in the same segment they are issued (meaning their age to persist is 0). This is in line with the results of the study performed by the authors of PMDebugger [DLCL21]. Additionally, some applications persist 90% of the stores at most five segments from their issue. In Section 4.7.2, we hypothesized that stores with higher ages to persist are connected to memory allocation. Let us look at Figure 6.8 to see how it helps reinforce this idea. This figure shows the same information as the previous one, although regarding the programs with more operations. With more operations and consequently higher instruction count, we notice an increase in this pattern of a higher age to persist. To explain this increase, let us look at the green line with crosses in both Figure 6.7 and Figure 6.8. In the first, the line starts between the values 0.7 and 0.8. This means that around 75% of the stores take 0 segments to persist. In the second figure, however, the line starts at around 0.3, which means that only 30% of the stores persist immediately. This means that when we consider the same program with more operations, we have a higher share of stores that take at least 20 segments to persist. Naturally, the programs we are analyzing in the second figure need to allocate more memory to be able to process the requested number of operations. As such, it is logical to conclude that having more memory allocation contributes towards increasing the number of stores that take longer to persist, justifying the difference between the two figures.

We have seen how the number of possible states at the end of a segment scales exponentially, so pruning stores that are not as interesting is a great way of increasing the efficiency of our tool without necessarily sacrificing completeness. Using this heuristic is especially helpful in eliminating stores that are never guaranteed to persist from the generation of snapshots through reorderings. Again, we can warn the user of these cases, and it is up to the user to decide whether the stores that took too long to persist are intentional behaviour.

To look at the impact of applying heuristic 2 with a limit of age to persist of 5, let us look at Table 6.4. This table is very similar to the one introduced in the previous section with heuristic 1. The difference is that we now compare the number of generated snapshots using only heuristic 1 and using both heuristic 1 and heuristic 2 together. We can observe how the different nature of each program results in very different numbers of snapshots. For instance, testing with PMDK's skiplist with 1000 operations, we only increase from 53191 snapshots (equivalent to the

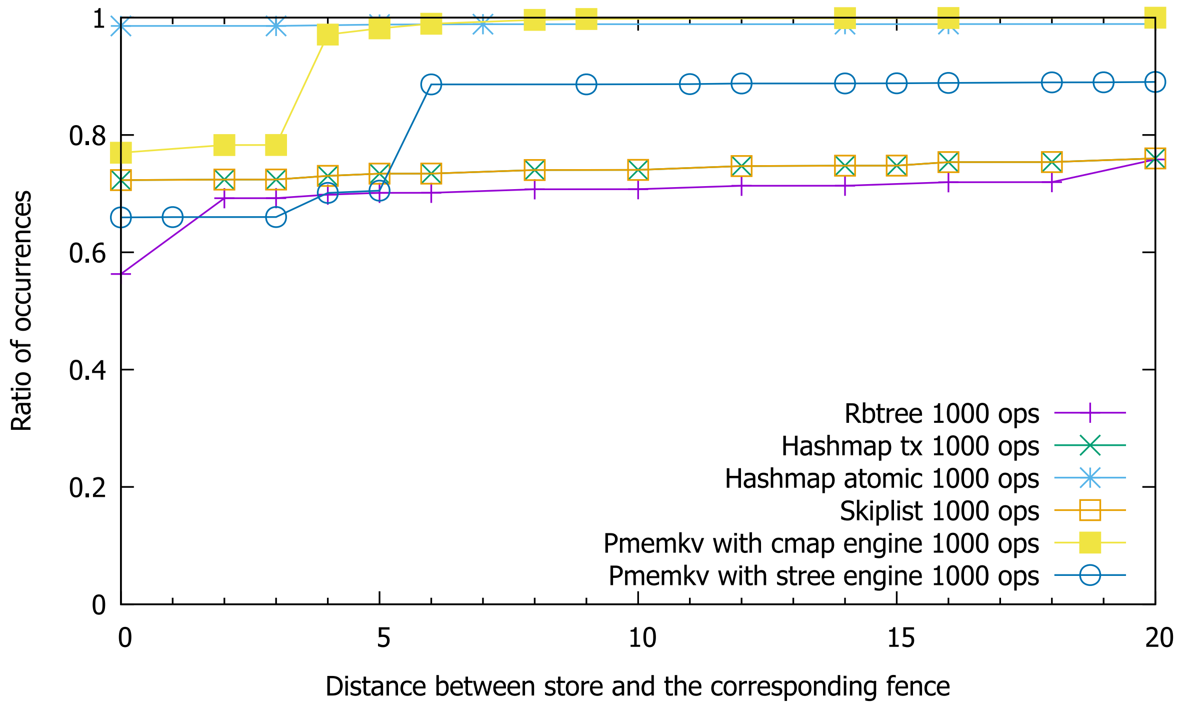


Figure 6.7: Age to persist up to 20 segments (programs with 1000 operations).

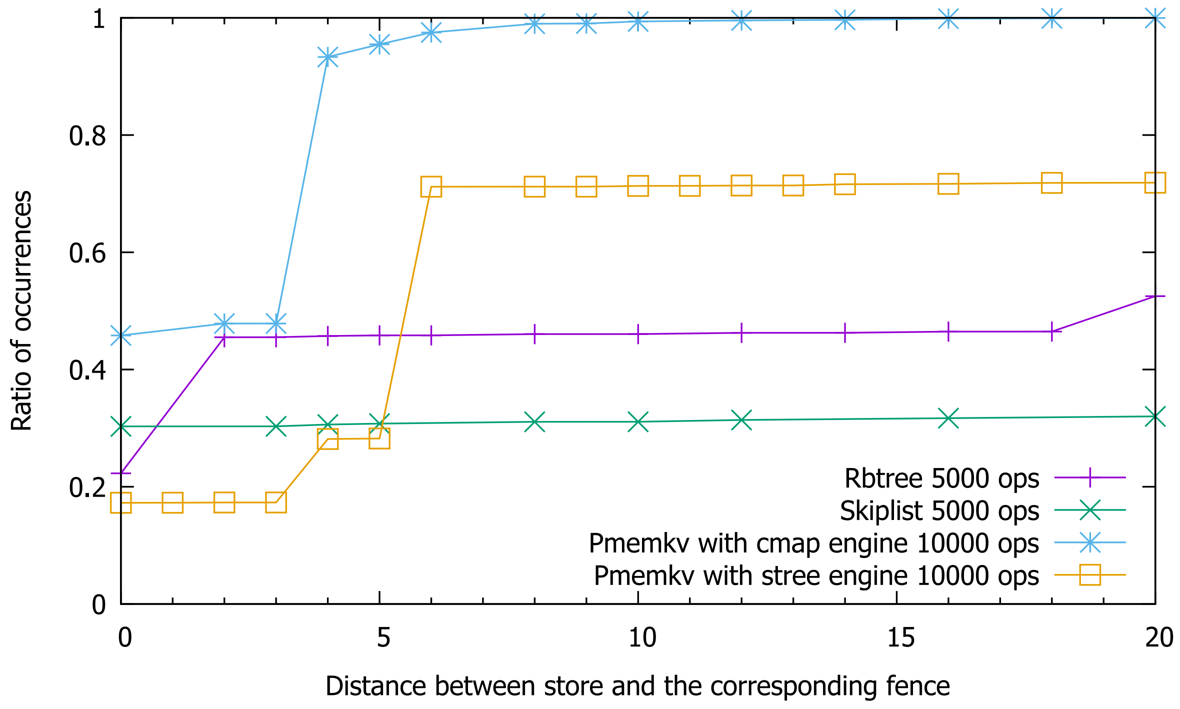


Figure 6.8: Age to persist up to 20 segments.

program	stores	flushes	snapshots w/ heuristic 1 (limit 8)	snapshots w/ heuristics 1 and 2 (limits 8 and 5)
rbtree 1000 ops	95889	20765	4.23×10^{12}	99111
rbtree 5000 ops	380312	104208	4.23×10^{12}	396712
hashmap tx 1000 ops	96271	12074	1.74×10^{19}	97140
skiplist 1000 ops	53191	7405	4.23×10^{12}	53220
skiplist 5000 ops	164768	36882	4.23×10^{12}	164981
pmemkv cmap 1000 ops	150022	13892	4.23×10^{12}	157633
pmemkv stree 1000 ops	137742	15415	1.51×10^{19}	270265

Table 6.4: Snapshots generated with heuristic 2.

number of writes) to 53220 snapshots. This is an increase of 29 snapshots or 0.05%. We observe something else entirely when we look at pmemkv's with stree engine and 1000 operations. We go from 137742 snapshots to 270265. This corresponds to an increase of 96.21% in generated snapshots.

6.5.3 Heuristic 3 - Reordering restrictions with timestamps

Due to the complexity associated with implementing this heuristic, we are not able to present results for it in the time span of this work. However, we comment here on the expected results. This heuristic imparts ordering restrictions to the values each address may hold at a specific state. It does not prune any states regarding program analysis like the previous two heuristics. This means that applying the logic of this heuristic is guaranteed to increase the time it takes for our tool to instrument the application (as it needs to check more conditions). However, it is also guaranteed to reduce the number of generated snapshots. In an extreme case, we may observe a reduction of 0 snapshots. This can happen in the case of a program that, for instance, only uses flush operations that can be reordered, such as `clflushopt` and `clwb`.

6.6 Parallelization of the recovery process

The recovery process is a fundamental part of testing an application. It is the process where we run the recovery program on each generated snapshot. We have seen that specific programs

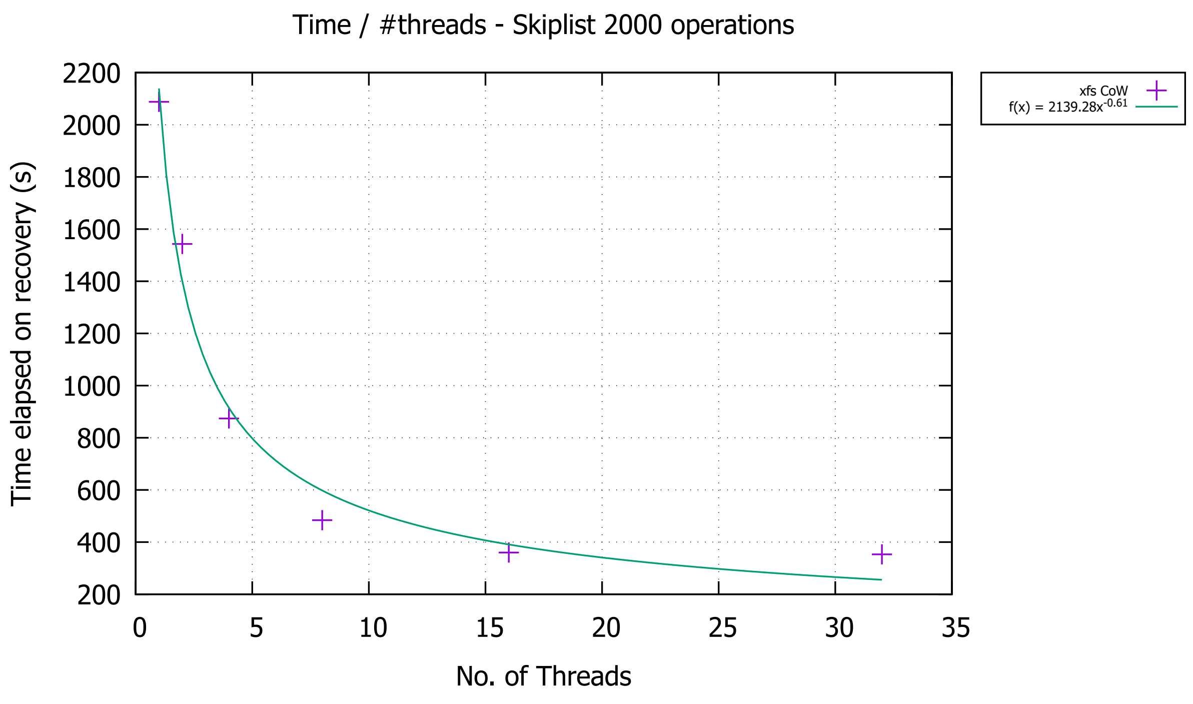


Figure 6.9: Time to recover for different thread counts - previous machine.

can generate hundreds of thousands of snapshots, even when using heuristics to prune certain states (see Table 6.4). As such, we wanted to ensure that we could make the recovery process as efficient as possible to reduce the overall testing time.

Section 5.1.4 detailed how we parallelize the recovery process to achieve the goal of efficiency. We could theoretically achieve perfect parallelization since the snapshots have no dependencies. This means that if we doubled the number of threads, the recovery process would halve. As we will see in this section, that is not the case, and we performed a series of tests using different environments to understand what was stopping us from achieving said perfect parallelization.

Note that the instrumentation logic was being implemented at the same time we performed the following tests. This means the values presented in the following sections may differ even when using the same programs with the same workloads. This does not change the point being made, as we are concerned with the scalability of parallelizing the recovery process and not the time it takes to finish executing.

6.6.1 Normal test case

Figure 6.9 shows the results of one of our initial tests of the time to execute recovery. In a perfectly parallelizable scenario, the coefficient of the power regression would be -1. In this case, we have -0.61, which is far from ideal. It is worth mentioning that we performed this test on a different machine than the one specified in Section 6.1. This is because the machine we used to perform our initial tests had a critical failure and remained unusable until the completion of this work. This is important to explain the following results.

6.6.2 Taskset

At this point, we had to perform our tests on a new machine whose specifications are the ones detailed in Section 6.1. The results of this test are depicted in Figure 6.1. It is interesting to notice that under the same conditions as the previous test, we now have a coefficient of -0.81 instead of -0.61. Using different machines will almost always net different results, but we did not expect such a high difference in scaling, represented by the power regression coefficient.

Our first hypothesis to explain this was the version of xfs being used. The first machine had an older version, so we performed the same test with this older version on the new machine, but the results were the same.

Because we were not satisfied with a coefficient of -0.81, we tried repeating the test with the use of taskset [Kerb]. Taskset allows pinning the execution of a process to a physical CPU core. We reason that this would stabilize the process scheduling and improve the efficiency of the recovery process. Figure 6.10 depicts the results of this test with different filesystems and techniques (Dax and CoW). The green line corresponds to xfs with CoW, and the orange line to xfs with CoW and taskset. We conclude that taskset lowers the overall recovery time but scales worse with higher thread counts, as the coefficient is further from -1.

We now shift our focus to Figure 6.11. This figure presents the results from the same tests as Figure 6.10, but we now include the recovery times when using xfs with CoW with more than 16 threads. We only present these values at this point in the evaluation to make it easier for the reader to understand our line of thought. The behaviour for thread counts higher than 16 becomes unstable. We did not find a reason for this, especially since we had a monotonically decreasing recovery time in the previous machine up to 32 threads, as seen in Figure 6.9.

Figure 6.12 shows the CPU share of the tests using taskset for xfs with CoW. We observe that, for thread counts higher than 16, the share of `wait` is not neglectable. This means that the filesystem is not handling this many threads well, as it creates contention. This may be due to several reasons. Our last hypothesis is an asymmetry in the processor's architecture, which we try to explore as explained in the next section.

We now present a view of the obtainable speedups in the overall execution and recovery process. The speedup values are displayed in Table 6.5. A speedup of 2 means that the tool takes $\frac{1}{2}$ the time to execute. The last line of the table lets us conclude that we can soundly reach

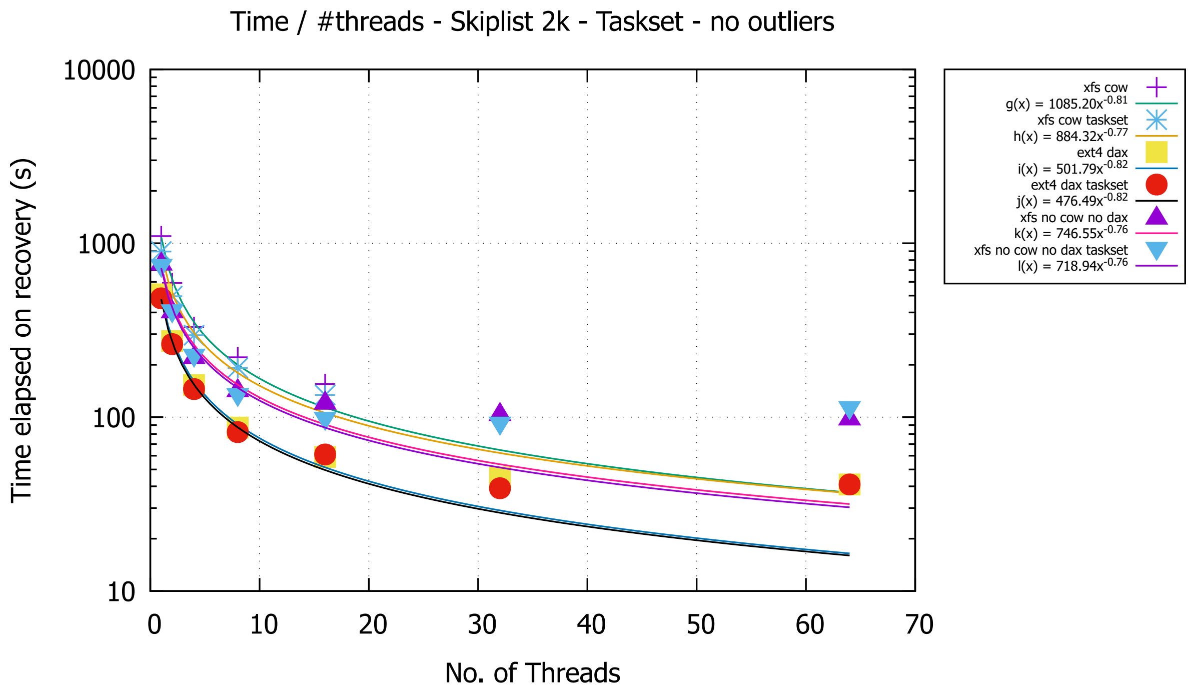


Figure 6.10: Time to recover for different thread counts using taskset and limiting xfs with CoW to 16 threads.

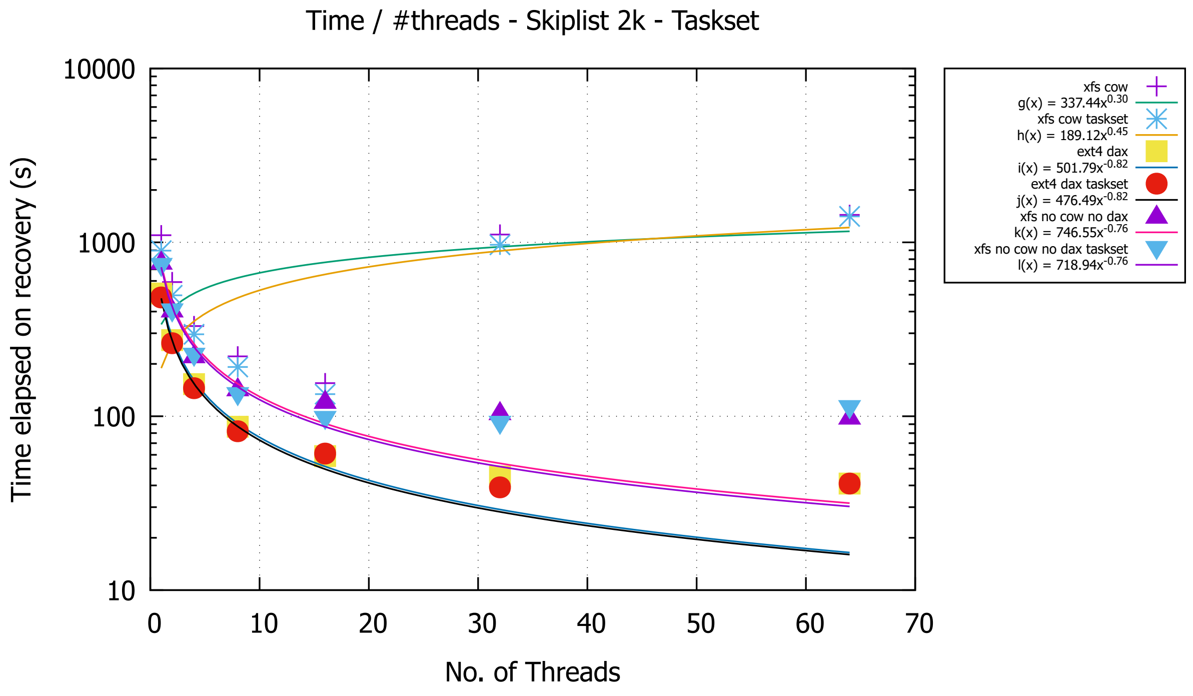


Figure 6.11: Time to recover for different thread counts using taskset

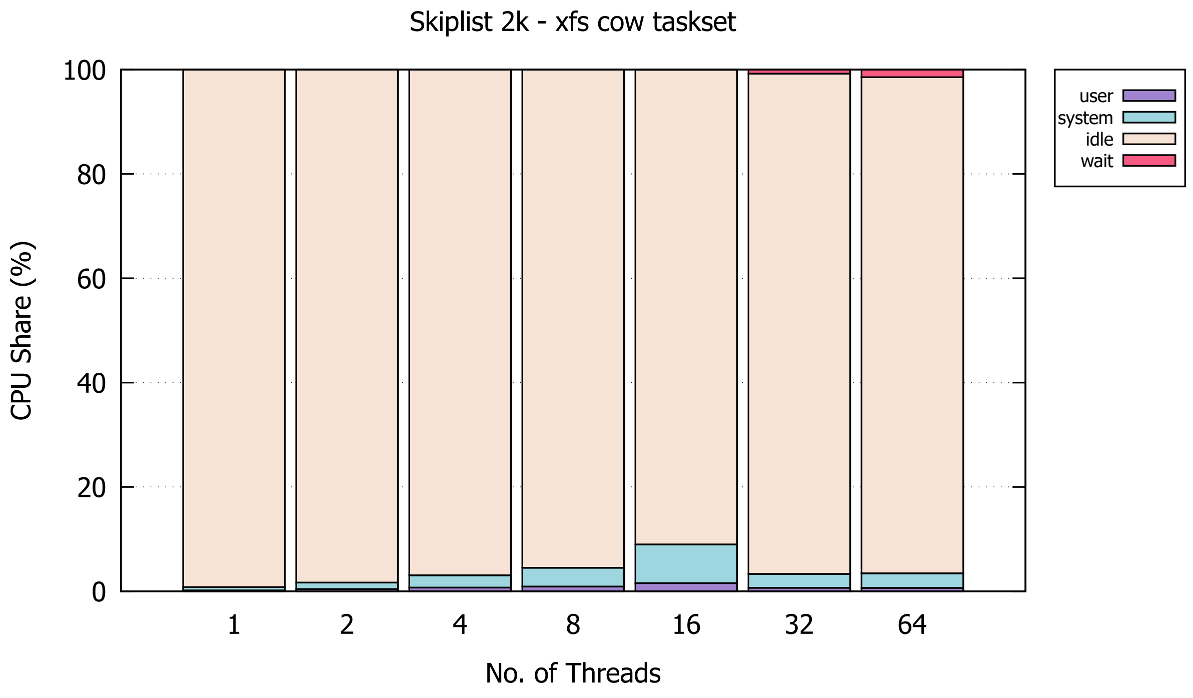


Figure 6.12: CPU usage for different thread counts using PMDK's skiplist with 2000 operations and taskset.

a speedup of around 7 in the recovery process using 16 threads. Logically, the tool speedup is always smaller than the recovery speedup since the instrumentation and creation of snapshots are single-threaded.

Thread count	Tool speedup	Recovery speedup
1	1.00	1.00
2	1.83	1.86
4	3.21	3.33
8	4.65	4.97
16	6.41	7.09

Table 6.5: Recovery process speedup from different thread counts.

6.6.3 Numactl

To explore whether the contention we observed in the previous section is due to processor asymmetry, we resort to a specific tool - numactl [Kle]. This tool allows pinning the execution of a process to a physical CPU core just like taskset, but it also allows limiting the allocation

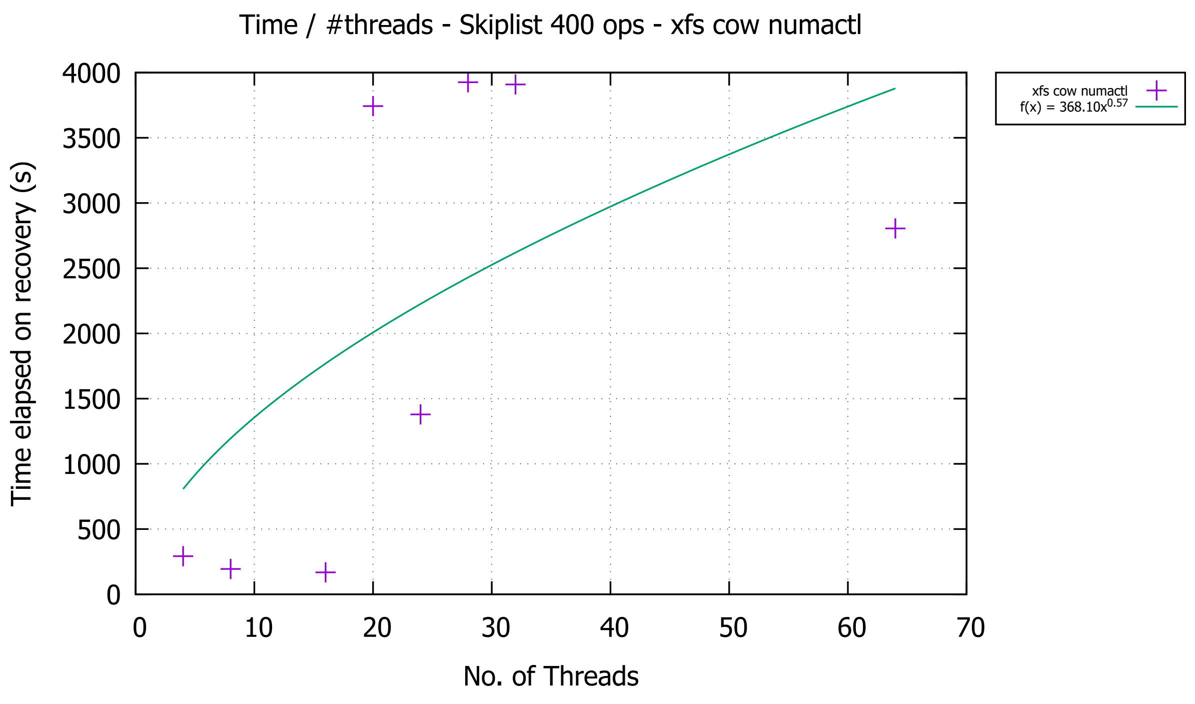


Figure 6.13: Time to recover for different thread counts using numactl

of memory to a processor's node. Since our processor has two nodes, each with 64 threads, and we execute our tests with a maximum of 64 threads, we pin the recovery process to one of the nodes to minimize the communication overhead within the processor.

Figure 6.13 shows our results with this approach which remain the same, with unexpected behaviour for thread counts higher than 16. Due to time constraints, we leave exploring this issue further to future work.

6.7 Recovery time estimation

The results in Figure 6.11 correspond to applying the recovery process to 6298 snapshots. We can use this information to estimate how much time it would take to recover the number of snapshots for each program in Table 6.4. With the results from the previous section, we use the time it took to recover 6298 snapshots with 16 threads, which is 155 seconds, as our base. Table 6.6 shows that for the considered example programs, all estimations point to a maximum of three hours to run the recovery process. This is considered an acceptable time frame to test a program. Note that we do not include the instrumentation and snapshot creation in this consideration, but the most intensive part of the testing process is always the recovery. Even if we assumed an instrumentation time equivalent to the recovery, we would still be able to test these applications in an acceptable time window.

Program	Number of snapshots	Estimated recovery time
rbtree 1000 ops	99111	40 min 39 sec
rbtree 5000 ops	396712	162 min 43 sec
hashmap tx 1000 ops	97140	39 min 51 sec
skiplist 1000 ops	53220	21 min 50 sec
skiplist 5000 ops	164981	67 min 40 sec
pmemkv cmap 1000 ops	157633	64 min 40 sec
pmemkv stree 1000 ops	270265	110 min 51 sec

Table 6.6: Estimation of recovery times.

6.8 Discussion

We will now briefly reiterate our results and discuss how they address the proposed goals for this work. We observed that:

1. We are able to resort to copy-on-write as a space-efficient way to store all the snapshots in memory without significantly sacrificing efficiency;
2. We implement a series of heuristics that transform a virtually infinite number of snapshots into a manageable collection of these snapshots;
3. Our parallelization of the recovery process significantly reduces the time to execute it. We demonstrate that we can stably exploit this behaviour up to 16 threads, speeding up the recovery process to 7 times.

Our main goal was to create a tool that debugs PM applications and is entirely automatic and efficient. We achieve automation by not requiring the user to recompile any code or develop custom annotations and oracles. Although we do not present actual values of the time our final tool takes to execute, we present several tests of our tool in the intermediate phases of development. These tests display execution times much lower than our defined limit of 12 hours so that the tool can be integrated into a development pipeline. We also present the number of snapshots we generate for some programs, comparable to the number of snapshots we use in some of the tests we present in this chapter. These considerations allow us to conclude that our tool can test many PM programs within the defined time limit, which meets the goal of efficiency.

Chapter 7

Conclusions

This chapter details this work's contribution to the state of the art by presenting the key points that make it stand out. It ends with a section listing the shortcomings and what the next steps in the future are to improve this work.

PM offers excellent performance but lacks guarantees regarding the persistence of stores. To help ensure crash-consistency, we have flush and fence instructions. Using these instructions is not trivial and writing correct PM programs becomes quite challenging. Therefore, several methods and tools have been proposed to detect bugs in PM programs.

When analyzing the available tools in the state of the art, we discovered several drawbacks. Some tools are automatic, but their efficiency is poor [LDK+14, GXD21]. Other tools have higher efficiency but require the user to recompile code or create custom oracles and annotations [DLCL21, LSW+20, LWZ+19, NRSQ20]. A few tools test only a limited set of PM applications [LDK+14, FKP+21].

We set the goal of developing a PM debugging tool that did not have any of the previously mentioned drawbacks. This work presents a tool that is automatic, application-agnostic, and efficient. To achieve this goal, we make the following contributions:

1. We develop an efficient shadow PM bookkeeping structure that allows us to keep track of the individual addresses in PM. This, in turn, allows our tool to test any PM application.

2. We explore the concept of copy-on-write to be able to execute the application a single time and have all generated snapshots in memory simultaneously.

3. We create three heuristics that allow us to have a manageable collection of snapshots when considering the possible reorderings of instructions by the hardware. If we did not have these heuristics, we would have infinite testing times for certain applications [LDK+14].

4. Focusing on parallelizing the recovery allows us to reduce the time it takes to test an application 6 to 7 times, compared to a linear recovery process.

All the considerations listed contribute to the efficiency of the tool. Using our heuristics boosts completeness as we can explore more states than those observed from the program order. We offer automation since we do not require the user to recompile any code or develop annotations or custom oracles.

7.1 Future Work

The previous chapter mentions some ideas we did not fully explore in this work due to time constraints. We reinforce those ideas here, explaining their projected contributions and other new ideas that can improve this work.

1. We were only able to stably use up to 16 threads in the parallelization of the recovery. It would be a great efficiency boost if we could explore how to parallelize the recovery further to use many more threads efficiently. Currently, we obtain a speedup of 6 to 7 times using 16 threads. If we could use 64 or even 100 threads, we could look at speedups up to 40 (assuming linear scalability).

2. We presented the rationale for the third heuristic that regards enforcing ordering constraints due to using `flush` instructions that cannot be reordered. The next step would be fully implementing this into our logic and performing the following tests. First, test the number of snapshots generated using this heuristic along with the other two. Second, analyze the tradeoff between adding this logic to the instrumentation and shortening the recovery since we have fewer snapshots when using this heuristic. We could have an extreme case where the reduction of snapshots is neglectable, and we observe a higher overhead in the instrumentation than the reduction we get in the recovery process.

3. In Chapter 6, we presented the number of generated snapshots using the first and second heuristics. We considered only a specific limit for each heuristic, which we considered ideal from our analysis of the PM programs. It would add value to this work to analyze how altering these limits can impact the number of generated snapshots for each type of PM program. This can later lead to the creation of profiles for different types of PM programs. These would allow our tool to suggest a limit for each heuristic. These limits would depend on the type of program being tested.

4. Picking up on the idea of the previous point, let us consider that we have an extensive collection of data. This data includes how different limits for each heuristic affect the number of generated snapshots. Then, with the help of artificial intelligence, we could increase the flexibility of our tool. For instance, the user could provide a time limit to test an application, and the tool would be able to choose limits for each heuristic that would cause the testing process to finish executing under that time limit. This would maximize the completeness of the tool in the time limit given.

5. This work focuses on correctness bugs, meaning we do not detect any performance bugs, as it is not possible by using the recovery process as the bug-finding oracle. Since we keep track of the states of memory addresses, we could build upon our instrumentation logic and detect some of these bugs, as a trace analysis tool would be able to do.

Bibliography

[BC03] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly and Associates, 2^{nd} edition, 2003. ISBN: 0-596-00213-0.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *USENIX Association*, 2008.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* Massachusetts Institute of Technology, 3^{rd} edition, 2009. ISBN: 978-0-262-03384-8.

[DLCL21] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. *ACM Transactions on Programming Languages and Systems*, April 2021.

[FKP$^+$21] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. *ACM, New York, NY*, October 2021.

[GXD21] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. *ACM Transactions on Programming Languages and Systems*, April 2021.

[Inta] Intel. Key/value datastore for persistent memory. `https://github.com/pmem/pmemkv`.

[Intb] Intel. Persistent memory development kit (pmdk). `https://pmem.io/pmdk/`.

[Intc] Intel. *PIN Controlling and Initializing.*

[Intd] Intel. Pmdk man page libpmemobj. `https://pmem.io/pmdk/manpages/linux/v1.3/libpmemobj.3.html`.

[Int12] Intel. Pin - a dynamic binary instrumentation tool, 2012.

[Int15] Intel. An introduction to pmemobj (part 2) - transactions. `https://pmem.io/2015/06/15/transactions.html`, June 2015.

[Int18] Intel. C++ transactions for persistent memory programming. `https://www.intel.com/content/www/us/en/developer/articles/technical/c-plus-plus-transactions-for-persistent-memory-programming.html`, 2018.

[Kera] Michael Kerrisk. *cp - Linux manual page.*

[Kerb] Michael Kerrisk. *taskset - Linux manual page.*

[Kle] Andi Kleen. *numactl - Linux man page.*

[LDK$^+$14] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. *USENIX Association*, June 2014.

[LSW$^+$20] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. *ACM Transactions on Programming Languages and Systems*, March 2020.

[LWZ$^+$19] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. *ACM Transactions on Programming Languages and Systems*, April 2019.

[NRSQ20] Ian Neal, Ben Reeves, Ben Stoler, and Andrew Quinn. Agamotto: How persistent is your persistent memory application? *USENIX Association*, November 2020.

[Ora] Oracle. Total store ordering (tso). `https://docs.oracle.com/cd/E19455-01/805-7378/hwovr-16/index.html`.

[PADAD05] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. *USENIX Association*, April 2005.

[pme] pmemio. What is direct-access (dax)? `https://kb.pmem.io/faq/100000008-What-is-DAX/`.

[Rud17] Andy Rudoff. Persistent memory programming. *;login:*, 42(2), 2017.

[RWNV20] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the intel-x86 architecture. *Proceedings of the ACM on Programming Languages*, 4(11), January 2020.

[Viz63] V. G Vizing. The cartesian product of graphs. *Vyčisl. Sistemy*, 9, 1963.